18 artist Module 535
19 Axes class 555
20 axis and tick API 793
21 backends 913
22 cbook 953
23 cm (colormap) 971
24 collections 975
25 colorbar 1145
26 colors 1151
27 container 1165
28 dates 1167
29 dviread 1181
30 figure 1185
31 finance 1207
32 font_manager 1219
33 gridspec 1227
34 image 1231
35 legend and legend_handler 1237
36 lines 1245
37 markers 1255
38 mathtext 1259
39 mlab 1279
40 offsetbox 1313
41 patches 1325
42 path 1367
43 patheffects 1375
44 projections 1379
Part I

User’s Guide
INTRODUCTION

Matplotlib is a library for making 2D plots of arrays in Python. Although it has its origins in emulating the MATLAB® graphics commands, it is independent of MATLAB, and can be used in a Pythonic, object oriented way. Although Matplotlib is written primarily in pure Python, it makes heavy use of NumPy and other extension code to provide good performance even for large arrays.

Matplotlib is designed with the philosophy that you should be able to create simple plots with just a few commands, or just one! If you want to see a histogram of your data, you shouldn’t need to instantiate objects, call methods, set properties, and so on; it should just work.

For years, I used to use MATLAB exclusively for data analysis and visualization. MATLAB excels at making nice looking plots easy. When I began working with EEG data, I found that I needed to write applications to interact with my data, and developed an EEG analysis application in MATLAB. As the application grew in complexity, interacting with databases, http servers, manipulating complex data structures, I began to strain against the limitations of MATLAB as a programming language, and decided to start over in Python. Python more than makes up for all of MATLAB’s deficiencies as a programming language, but I was having difficulty finding a 2D plotting package (for 3D VTK more than exceeds all of my needs).

When I went searching for a Python plotting package, I had several requirements:

- Plots should look great - publication quality. One important requirement for me is that the text looks good (antialiased, etc.)
- Postscript output for inclusion with TeX documents
- Embeddable in a graphical user interface for application development
- Code should be easy enough that I can understand it and extend it
- Making plots should be easy

Finding no package that suited me just right, I did what any self-respecting Python programmer would do: rolled up my sleeves and dived in. Not having any real experience with computer graphics, I decided to emulate MATLAB’s plotting capabilities because that is something MATLAB does very well. This had the added advantage that many people have a lot of MATLAB experience, and thus they can quickly get up to steam plotting in python. From a developer’s perspective, having a fixed user interface (the pylab interface) has been very useful, because the guts of the code base can be redesigned without affecting user code.

The Matplotlib code is conceptually divided into three parts: the *pylab interface* is the set of functions provided by matplotlib.pylab which allow the user to create plots with code quite similar to MATLAB.
figure generating code (Pyplot tutorial). The Matplotlib frontend or Matplotlib API is the set of classes that do the heavy lifting, creating and managing figures, text, lines, plots and so on (Artist tutorial). This is an abstract interface that knows nothing about output. The backends are device-dependent drawing devices, aka renderers, that transform the frontend representation to hardcopy or a display device (What is a backend?). Example backends: PS creates PostScript® hardcopy, SVG creates Scalable Vector Graphics hardcopy, Agg creates PNG output using the high quality Anti-Grain Geometry library that ships with Matplotlib, GTK embeds Matplotlib in a Gtk+ application, GTKAgg uses the Anti-Grain renderer to create a figure and embed it in a Gtk+ application, and so on for PDF, WxWidgets, Tkinter, etc.

Matplotlib is used by many people in many different contexts. Some people want to automatically generate PostScript files to send to a printer or publishers. Others deploy Matplotlib on a web application server to generate PNG output for inclusion in dynamically-generated web pages. Some use Matplotlib interactively from the Python shell in Tkinter on Windows™. My primary use is to embed Matplotlib in a Gtk+ EEG application that runs on Windows, Linux and Macintosh OS X.
There are many different ways to install matplotlib, and the best way depends on what operating system you are using, what you already have installed, and how you want to use it. To avoid wading through all the details (and potential complications) on this page, there are several convenient options.

### 2.1 Installing pre-built packages

#### 2.1.1 Most platforms: scientific Python distributions

The first option is to use one of the pre-packaged python distributions that already provide matplotlib built-in. The Continuum.io Python distribution (Anaconda or miniconda) and the Enthought distribution (Canopy) are both excellent choices that “just work” out of the box for Windows, OSX and common Linux platforms. Both of these distributions include matplotlib and lots of other useful tools.

#### 2.1.2 Linux: using your package manager

If you are on Linux, you might prefer to use your package manager. matplotlib is packaged for almost every major Linux distribution.

- Debian/Ubuntu: `sudo apt-get install python-matplotlib`
- Fedora/Redhat: `sudo yum install python-matplotlib`

#### 2.1.3 Mac OSX: using pip

If you are on Mac OSX you can probably install matplotlib binaries using the standard Python installation program pip. See *Installing OSX binary wheels*.

#### 2.1.4 Windows

If you don’t already have Python installed, we recommend using one of the scipy-stack compatible Python distributions such as WinPython, Python(x,y), Enthought Canopy, or Continuum Anaconda, which have matplotlib and many of its dependencies, plus other useful packages, preinstalled.

For standard Python installations, install matplotlib using pip:
Matplotlib, Release 2.0.2

```
python -m pip install -U pip setuptools
python -m pip install matplotlib
```

In case Python 2.7 or 3.4 are not installed for all users, the Microsoft Visual C++ 2008 (64 bit or 32 bit for Python 2.7) or Microsoft Visual C++ 2010 (64 bit or 32 bit for Python 3.4) redistributable packages need to be installed.

Matplotlib depends on Pillow for reading and saving JPEG, BMP, and TIFF image files. Matplotlib requires MiKTeX and GhostScript for rendering text with LaTeX. FFmpeg, avconv, mencoder, or ImageMagick are required for the animation module.

The following backends should work out of the box: agg, tkagg, ps, pdf and svg. For other backends you may need to install pycairo, PyQt4, PyQt5, PySide, wxPython, PyGTK, Tornado, or GhostScript.

TkAgg is probably the best backend for interactive use from the standard Python shell or IPython. It is enabled as the default backend for the official binaries. GTK3 is not supported on Windows.

The Windows wheels (*.whl) on the PyPI download page do not contain test data or example code. If you want to try the many demos that come in the matplotlib source distribution, download the *.tar.gz file and look in the examples subdirectory. To run the test suite, copy the lib\matplotlib\tests and lib\mpl_toolkits\tests directories from the source distribution to sys. prefix\Lib\site-packages\matplotlib and sys. prefix\Lib\site-packages\mpl_toolkits respectively, and install nose, mock, Pillow, MiKTeX, GhostScript, ffmpeg, avconv, mencoder, ImageMagick, and Inkscape.

### 2.2 Installing from source

If you are interested in contributing to matplotlib development, running the latest source code, or just like to build everything yourself, it is not difficult to build matplotlib from source. Grab the latest tar.gz release file from the PyPI files page, or if you want to develop matplotlib or just need the latest bugfixed version, grab the latest git version Source install from git.

The standard environment variables CC, CXX, PKG_CONFIG are respected. This means you can set them if your toolchain is prefixed. This may be used for cross compiling.

```
export CC=x86_64-pc-linux-gnu-gcc
export CXX=x86_64-pc-linux-gnu-g++
export PKG_CONFIG=x86_64-pc-linux-gnu-pkg-config
```

Once you have satisfied the requirements detailed below (mainly python, numpy, libpng and freetype), you can build matplotlib.

```
cd matplotlib
python setup.py build
python setup.py install
```

We provide a setup.cfg file that goes with setup.py which you can use to customize the build process. For example, which default backend to use, whether some of the optional libraries that matplotlib ships with are installed, and so on. This file will be particularly useful to those packaging matplotlib.
If you have installed prerequisites to nonstandard places and need to inform matplotlib where they are, edit setupext.py and add the base dirs to the basedir dictionary entry for your sys.platform. e.g., if the header to some required library is in /some/path/include/someheader.h, put /some/path in the basedir list for your platform.

### 2.2.1 Build requirements

These are external packages which you will need to install before installing matplotlib. If you are building on OSX, see Building on OSX. If you are building on Windows, see Building on Windows. If you are installing dependencies with a package manager on Linux, you may need to install the development packages (look for a “-dev” postfix) in addition to the libraries themselves.

#### Required Dependencies

- **python 2.7, 3.4, 3.5 or 3.6**  Download python.
- **numpy 1.7.1 (or later)**  array support for python (download numpy)
- **setuptools**  Setuptools provides extensions for python package installation.
- **dateutil 1.1 or later**  Provides extensions to python datetime handling. If using pip, easy_install or installing from source, the installer will attempt to download and install python_dateutil from PyPI.
- **pyparsing**  Required for matplotlib’s mathtext math rendering support. If using pip, easy_install or installing from source, the installer will attempt to download and install pyparsing from PyPI.
- **libpng 1.2 (or later)**  library for loading and saving PNG files (download). libpng requires zlib.
- **pytz**  Used to manipulate time-zone aware datetimes. https://pypi.python.org/pypi/pytz
- **FreeType 2.3 or later**  Library for reading true type font files. If using pip, easy_install or installing from source, the installer will attempt to locate FreeType in expected locations. If it cannot, try installing pkg-config, a tool used to find required non-python libraries.
- **cycler 0.10.0 or later**  Composable cycle class used for constructing style-cycles
- **six**  Required for compatibility between python 2 and python 3

#### Dependencies for python 2

- **functools32**  Required for compatibility if running on Python 2.7.
- **subprocess32**  Optional, unix only. Backport of the subprocess standard library from 3.2+ for Python 2.7. It provides better error messages and timeout support.

#### Optional GUI framework

These are optional packages which you may want to install to use matplotlib with a user interface toolkit. See What is a backend? for more details on the optional matplotlib backends and the capabilities they provide.

### 2.2. Installing from source
tk 8.3 or later, not 8.6.0 or 8.6.1 The TCL/Tk widgets library used by the TkAgg backend.

Versions 8.6.0 and 8.6.1 are known to have issues that may result in segfaults when closing multiple windows in the wrong order.

pyqt 4.4 or later The Qt4 widgets library python wrappers for the Qt4Agg backend

pygtk 2.4 or later The python wrappers for the GTK widgets library for use with the GTK or GTKAgg backend

wxpython 2.8 or later The python wrappers for the wx widgets library for use with the WX or WXAgg backend

Optional external programs

ffmpeg/avconv or mencoder Required for the animation module to be save output to movie formats.

ImageMagick Required for the animation module to be able to save to animated gif.

Optional dependencies

Pillow If Pillow is installed, matplotlib can read and write a larger selection of image file formats.

pkg-config A tool used to find required non-python libraries. This is not strictly required, but can make installation go more smoothly if the libraries and headers are not in the expected locations.

Required libraries that ship with matplotlib

agg 2.4 The antigrain C++ rendering engine. matplotlib links against the agg template source statically, so it will not affect anything on your system outside of matplotlib.

qhull 2012.1 A library for computing Delaunay triangulations.

ttconv truetype font utility

2.2.2 Building on Linux

It is easiest to use your system package manager to install the dependencies.

If you are on Debian/Ubuntu, you can get all the dependencies required to build matplotlib with:

```
sudo apt-get build-dep python-matplotlib
```

If you are on Fedora/RedHat, you can get all the dependencies required to build matplotlib by first installing yum-builddep and then running:

```
su -c "yum-builddep python-matplotlib"
```

This does not build matplotlib, but it does get the install the build dependencies, which will make building from source easier.
2.2.3 Building on OSX

The build situation on OSX is complicated by the various places one can get the libpng and freetype requirements (darwinports, fink, /usr/X11R6) and the different architectures (e.g., x86, ppc, universal) and the different OSX version (e.g., 10.4 and 10.5). We recommend that you build the way we do for the OSX release: get the source from the tarball or the git repository and follow the instruction in README.osx.

2.2.4 Building on Windows

The Python shipped from https://www.python.org is compiled with Visual Studio 2008 for versions before 3.3, Visual Studio 2010 for 3.3 and 3.4, and Visual Studio 2015 for 3.5 and 3.6. Python extensions are recommended to be compiled with the same compiler.

Since there is no canonical Windows package manager, the methods for building freetype, zlib, and libpng from source code are documented as a build script at matplotlib-winbuild.
3.1 Introductory

3.1.1 Pyplot tutorial

`matplotlib.pyplot` is a collection of command style functions that make matplotlib work like MATLAB. Each `pyplot` function makes some change to a figure: e.g., creates a figure, creates a plotting area in a figure, plots some lines in a plotting area, decorates the plot with labels, etc. In `matplotlib.pyplot` various states are preserved across function calls, so that it keeps track of things like the current figure and plotting area, and the plotting functions are directed to the current axes (please note that “axes” here and in most places in the documentation refers to the *axes part of a figure* and not the strict mathematical term for more than one axis).

```python
import matplotlib.pyplot as plt
plt.plot([1,2,3,4])
plt.ylabel('some numbers')
plt.show()
```
You may be wondering why the x-axis ranges from 0-3 and the y-axis from 1-4. If you provide a single list or array to the `plot()` command, matplotlib assumes it is a sequence of y values, and automatically generates the x values for you. Since python ranges start with 0, the default x vector has the same length as y but starts with 0. Hence the x data are [0, 1, 2, 3].

`plot()` is a versatile command, and will take an arbitrary number of arguments. For example, to plot x versus y, you can issue the command:

```python
plt.plot([1, 2, 3, 4], [1, 4, 9, 16])
```

For every x, y pair of arguments, there is an optional third argument which is the format string that indicates the color and line type of the plot. The letters and symbols of the format string are from MATLAB, and you concatenate a color string with a line style string. The default format string is ‘b-‘, which is a solid blue line. For example, to plot the above with red circles, you would issue

```python
import matplotlib.pyplot as plt
plt.plot([1, 2, 3, 4], [1, 4, 9, 16], 'ro')
plt.axis([0, 6, 0, 20])
plt.show()
```
See the `plot()` documentation for a complete list of line styles and format strings. The `axis()` command in the example above takes a list of `[xmin, xmax, ymin, ymax]` and specifies the viewport of the axes.

If matplotlib were limited to working with lists, it would be fairly useless for numeric processing. Generally, you will use numpy arrays. In fact, all sequences are converted to numpy arrays internally. The example below illustrates a plotting several lines with different format styles in one command using arrays.

```python
import numpy as np
import matplotlib.pyplot as plt

# evenly sampled time at 200ms intervals
t = np.arange(0., 5., 0.2)

# red dashes, blue squares and green triangles
plt.plot(t, t, 'r--', t, t**2, 'bs', t, t**3, 'g^')
plt.show()
```
Controlling line properties

Lines have many attributes that you can set: linewidth, dash style, antialiased, etc; see `matplotlib.lines`. There are several ways to set line properties

- Use keyword args:

  ```python
  plt.plot(x, y, linewidth=2.0)
  ```

- Use the setter methods of a Line2D instance. `plot` returns a list of Line2D objects; e.g., `line1, line2 = plot(x1, y1, x2, y2)`. In the code below we will suppose that we have only one line so that the list returned is of length 1. We use tuple unpacking with `line`, to get the first element of that list:

  ```python
  line, = plt.plot(x, y, '-')
  line.set_antialiased(False)  # turn off antialiasing
  ```

- Use the `setp()` command. The example below uses a MATLAB-style command to set multiple properties on a list of lines. `setp` works transparently with a list of objects or a single object. You can either use python keyword arguments or MATLAB-style string/value pairs:

  ```python
  lines = plt.plot(x1, y1, x2, y2)
  # use keyword args
  plt.setp(lines, color='r', linewidth=2.0)
  ```
Here are the available Line2D properties.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>alpha</td>
<td>float</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transform.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>a Path instance and a Transform instance, a Patch</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>the hit testing function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>data</td>
<td>(np.array xdata, np.array ydata)</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>label</td>
<td>any string</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['-'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>lod</td>
<td>[True</td>
</tr>
<tr>
<td>marker</td>
<td>['+'</td>
</tr>
<tr>
<td>markeredgecolor or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerset or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>[ None</td>
</tr>
<tr>
<td>picker</td>
<td>used in interactive line selection</td>
</tr>
<tr>
<td>pickradius</td>
<td>the line pick selection radius</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>np.array</td>
</tr>
<tr>
<td>ydata</td>
<td>np.array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

To get a list of settable line properties, call the `setp()` function with a line or lines as argument.

```
In [69]: lines = plt.plot([1, 2, 3])

In [70]: plt.setp(lines)
alpha:  float
animated: [True | False]
antialiased or aa: [True | False]
```
Working with multiple figures and axes

MATLAB, and `pyplot`, have the concept of the current figure and the current axes. All plotting commands apply to the current axes. The function `gca()` returns the current axes (a `matplotlib.axes.Axes` instance), and `gcf()` returns the current figure (a `matplotlib.figure.Figure` instance). Normally, you don’t have to worry about this, because it is all taken care of behind the scenes. Below is a script to create two subplots.

```python
import numpy as np
import matplotlib.pyplot as plt

def f(t):
    return np.exp(-t) * np.cos(2*np.pi*t)

t1 = np.arange(0.0, 5.0, 0.1)
t2 = np.arange(0.0, 5.0, 0.02)

plt.figure(1)
plt.subplot(211)
plt.plot(t1, f(t1), 'bo', t2, f(t2), 'k')
plt.subplot(212)
plt.plot(t2, np.cos(2*np.pi*t2), 'r--')
plt.show()
```
The `figure()` command here is optional because `figure(1)` will be created by default, just as a `subplot(111)` will be created by default if you don’t manually specify any axes. The `subplot()` command specifies `numrows, numcols, fignum` where `fignum` ranges from 1 to `numrows*numcols`. The commas in the `subplot` command are optional if `numrows*numcols<10`. So `subplot(211)` is identical to `subplot(2, 1, 1)`. You can create an arbitrary number of subplots and axes. If you want to place an axes manually, i.e., not on a rectangular grid, use the `axes()` command, which allows you to specify the location as `axes([left, bottom, width, height])` where all values are in fractional (0 to 1) coordinates. See `pylab_examples` example code: `axes_demo.py` for an example of placing axes manually and `pylab_examples` example code: `subplots_demo.py` for an example with lots of subplots.

You can create multiple figures by using multiple `figure()` calls with an increasing figure number. Of course, each figure can contain as many axes and subplots as your heart desires:

```python
import matplotlib.pyplot as plt
plt.figure(1) # the first figure
plt.subplot(211) # the first subplot in the first figure
plt.plot([1, 2, 3])
plt.subplot(212) # the second subplot in the first figure
plt.plot([4, 5, 6])
plt.figure(2) # a second figure
plt.plot([4, 5, 6]) # creates a subplot(111) by default
plt.figure(1) # figure 1 current; subplot(212) still current
```
You can clear the current figure with `clf()` and the current axes with `cla()`. If you find it annoying that states (specifically the current image, figure and axes) are being maintained for you behind the scenes, don’t despair: this is just a thin stateful wrapper around an object oriented API, which you can use instead (see Artist tutorial)

If you are making lots of figures, you need to be aware of one more thing: the memory required for a figure is not completely released until the figure is explicitly closed with `close()`. Deleting all references to the figure, and/or using the window manager to kill the window in which the figure appears on the screen, is not enough, because pyplot maintains internal references until `close()` is called.

**Working with text**

The `text()` command can be used to add text in an arbitrary location, and the `xlabel()`, `ylabel()` and `title()` are used to add text in the indicated locations (see Text introduction for a more detailed example)
All of the `text()` commands return an `matplotlib.text.Text` instance. Just as with lines above, you can customize the properties by passing keyword arguments into the text functions or using `setp()`:

```python
import matplotlib.pyplot as plt

# Example

t = plt.xlabel('my data', fontsize=14, color='red')
```

These properties are covered in more detail in *Text properties and layout*.

### Using mathematical expressions in text

`matplotlib` accepts TeX equation expressions in any text expression. For example to write the expression $\sigma_i = 15$ in the title, you can write a TeX expression surrounded by dollar signs:

```python
plt.title(r'\sigma_i=15')
```

The `r` preceding the title string is important – it signifies that the string is a *raw* string and not to treat backslashes as python escapes. `matplotlib` has a built-in TeX expression parser and layout engine, and ships its own math fonts – for details see *Writing mathematical expressions*. Thus you can use mathematical text across platforms without requiring a TeX installation. For those who have LaTeX and dvipng installed, you can also use LaTeX to format your text and incorporate the output directly into your display figures or saved postscript – see *Text rendering With LaTeX*.

---

**3.1. Introductory**
Annotating text

The uses of the basic `text()` command above place text at an arbitrary position on the Axes. A common use for text is to annotate some feature of the plot, and the `annotate()` method provides helper functionality to make annotations easy. In an annotation, there are two points to consider: the location being annotated represented by the argument `xy` and the location of the text `xytext`. Both of these arguments are `(x,y)` tuples.

```python
import numpy as np
import matplotlib.pyplot as plt

ax = plt.subplot(111)

t = np.arange(0.0, 5.0, 0.01)
s = np.cos(2*np.pi*t)
line, = plt.plot(t, s, lw=2)

plt.annotate('local max', xy=(2, 1), xytext=(3, 1.5),
             arrowprops=dict(facecolor='black', shrink=0.05),
            )

plt.ylim(-2,2)
plt.show()
```

In this basic example, both the `xy` (arrow tip) and `xytext` locations (text location) are in data coordinates.
There are a variety of other coordinate systems one can choose – see *Basic annotation* and *Advanced Annotation* for details. More examples can be found in *pylab_examples example code: annotation_demo.py*.

Logarithmic and other nonlinear axes

*matplotlib.pyplot* supports not only linear axis scales, but also logarithmic and logit scales. This is commonly used if data spans many orders of magnitude. Changing the scale of an axis is easy:

```
plt.xscale('log')
```

An example of four plots with the same data and different scales for the y axis is shown below.

```python
import numpy as np
import matplotlib.pyplot as plt

from matplotlib.ticker import NullFormatter  # useful for 'logit' scale

# Fixing random state for reproducibility
np.random.seed(19680801)

# make up some data in the interval ]0, 1[
y = np.random.normal(loc=0.5, scale=0.4, size=1000)
y = y[(y > 0) & (y < 1)]
y.sort()
x = np.arange(len(y))

# plot with various axes scales
plt.figure(1)

# linear
plt.subplot(221)
plt.plot(x, y)
plt.yscale('linear')
plt.title('linear')
plt.grid(True)

# log
plt.subplot(222)
plt.plot(x, y)
plt.yscale('log')
plt.title('log')
plt.grid(True)

# symmetric log
plt.subplot(223)
plt.plot(x, y - y.mean())
plt.yscale('symlog', linthreshy=0.01)
plt.title('symlog')
plt.grid(True)

# logit
```
It is also possible to add your own scale, see Developer’s guide for creating scales and transformations for details.

3.1.2 Image tutorial
**Startup commands**

First, let’s start IPython. It is a most excellent enhancement to the standard Python prompt, and it ties in especially well with Matplotlib. Start IPython either at a shell, or the IPython Notebook now.

With IPython started, we now need to connect to a GUI event loop. This tells IPython where (and how) to display plots. To connect to a GUI loop, execute the `%matplotlib` magic at your IPython prompt. There’s more detail on exactly what this does at IPython’s documentation on GUI event loops.

If you’re using IPython Notebook, the same commands are available, but people commonly use a specific argument to the `%matplotlib` magic:

```python
In [1]: %matplotlib inline
```

This turns on inline plotting, where plot graphics will appear in your notebook. This has important implications for interactivity. For inline plotting, commands in cells below the cell that outputs a plot will not affect the plot. For example, changing the color map is not possible from cells below the cell that creates a plot. However, for other backends, such as qt4, that open a separate window, cells below those that create the plot will change the plot - it is a live object in memory.

This tutorial will use matplotlib’s imperative-style plotting interface, pyplot. This interface maintains global state, and is very useful for quickly and easily experimenting with various plot settings. The alternative is the object-oriented interface, which is also very powerful, and generally more suitable for large application development. If you’d like to learn about the object-oriented interface, a great place to start is our FAQ on usage. For now, let’s get on with the imperative-style approach:

```python
In [2]: import matplotlib.pyplot as plt
In [3]: import matplotlib.image as mpimg
In [4]: import numpy as np
```

**Importing image data into Numpy arrays**

Loading image data is supported by the Pillow library. Natively, matplotlib only supports PNG images. The commands shown below fall back on Pillow if the native read fails.

The image used in this example is a PNG file, but keep that Pillow requirement in mind for your own data.

Here’s the image we’re going to play with:
It's a 24-bit RGB PNG image (8 bits for each of R, G, B). Depending on where you get your data, the other kinds of image that you'll most likely encounter are RGBA images, which allow for transparency, or single-channel grayscale (luminosity) images. You can right click on it and choose “Save image as” to download it to your computer for the rest of this tutorial.

And here we go...

```python
In [5]: img = mpimg.imread('stinkbug.png')
Out[5]:
array([[ 0.40784314,  0.40784314,  0.40784314],
       [ 0.40784314,  0.40784314,  0.40784314],
       [ 0.40784314,  0.40784314,  0.40784314],
       ..., 0.42745098, 0.42745098, 0.42745098],
       [ 0.42745098,  0.42745098,  0.42745098],
       [ 0.42745098,  0.42745098,  0.42745098],
       [ 0.42745098,  0.42745098,  0.42745098],
       ...
       [[ 0.44313726,  0.44313726,  0.44313726],
       [ 0.4509804 ,  0.4509804 ,  0.4509804 ],
       [ 0.4509804 ,  0.4509804 ,  0.4509804 ],
       ..., 0.44705883, 0.44705883, 0.44705883]),
```

Chapter 3. Tutorials
Note the dtype there - float32. Matplotlib has rescaled the 8 bit data from each channel to floating point data between 0.0 and 1.0. As a side note, the only datatype that Pillow can work with is uint8. Matplotlib plotting can handle float32 and uint8, but image reading/writing for any format other than PNG is limited to uint8 data. Why 8 bits? Most displays can only render 8 bits per channel worth of color gradation. Why can they only render 8 bits/channel? Because that’s about all the human eye can see. More here (from a photography standpoint): Luminous Landscape bit depth tutorial.

Each inner list represents a pixel. Here, with an RGB image, there are 3 values. Since it’s a black and white image, R, G, and B are all similar. An RGBA (where A is alpha, or transparency), has 4 values per inner list, and a simple luminance image just has one value (and is thus only a 2-D array, not a 3-D array). For RGB and RGBA images, matplotlib supports float32 and uint8 data types. For grayscale, matplotlib supports only float32. If your array data does not meet one of these descriptions, you need to rescale it.

### Plotting numpy arrays as images

So, you have your data in a numpy array (either by importing it, or by generating it). Let’s render it. In Matplotlib, this is performed using the `imshow()` function. Here we’ll grab the plot object. This object gives you an easy way to manipulate the plot from the prompt.

```python
In [6]: imgplot = plt.imshow(img)
```
You can also plot any numpy array.

**Applying pseudocolor schemes to image plots**

Pseudocolor can be a useful tool for enhancing contrast and visualizing your data more easily. This is especially useful when making presentations of your data using projectors - their contrast is typically quite poor.

Pseudocolor is only relevant to single-channel, grayscale, luminosity images. We currently have an RGB image. Since R, G, and B are all similar (see for yourself above or in your data), we can just pick one channel of our data:

```python
In [7]: lum_img = img[:,:,0]
```

This is array slicing. You can read more in the [Numpy tutorial](https://numpy.org/doc/).

```python
In [8]: plt.imshow(lum_img)
```

![Image plot](image.png)

Now, with a luminosity (2D, no color) image, the default colormap (aka lookup table, LUT), is applied. The default is called viridis. There are plenty of others to choose from.

```python
In [9]: plt.imshow(lum_img, cmap="hot")
```
Note that you can also change colormaps on existing plot objects using the `set_cmap()` method:

```python
In [10]: imgplot = plt.imshow(lum_img)
In [11]: imgplot.set_cmap('nipy_spectral')
```
Note: However, remember that in the IPython notebook with the inline backend, you can’t make changes to plots that have already been rendered. If you create imgplot here in one cell, you cannot call set_cmap() on it in a later cell and expect the earlier plot to change. Make sure that you enter these commands together in one cell. plt commands will not change plots from earlier cells.

There are many other colormap schemes available. See the list and images of the colormaps.

Color scale reference

It’s helpful to have an idea of what value a color represents. We can do that by adding color bars.

```python
In [12]: imgplot = plt.imshow(lum_img)
In [13]: plt.colorbar()
```
This adds a colorbar to your existing figure. This won’t automatically change if you change you switch to a different colormap - you have to re-create your plot, and add in the colorbar again.

**Examining a specific data range**

Sometimes you want to enhance the contrast in your image, or expand the contrast in a particular region while sacrificing the detail in colors that don’t vary much, or don’t matter. A good tool to find interesting regions is the histogram. To create a histogram of our image data, we use the `hist()` function.

```
In [14]: plt.hist(lum_img.ravel(), bins=256, range=(0.0, 1.0), fc='k', ec='k')
```
Most often, the “interesting” part of the image is around the peak, and you can get extra contrast by clipping the regions above and/or below the peak. In our histogram, it looks like there’s not much useful information in the high end (not many white things in the image). Let’s adjust the upper limit, so that we effectively “zoom in on” part of the histogram. We do this by passing the clim argument to imshow. You could also do this by calling the set_clim() method of the image plot object, but make sure that you do so in the same cell as your plot command when working with the IPython Notebook - it will not change plots from earlier cells.

```
In [15]: imgplot = plt.imshow(lum_img, clim=(0.0, 0.7))
```
Array Interpolation schemes

Interpolation calculates what the color or value of a pixel “should” be, according to different mathematical schemes. One common place that this happens is when you resize an image. The number of pixels change, but you want the same information. Since pixels are discrete, there’s missing space. Interpolation is how you fill that space. This is why your images sometimes come out looking pixelated when you blow them up. The effect is more pronounced when the difference between the original image and the expanded image is greater. Let’s take our image and shrink it. We’re effectively discarding pixels, only keeping a select few. Now when we plot it, that data gets blown up to the size on your screen. The old pixels aren’t there anymore, and the computer has to draw in pixels to fill that space.

We’ll use the Pillow library that we used to load the image also to resize the image.

```python
In [16]: from PIL import Image
In [17]: img = Image.open('../_static/stinkbug.png')
In [18]: img.thumbnail((64, 64), Image.ANTIALIAS) # resizes image in-place
In [19]: imgplot = plt.imshow(img)
```
Here we have the default interpolation, bilinear, since we did not give `imshow()` any interpolation argument. Let’s try some others:

```
In [20]: imgplot = plt.imshow(img, interpolation="nearest")
```
In [21]: imgplot = plt.imshow(img, interpolation="bicubic")
Bicubic interpolation is often used when blowing up photos - people tend to prefer blurry over pixelated.

### 3.1.3 Customizing Location of Subplot Using GridSpec

**GridSpec** specifies the geometry of the grid that a subplot will be placed. The number of rows and number of columns of the grid need to be set. Optionally, the subplot layout parameters (e.g., left, right, etc.) can be tuned.

**SubplotSpec** specifies the location of the subplot in the given **GridSpec**.

**subplot2grid()** a helper function that is similar to **subplot()** but uses 0-based indexing and let subplot to occupy multiple cells.

#### Basic Example of using subplot2grid

To use **subplot2grid()**, you provide geometry of the grid and the location of the subplot in the grid. For a simple single-cell subplot:

```python
ax = plt.subplot2grid((2, 2), (0, 0))
```

is identical to
ax = plt.subplot(2, 2, 1)

Note that, unlike Matplotlib's subplot, the index starts from 0 in GridSpec.

To create a subplot that spans multiple cells,

```python
ax2 = plt.subplot2grid((3, 3), (1, 0), colspan=2)
ax3 = plt.subplot2grid((3, 3), (1, 2), rowspan=2)
```

For example, the following commands

```python
ax1 = plt.subplot2grid((3, 3), (0, 0), colspan=3)
ax2 = plt.subplot2grid((3, 3), (1, 0), colspan=2)
ax3 = plt.subplot2grid((3, 3), (1, 2), rowspan=2)
ax4 = plt.subplot2grid((3, 3), (2, 0))
ax5 = plt.subplot2grid((3, 3), (2, 1))
```

creates

```
ax1
ax2
ax3
ax4
```

**GridSpec and SubplotSpec**

You can create `GridSpec` explicitly and use them to create a subplot.

For example,
ax = plt.subplot2grid((2, 2), (0, 0))

is equal to

```python
import matplotlib.gridspec as gridspec
gs = gridspec.GridSpec(2, 2)
ax = plt.subplot(gs[0, 0])
```

A GridSpec instance provides array-like (2d or 1d) indexing that returns the SubplotSpec instance. For a SubplotSpec that spans multiple cells, use slice.

```python
ax2 = plt.subplot(gs[1, :-1])
ax3 = plt.subplot(gs[1:, -1])
```

The above example becomes

```python
gs = gridspec.GridSpec(3, 3)
ax1 = plt.subplot(gs[0, :])
ax2 = plt.subplot(gs[1, :-1])
ax3 = plt.subplot(gs[1:, -1])
ax4 = plt.subplot(gs[-1, 0])
ax5 = plt.subplot(gs[-1, -2])
```

![GridSpec Diagram](attachment:image.png)
**Adjust GridSpec layout**

When a GridSpec is explicitly used, you can adjust the layout parameters of subplots that are created from the GridSpec.

```python
gs1 = gridspec.GridSpec(3, 3)
gs1.update(left=0.05, right=0.48, wspace=0.05)
```

This is similar to `subplots_adjust()`, but it only affects the subplots that are created from the given GridSpec.

The code below

```python
gs1 = gridspec.GridSpec(3, 3)
gs1.update(left=0.05, right=0.48, wspace=0.05)
ax1 = plt.subplot(gs1[:-1, :])
ax2 = plt.subplot(gs1[-1, :-1])
ax3 = plt.subplot(gs1[-1, -1])

gs2 = gridspec.GridSpec(3, 3)
gs2.update(left=0.55, right=0.98, hspace=0.05)
ax4 = plt.subplot(gs2[:, :-1])
ax5 = plt.subplot(gs2[:-1, -1])
ax6 = plt.subplot(gs2[-1, -1])
```

creates

**GridSpec w/ different subplotpars**

![Diagram of GridSpec with different subplotpars]
**GridSpec using SubplotSpec**

You can create GridSpec from the `SubplotSpec`, in which case its layout parameters are set to that of the location of the given SubplotSpec.

```python
gs0 = gridspec.GridSpec(1, 2)
gs00 = gridspec.GridSpecFromSubplotSpec(3, 3, subplot_spec=gs0[0])
gs01 = gridspec.GridSpecFromSubplotSpec(3, 3, subplot_spec=gs0[1])
```

**GridSpec Inside GridSpec**

![Diagram of GridSpec layout]

**A Complex Nested GridSpec using SubplotSpec**

Here’s a more sophisticated example of nested GridSpec where we put a box around each cell of the outer 4x4 grid, by hiding appropriate spines in each of the inner 3x3 grids.
GridSpec with Varying Cell Sizes

By default, GridSpec creates cells of equal sizes. You can adjust relative heights and widths of rows and columns. Note that absolute values are meaningless, only their relative ratios matter.

```python
gs = gridspec.GridSpec(2, 2,
                        width_ratios=[1, 2],
                        height_ratios=[4, 1])

ax1 = plt.subplot(gs[0])
ax2 = plt.subplot(gs[1])
```
3.1.4 Tight Layout guide

tight_layout automatically adjusts subplot params so that the subplot(s) fits in to the figure area. This is an experimental feature and may not work for some cases. It only checks the extents of ticklabels, axis labels, and titles.

Simple Example

In matplotlib, the location of axes (including subplots) are specified in normalized figure coordinates. It can happen that your axis labels or titles (or sometimes even ticklabels) go outside the figure area, and are thus clipped.

```python
plt.rcParams['savefig.facecolor'] = "0.8"

def example_plot(ax, fontsize=12):
    ax.plot([1, 2])
    ax.locator_params(nbins=3)
    ax.set_xlabel('x-label', fontsize=fontsize)
    ax.set_ylabel('y-label', fontsize=fontsize)
```
To prevent this, the location of axes needs to be adjusted. For subplots, this can be done by adjusting the subplot params (Move the edge of an axes to make room for tick labels). Matplotlib v1.1 introduces a new command tight_layout() that does this automatically for you.

```python
plt.tight_layout()
```
When you have multiple subplots, often you see labels of different axes overlapping each other.

```python
plt.close('all')
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(nrows=2, ncols=2)
exmaple_plot(ax1)
exmaple_plot(ax2)
exmaple_plot(ax3)
exmaple_plot(ax4)
```
`tight_layout()` will also adjust spacing between subplots to minimize the overlaps.

```python
plt.tight_layout()
```
tight_layout() can take keyword arguments of pad, w_pad and h_pad. These control the extra padding around the figure border and between subplots. The pads are specified in fraction of fontsize.

```python
plt.tight_layout(pad=0.4, w_pad=0.5, h_pad=1.0)
```
tight_layout() will work even if the sizes of subplots are different as far as their grid specification is compatible. In the example below, ax1 and ax2 are subplots of a 2x2 grid, while ax3 is of a 1x2 grid.

```python
plt.close('all')
fig = plt.figure()
ax1 = plt.subplot(221)
ax2 = plt.subplot(223)
ax3 = plt.subplot(122)
example_plot(ax1)
example_plot(ax2)
example_plot(ax3)
plt.tight_layout()
```
It works with subplots created with `subplot2grid()`. In general, subplots created from the gridspec (Customizing Location of Subplot Using GridSpec) will work.

```python
plt.close('all')
fig = plt.figure()

ax1 = plt.subplot2grid((3, 3), (0, 0))
ax2 = plt.subplot2grid((3, 3), (0, 1), colspan=2)
ax3 = plt.subplot2grid((3, 3), (1, 0), colspan=2, rowspan=2)
ax4 = plt.subplot2grid((3, 3), (1, 2), rowspan=2)

example_plot(ax1)
example_plot(ax2)
example_plot(ax3)
example_plot(ax4)

plt.tight_layout()
```
Although not thoroughly tested, it seems to work for subplots with aspect != “auto” (e.g., axes with images).

```python
arr = np.arange(100).reshape((10,10))
fig = plt.figure(figsize=(5,4))
ax = plt.subplot(111)
im = ax.imshow(arr, interpolation="none")
plt.tight_layout()
```
Caveats

- `tight_layout()` only considers ticklabels, axis labels, and titles. Thus, other artists may be clipped and also may overlap.

- It assumes that the extra space needed for ticklabels, axis labels, and titles is independent of original location of axes. This is often true, but there are rare cases where it is not.

- `pad=0` clips some of the texts by a few pixels. This may be a bug or a limitation of the current algorithm and it is not clear why it happens. Meanwhile, use of pad at least larger than 0.3 is recommended.

Use with GridSpec

GridSpec has its own `tight_layout()` method (the pyplot api `tight_layout()` also works).

```python
plt.close('all')
fig = plt.figure()

import matplotlib.gridspec as gridspec
gs1 = gridspec.GridSpec(2, 1)
ax1 = fig.add_subplot(gs1[0])
ax2 = fig.add_subplot(gs1[1])
```
You may provide an optional `rect` parameter, which specifies the bounding box that the subplots will be fit inside. The coordinates must be in normalized figure coordinates and the default is (0, 0, 1, 1).

gs1.tight_layout(fig, rect=[0, 0, 0.5, 1])
For example, this can be used for a figure with multiple gridspecs.

```python
gs2 = gridspec.GridSpec(3, 1)

for ss in gs2:
    ax = fig.add_subplot(ss)
    example_plot(ax)
    ax.set_title('')
    ax.set_xlabel('')

ax.set_xlabel('x-label', fontsize=12)

gs2.tight_layout(fig, rect=[0.5, 0, 1, 1], h_pad=0.5)
```
We may try to match the top and bottom of two grids:

```python
import matplotlib.transforms as mtransforms

top = min(gs1.top, gs2.top)
bottom = max(gs1.bottom, gs2.bottom)

gs1.update(top=top, bottom=bottom)
gs2.update(top=top, bottom=bottom)
```

While this should be mostly good enough, adjusting top and bottom may require adjustment of hspace also. To update hspace & vspace, we call `tight_layout()` again with updated rect argument. Note that the rect argument specifies the area including the ticklabels, etc. Thus, we will increase the bottom (which is 0 for the normal case) by the difference between the `bottom` from above and the bottom of each gridspec. Same thing for the top:

```python

top = min(gs1.top, gs2.top)
bottom = max(gs1.bottom, gs2.bottom)

    gs1.tight_layout(fig, rect=[None, 0 + (bottom-gs1.bottom), 0.5, 1 - (gs1.top-top)])
    gs2.tight_layout(fig, rect=[0.5, 0 + (bottom-gs2.bottom), None, 1 - (gs2.top-top)], h_pad=0.5)
```
Use with AxesGrid1

While limited, the axes_grid1 toolkit is also supported.

```python
plt.close('all')
fig = plt.figure()

from mpl_toolkits.axes_grid1 import Grid
grid = Grid(fig, rect=111, nrows_ncols=(2,2),
            axes_pad=0.25, label_mode='L',
            )

for ax in grid:
    example_plot(ax)
    ax.title.set_visible(False)

plt.tight_layout()
```
Colorbar

If you create a colorbar with the `colorbar()` command, the created colorbar is an instance of Axes, not Subplot, so tight_layout does not work. With Matplotlib v1.1, you may create a colorbar as a subplot using the gridspec.

```python
plt.close('all')
arr = np.arange(100).reshape((10, 10))
fig = plt.figure(figsize=(4, 4))
im = plt.imshow(arr, interpolation="none")
plt.colorbar(im, use_gridspec=True)
plt.tight_layout()
```
Another option is to use AxesGrid1 toolkit to explicitly create an axes for colorbar.

```python
plt.close('all')
arr = np.arange(100).reshape((10, 10))
fig = plt.figure(figsize=(4, 4))
im = plt.imshow(arr, interpolation="none")

divider = make_axes_locatable(plt.gca())
cax = divider.append_axes("right", "5%", pad="3%")
plt.colorbar(im, cax=cax)
plt.tight_layout()
```
3.2 Intermediate

3.2.1 Artist tutorial

There are three layers to the matplotlib API. The *matplotlib.backend_bases.FigureCanvas* is the area onto which the figure is drawn, the *matplotlib.backend_bases.Renderer* is the object which knows how to draw on the *FigureCanvas*, and the *matplotlib.artist.Artist* is the object that knows how to use a renderer to paint onto the canvas. The *FigureCanvas* and *Renderer* handle all the details of talking to user interface toolkits like *wxPython* or drawing languages like *PostScript®*, and the *Artist* handles all the high level constructs like representing and laying out the figure, text, and lines. The typical user will spend 95% of their time working with the *Artists*.

There are two types of *Artists*: primitives and containers. The primitives represent the standard graphical objects we want to paint onto our canvas: *Line2D, Rectangle, Text, AxesImage*, etc., and the containers are places to put them (*Axis, Axes* and *Figure*). The standard use is to create a *Figure* instance, use the *Figure* to create one or more *Axes* or *Subplot* instances, and use the *Axes* instance helper methods to create the primitives. In the example below, we create a *Figure* instance using *matplotlib.pyplot.figure()*, which is a convenience method for instantiating *Figure* instances and connecting them with your user interface or drawing toolkit *FigureCanvas*. As we will discuss below, this is not necessary – you can work directly with *PostScript*, *PDF* *Gtk+*, or *wxPython* *FigureCanvas* instances, instantiate your *Figures* directly and connect them yourselves – but since we are focusing here on the *Artist* API we’ll let *pyplot* handle some of those details for us:
import matplotlib.pyplot as plt
fig = plt.figure()
ax = fig.add_subplot(2,1,1)  # two rows, one column, first plot

The Axes is probably the most important class in the matplotlib API, and the one you will be working with most of the time. This is because the Axes is the plotting area into which most of the objects go, and the Axes has many special helper methods (plot(), text(), hist(), imshow()) to create the most common graphics primitives (Line2D, Text, Rectangle, Image, respectively). These helper methods will take your data (e.g., numpy arrays and strings) and create primitive Artist instances as needed (e.g., Line2D), add them to the relevant containers, and draw them when requested. Most of you are probably familiar with the Subplot, which is just a special case of an Axes that lives on a regular rows by columns grid of Subplot instances. If you want to create an Axes at an arbitrary location, simply use the add_axes() method which takes a list of [left, bottom, width, height] values in 0-1 relative figure coordinates:

fig2 = plt.figure()
ax2 = fig2.add_axes([0.15, 0.1, 0.7, 0.3])

Continuing with our example:

import numpy as np
t = np.arange(0.0, 1.0, 0.01)
s = np.sin(2*np.pi*t)
line, = ax.plot(t, s, color='blue', lw=2)

In this example, ax is the Axes instance created by the fig.add_subplot call above (remember Subplot is just a subclass of Axes) and when you call ax.plot, it creates a Line2D instance and adds it to the Axes.lines list. In the interactive ipython session below, you can see that the Axes.lines list is length one and contains the same line that was returned by the line, = ax.plot... call:

In [101]: ax.lines[0]
Out[101]: <matplotlib.lines.Line2D instance at 0x19a95710>

In [102]: line
Out[102]: <matplotlib.lines.Line2D instance at 0x19a95710>

If you make subsequent calls to ax.plot (and the hold state is “on” which is the default) then additional lines will be added to the list. You can remove lines later simply by calling the list methods; either of these will work:

del ax.lines[0]
ax.lines.remove(line)  # one or the other, not both!

The Axes also has helper methods to configure and decorate the x-axis and y-axis tick, tick labels and axis labels:

xtext = ax.set_xlabel('my xdata')  # returns a Text instance
ytext = ax.set_ylabel('my ydata')

When you call ax.set_xlabel, it passes the information on the Text instance of the XAxis. Each Axes instance contains an XAxis and a YAxis instance, which handle the layout and drawing of the ticks, tick labels and axis labels.
Try creating the figure below.

Customizing your objects

Every element in the figure is represented by a matplotlib Artist, and each has an extensive list of properties to configure its appearance. The figure itself contains a Rectangle exactly the size of the figure, which you can use to set the background color and transparency of the figures. Likewise, each Axes bounding box (the standard white box with black edges in the typical matplotlib plot, has a Rectangle instance that determines the color, transparency, and other properties of the Axes. These instances are stored as member variables Figure.patch and Axes.patch (“Patch” is a name inherited from MATLAB, and is a 2D “patch” of color on the figure, e.g., rectangles, circles and polygons). Every matplotlib Artist has the following properties
Matplotlib, Release 2.0.2

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>alpha</td>
<td>The transparency - a scalar from 0-1</td>
</tr>
<tr>
<td>animated</td>
<td>A boolean that is used to facilitate animated drawing</td>
</tr>
<tr>
<td>axes</td>
<td>The axes that the Artist lives in, possibly None</td>
</tr>
<tr>
<td>clip_box</td>
<td>The bounding box that clips the Artist</td>
</tr>
<tr>
<td>clip_on</td>
<td>Whether clipping is enabled</td>
</tr>
<tr>
<td>clip_path</td>
<td>The path the artist is clipped to</td>
</tr>
<tr>
<td>contains</td>
<td>A picking function to test whether the artist contains the pick point</td>
</tr>
<tr>
<td>figure</td>
<td>The figure instance the artist lives in, possibly None</td>
</tr>
<tr>
<td>label</td>
<td>A text label (e.g., for auto-labeling)</td>
</tr>
<tr>
<td>picker</td>
<td>A python object that controls object picking</td>
</tr>
<tr>
<td>transform</td>
<td>The transformation</td>
</tr>
<tr>
<td>visible</td>
<td>A boolean whether the artist should be drawn</td>
</tr>
<tr>
<td>zorder</td>
<td>A number which determines the drawing order</td>
</tr>
<tr>
<td>rasterized</td>
<td>Boolean; Turns vectors into rastergraphics: (for compression &amp; eps transparency)</td>
</tr>
</tbody>
</table>

Each of the properties is accessed with an old-fashioned setter or getter (yes we know this irritates Pythonistas and we plan to support direct access via properties or traits but it hasn’t been done yet). For example, to multiply the current alpha by a half:

```python
a = o.get_alpha()
o.set_alpha(0.5*a)
```

If you want to set a number of properties at once, you can also use the `set` method with keyword arguments. For example:

```python
o.set(alpha=0.5, zorder=2)
```

If you are working interactively at the python shell, a handy way to inspect the Artist properties is to use the `matplotlib.artist.getp()` function (simply `getp()` in pylab), which lists the properties and their values. This works for classes derived from Artist as well, e.g., Figure and Rectangle. Here are the Figure rectangle properties mentioned above:

```python
In [149]: matplotlib.artist.getp(fig.patch)
    alpha = 1.0
    animated = False
    antialiased or aa = True
    axes = None
    clip_box = None
    clip_on = False
    clip_path = None
    contains = None
    edgcolor or ec = w
    facecolor or fc = 0.75
    figure = Figure(8.125x6.125)
    fill = 1
    hatch = None
    height = 1
    label =
    linewidth or lw = 1.0
    picker = None
```
The docstrings for all of the classes also contain the Artist properties, so you can consult the interactive “help” or the artist Module for a listing of properties for a given object.

Object containers

Now that we know how to inspect and set the properties of a given object we want to configure, we need to know how to get at that object. As mentioned in the introduction, there are two kinds of objects: primitives and containers. The primitives are usually the things you want to configure (the font of a Text instance, the width of a Line2D) although the containers also have some properties as well – for example the Axes Artist is a container that contains many of the primitives in your plot, but it also has properties like the xscale to control whether the xaxis is ‘linear’ or ‘log’. In this section we’ll review where the various container objects store the Artists that you want to get at.

Figure container

The top level container Artist is the matplotlib.figure.Figure, and it contains everything in the figure. The background of the figure is a Rectangle which is stored in Figure.patch. As you add subplots (add_subplot()) and axes (add_axes()) to the figure these will be appended to the Figure.axes. These are also returned by the methods that create them:

```python
In [156]: fig = plt.figure()
In [157]: ax1 = fig.add_subplot(211)
In [158]: ax2 = fig.add_axes([0.1, 0.1, 0.7, 0.3])
In [159]: ax1
Out[159]: <matplotlib.axes.Subplot instance at 0xd54b26c>
In [160]: print fig.axes
   [<matplotlib.axes.Subplot instance at 0xd54b26c>, <matplotlib.axes.Axes instance at 0xd3f0b2c>]
```

Because the figure maintains the concept of the “current axes” (see Figure.gca and Figure.sca) to support the pylab/pyplot state machine, you should not insert or remove axes directly from the axes list, but rather use the add_subplot() and add_axes() methods to insert, and the delaxes() method to delete. You are free however, to iterate over the list of axes or index into it to get access to Axes instances you want to customize. Here is an example which turns all the axes grids on:
for ax in fig.axes:
    ax.grid(True)

The figure also has its own text, lines, patches and images, which you can use to add primitives directly. The
default coordinate system for the Figure will simply be in pixels (which is not usually what you want) but
you can control this by setting the transform property of the Artist you are adding to the figure.

More useful is “figure coordinates” where (0, 0) is the bottom-left of the figure and (1, 1) is the top-right of
the figure which you can obtain by setting the Artist transform to fig.transFigure:

In [191]: fig = plt.figure()

In [192]: l1 = matplotlib.lines.Line2D([0, 1], [0, 1],
                                 transform=fig.transFigure, figure=fig)

In [193]: l2 = matplotlib.lines.Line2D([0, 1], [1, 0],
                                 transform=fig.transFigure, figure=fig)

In [194]: fig.lines.extend([l1, l2])

In [195]: fig.canvas.draw()

Here is a summary of the Artists the figure contains
<table>
<thead>
<tr>
<th>Figure attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>axes</td>
<td>A list of Axes instances (includes Subplot)</td>
</tr>
<tr>
<td>patch</td>
<td>The Rectangle background</td>
</tr>
<tr>
<td>images</td>
<td>A list of FigureImages patches - useful for raw pixel display</td>
</tr>
<tr>
<td>legends</td>
<td>A list of Figure Legend instances (different from Axes.legends)</td>
</tr>
<tr>
<td>lines</td>
<td>A list of Figure Line2D instances (rarely used, see Axes.lines)</td>
</tr>
<tr>
<td>patches</td>
<td>A list of Figure patches (rarely used, see Axes.patches)</td>
</tr>
<tr>
<td>texts</td>
<td>A list Figure Text instances</td>
</tr>
</tbody>
</table>

**Axes container**

The `matplotlib.axes.Axes` is the center of the matplotlib universe – it contains the vast majority of all the Artists used in a figure with many helper methods to create and add these Artists to itself, as well as helper methods to access and customize the Artists it contains. Like the Figure, it contains a Patch patch which is a Rectangle for Cartesian coordinates and a Circle for polar coordinates; this patch determines the shape, background and border of the plotting region:

```python
ax = fig.add_subplot(111)
rect = ax.patch  # a Rectangle instance
rect.set_facecolor('green')
```

When you call a plotting method, e.g., the canonical `plot()` and pass in arrays or lists of values, the method will create a `matplotlib.lines.Line2D()` instance, update the line with all the Line2D properties passed as keyword arguments, add the line to the Axes.lines container, and returns it to you:

```python
In [213]: x, y = np.random.rand(2, 100)
In [214]: line, = ax.plot(x, y, '-', color='blue', linewidth=2)
```

`plot` returns a list of lines because you can pass in multiple x, y pairs to plot, and we are unpacking the first element of the length one list into the line variable. The line has been added to the Axes.lines list:

```python
In [229]: print ax.lines
[[<matplotlib.lines.Line2D instance at 0xd378b0c>]]
```

Similarly, methods that create patches, like `bar()` creates a list of rectangles, will add the patches to the Axes.patches list:

```python
In [233]: n, bins, rectangles = ax.hist(np.random.randn(1000), 50, facecolor='yellow')
In [234]: rectangles
Out[234]: <a list of 50 Patch objects>
In [235]: print len(ax.patches)
```

You should not add objects directly to the Axes.lines or Axes.patches lists unless you know exactly what you are doing, because the Axes needs to do a few things when it creates and adds an object. It sets the figure and axes property of the Artist, as well as the default Axes transformation (unless a transformation is set). It also inspects the data contained in the Artist to update the data structures controlling auto-scaling,
so that the view limits can be adjusted to contain the plotted data. You can, nonetheless, create objects
yourself and add them directly to the Axes using helper methods like add_line() and add_patch().
Here is an annotated interactive session illustrating what is going on:

```
In [261]: fig = plt.figure()
In [262]: ax = fig.add_subplot(111)

# create a rectangle instance
In [263]: rect = matplotlib.patches.Rectangle( (1,1), width=5, height=12)

# by default the axes instance is None
In [264]: print rect.get_axes()
None

# and the transformation instance is set to the "identity transform"
In [265]: print rect.get_transform()
<Affine object at 0x13695544>

# now we add the Rectangle to the Axes
In [266]: ax.add_patch(rect)

# and notice that the ax.add_patch method has set the axes
# instance
In [267]: print rect.get_axes()
Axes(0.125,0.1;0.775x0.8)

# and the transformation has been set too
In [268]: print rect.get_transform()
<Affine object at 0x15009ca4>

# the default axes transformation is ax.transData
In [269]: print ax.transData
<Affine object at 0x15009ca4>

# notice that the xlims of the Axes have not been changed
In [270]: print ax.get_xlim()
(0.0, 1.0)

# but the data limits have been updated to encompass the rectangle
In [271]: print ax.dataLim.bounds
(1.0, 1.0, 5.0, 12.0)

# we can manually invoke the auto-scaling machinery
In [272]: ax.autoscale_view()

# and now the xlim are updated to encompass the rectangle
In [273]: print ax.get_xlim()
(1.0, 6.0)

# we have to manually force a figure draw
In [274]: ax.figure.canvas.draw()
```
There are many, many \texttt{Axes} helper methods for creating primitive \texttt{Artist}s and adding them to their respective containers. The table below summarizes a small sampling of them, the kinds of \texttt{Artist} they create, and where they store them.

<table>
<thead>
<tr>
<th>Helper method</th>
<th>Artist</th>
<th>Container</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{ax.annotate}</td>
<td>\texttt{Annotate}</td>
<td>\texttt{ax.texts}</td>
</tr>
<tr>
<td>\texttt{ax.bar}</td>
<td>\texttt{Rectangle}</td>
<td>\texttt{ax.patches}</td>
</tr>
<tr>
<td>\texttt{ax.errorbar}</td>
<td>\texttt{Line2D} and \texttt{Rectangle}</td>
<td>\texttt{ax.lines} and \texttt{ax.patches}</td>
</tr>
<tr>
<td>\texttt{ax.fill}</td>
<td>\texttt{Polygon}</td>
<td>\texttt{ax.patches}</td>
</tr>
<tr>
<td>\texttt{ax.hist}</td>
<td>\texttt{Rectangle}</td>
<td>\texttt{ax.patches}</td>
</tr>
<tr>
<td>\texttt{ax.imshow}</td>
<td>\texttt{AxesImage}</td>
<td>\texttt{ax.images}</td>
</tr>
<tr>
<td>\texttt{ax.legend}</td>
<td>\texttt{Legend}</td>
<td>\texttt{ax.legend}</td>
</tr>
<tr>
<td>\texttt{ax.plot}</td>
<td>\texttt{Line2D}</td>
<td>\texttt{ax.lines}</td>
</tr>
<tr>
<td>\texttt{ax.scatter}</td>
<td>\texttt{PolygonCollection}</td>
<td>\texttt{ax.collections}</td>
</tr>
<tr>
<td>\texttt{ax.text}</td>
<td>\texttt{Text}</td>
<td>\texttt{ax.texts}</td>
</tr>
</tbody>
</table>

In addition to all of these \texttt{Artist}s, the \texttt{Axes} contains two important \texttt{Artist} containers: the \texttt{XAxis} and \texttt{YAxis}, which handle the drawing of the ticks and labels. These are stored as instance variables \texttt{xaxis} and \texttt{yaxis}. The \texttt{XAxis} and \texttt{YAxis} containers will be detailed below, but note that the \texttt{Axes} contains many helper methods which forward calls on to the \texttt{Axis} instances so you often do not need to work with them directly unless you want to. For example, you can set the font color of the \texttt{XAxis} ticklabels using the \texttt{Axes} helper method:

```python
for label in ax.get_xticklabels():
    label.set_color('orange')
```

Below is a summary of the Artists that the \texttt{Axes} contains.

<table>
<thead>
<tr>
<th>Axes attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{artists}</td>
<td>A list of Artist instances</td>
</tr>
<tr>
<td>\texttt{patch}</td>
<td>Rectangle instance for Axes background</td>
</tr>
<tr>
<td>\texttt{collections}</td>
<td>A list of Collection instances</td>
</tr>
<tr>
<td>\texttt{images}</td>
<td>A list of AxesImage</td>
</tr>
<tr>
<td>\texttt{legends}</td>
<td>A list of Legend instances</td>
</tr>
<tr>
<td>\texttt{lines}</td>
<td>A list of Line2D instances</td>
</tr>
<tr>
<td>\texttt{patches}</td>
<td>A list of Patch instances</td>
</tr>
<tr>
<td>\texttt{texts}</td>
<td>A list of Text instances</td>
</tr>
<tr>
<td>\texttt{xaxis}</td>
<td>matplotlib.axis.XAxis instance</td>
</tr>
<tr>
<td>\texttt{yaxis}</td>
<td>matplotlib.axis.YAxis instance</td>
</tr>
</tbody>
</table>

**Axis containers**

The \texttt{matplotlib.axis.Axis} instances handle the drawing of the tick lines, the grid lines, the tick labels and the axis label. You can configure the left and right ticks separately for the y-axis, and the upper and lower ticks separately for the x-axis. The \texttt{Axis} also stores the data and view intervals used in auto-scaling, panning and zooming, as well as the \texttt{Locator} and \texttt{Formatter} instances which control where the ticks are placed and how they are represented as strings.

Each \texttt{Axis} object contains a \texttt{label} attribute (this is what \texttt{pylab} modifies in calls to \texttt{xlabel()} and \texttt{ylabel()}).
ylabel() as well as a list of major and minor ticks. The ticks are \texttt{XTick} and \texttt{YTick} instances, which contain the actual line and text primitives that render the ticks and ticklabels. Because the ticks are dynamically created as needed (e.g., when panning and zooming), you should access the lists of major and minor ticks through their accessor methods \texttt{get\_major\_ticks()} and \texttt{get\_minor\_ticks()}. Although the ticks contain all the primitives and will be covered below, \texttt{Axis} instances have accessor methods that return the tick lines, tick labels, tick locations etc.:

\begin{Verbatim}
In [285]: axis = ax.xaxis

In [286]: axis.get_ticklocs()
Out[286]: array([ 0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])

In [287]: axis.get_ticklabels()
Out[287]: <a list of 10 Text major ticklabel objects>

# note there are twice as many ticklines as labels because by
# default there are tick lines at the top and bottom but only tick
# labels below the xaxis; this can be customized
In [288]: axis.get_ticklines()
Out[288]: <a list of 20 Line2D ticklines objects>

# by default you get the major ticks back
In [291]: axis.get_ticklines()
Out[291]: <a list of 20 Line2D ticklines objects>

# but you can also ask for the minor ticks
In [292]: axis.get_ticklines(minor=True)
Out[292]: <a list of 0 Line2D ticklines objects>
\end{Verbatim}

Here is a summary of some of the useful accessor methods of the \texttt{Axis} (these have corresponding setters where useful, such as \texttt{set\_major\_formatter})

<table>
<thead>
<tr>
<th>Accessor method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_scale</td>
<td>The scale of the axis, e.g., ‘log’ or ‘linear’</td>
</tr>
<tr>
<td>get_view_interval</td>
<td>The interval instance of the axis view limits</td>
</tr>
<tr>
<td>get_data_interval</td>
<td>The interval instance of the axis data limits</td>
</tr>
<tr>
<td>get_gridlines</td>
<td>A list of grid lines for the Axis</td>
</tr>
<tr>
<td>get_label</td>
<td>The axis label - a Text instance</td>
</tr>
<tr>
<td>get_ticklabels</td>
<td>A list of Text instances - keyword minor=True</td>
</tr>
<tr>
<td>get_ticklines</td>
<td>A list of Line2D instances - keyword minor=True</td>
</tr>
<tr>
<td>get_ticklocs</td>
<td>A list of Tick locations - keyword minor=True</td>
</tr>
<tr>
<td>get_major_locator</td>
<td>The matplotlib.ticker.Locator instance for major ticks</td>
</tr>
<tr>
<td>get_major_formatter</td>
<td>The matplotlib.ticker.Formatter instance for major ticks</td>
</tr>
<tr>
<td>get_minor_locator</td>
<td>The matplotlib.ticker.Locator instance for minor ticks</td>
</tr>
<tr>
<td>get_minor_formatter</td>
<td>The matplotlib.ticker.Formatter instance for minor ticks</td>
</tr>
<tr>
<td>get_major_ticks</td>
<td>A list of Tick instances for major ticks</td>
</tr>
<tr>
<td>get_minor_ticks</td>
<td>A list of Tick instances for minor ticks</td>
</tr>
<tr>
<td>grid</td>
<td>Turn the grid on or off for the major or minor ticks</td>
</tr>
</tbody>
</table>

Here is an example, not recommended for its beauty, which customizes the axes and tick properties
import numpy as np
import matplotlib.pyplot as plt

# plt.figure creates a matplotlib.figure.Figure instance
fig = plt.figure()
rect = fig.patch  # a rectangle instance
rect.set_facecolor('lightgoldenrodyellow')

ax1 = fig.add_axes([0.1, 0.3, 0.4, 0.4])
rect = ax1.patch
rect.set_facecolor('lightslategrey')

for label in ax1.xaxis.get_ticklabels():  # label is a Text instance
    label.set_color('red')
    label.set_rotation(45)
    label.set_fontsize(16)

for line in ax1.yaxis.get_ticklines():  # line is a Line2D instance
    line.set_color('green')
    line.set_markersize(25)
    line.set_markeredgewidth(3)

plt.show()
Tick containers

The `matplotlib.axis.Tick` is the final container object in our descent from the `Figure` to the `Axes` to the `Axis` to the `Tick`. The `Tick` contains the tick and grid line instances, as well as the label instances for the upper and lower ticks. Each of these is accessible directly as an attribute of the `Tick`. In addition, there are boolean variables that determine whether the upper labels and ticks are on for the x-axis and whether the right labels and ticks are on for the y-axis.

<table>
<thead>
<tr>
<th>Tick attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tick1line</td>
<td>Line2D instance</td>
</tr>
<tr>
<td>tick2line</td>
<td>Line2D instance</td>
</tr>
<tr>
<td>gridline</td>
<td>Line2D instance</td>
</tr>
<tr>
<td>label1</td>
<td>Text instance</td>
</tr>
<tr>
<td>label2</td>
<td>Text instance</td>
</tr>
<tr>
<td>gridOn</td>
<td>boolean which determines whether to draw the gridline</td>
</tr>
<tr>
<td>tick1On</td>
<td>boolean which determines whether to draw the 1st tickline</td>
</tr>
<tr>
<td>tick2On</td>
<td>boolean which determines whether to draw the 2nd tickline</td>
</tr>
<tr>
<td>label1On</td>
<td>boolean which determines whether to draw the 1st tick label</td>
</tr>
<tr>
<td>label2On</td>
<td>boolean which determines whether to draw the 2nd tick label</td>
</tr>
</tbody>
</table>

Here is an example which sets the formatter for the right side ticks with dollar signs and colors them green on the right side of the yaxis.
```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker

# Fixing random state for reproducibility
np.random.seed(19680801)

fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(100*np.random.rand(20))

formatter = ticker.FormatStrFormatter('$%1.2f')
ax.yaxis.set_major_formatter(formatter)

for tick in ax.yaxis.get_major_ticks():
    tick.label1On = False
    tick.label2On = True
    tick.label2.set_color('green')

plt.show()
```
3.2.2 Legend guide

This legend guide is an extension of the documentation available at `legend()` - please ensure you are familiar with contents of that documentation before proceeding with this guide.

This guide makes use of some common terms, which are documented here for clarity:

- **legend entry**  A legend is made up of one or more legend entries. An entry is made up of exactly one key and one label.
- **legend key**    The colored/patterned marker to the left of each legend label.
- **legend label**  The text which describes the handle represented by the key.
- **legend handle** The original object which is used to generate an appropriate entry in the legend.

Controlling the legend entries

Calling `legend()` with no arguments automatically fetches the legend handles and their associated labels. This functionality is equivalent to:

```python
handles, labels = ax.get_legend_handles_labels()
ax.legend(handles, labels)
```

The `get_legend_handles_labels()` function returns a list of handles/artists which exist on the Axes which can be used to generate entries for the resulting legend - it is worth noting however that not all artists can be added to a legend, at which point a “proxy” will have to be created (see Creating artists specifically for adding to the legend (aka. Proxy artists) for further details).

For full control of what is being added to the legend, it is common to pass the appropriate handles directly to `legend()`:

```python
line_up, = plt.plot([1,2,3], label='Line 2')
line_down, = plt.plot([3,2,1], label='Line 1')
plt.legend(handles=[line_up, line_down])
```

In some cases, it is not possible to set the label of the handle, so it is possible to pass through the list of labels to `legend()`:

```python
line_up, = plt.plot([1,2,3], label='Line 2')
line_down, = plt.plot([3,2,1], label='Line 1')
plt.legend([line_up, line_down], ['Line Up', 'Line Down'])
```

Creating artists specifically for adding to the legend (aka. Proxy artists)

Not all handles can be turned into legend entries automatically, so it is often necessary to create an artist which can. Legend handles don’t have to exists on the Figure or Axes in order to be used.

Suppose we wanted to create a legend which has an entry for some data which is represented by a red color:
import matplotlib.patches as mpatches
import matplotlib.pyplot as plt

red_patch = mpatches.Patch(color='red', label='The red data')
plt.legend(handles=[red_patch])
plt.show()

There are many supported legend handles, instead of creating a patch of color we could have created a line with a marker:

import matplotlib.lines as mlines
import matplotlib.pyplot as plt

blue_line = mlines.Line2D([], [], color='blue', marker='*',
                          markersize=15, label='Blue stars')
plt.legend(handles=[blue_line])
plt.show()
**Legend location**

The location of the legend can be specified by the keyword argument `loc`. Please see the documentation at `legend()` for more details.

The `bbox_to_anchor` keyword gives a great degree of control for manual legend placement. For example, if you want your axes legend located at the figure’s top right-hand corner instead of the axes’ corner, simply specify the corner’s location, and the coordinate system of that location:

```python
g = plt.legend(bbox_to_anchor=(1, 1),
               bbox_transform=plt.gcf().transFigure)
```

More examples of custom legend placement:

```python
import matplotlib.pyplot as plt

plt.subplot(211)
plt.plot([1, 2, 3], label="test1")
plt.plot([3, 2, 1], label="test2")
# Place a legend above this subplot, expanding itself to
# fully use the given bounding box.
g = plt.legend(bbox_to_anchor=(0., 1.02, 1., .102), loc=3,
               ncol=2, mode="expand", borderaxespad=0.)
```
plt.subplot(223)
plt.plot([1,2,3], label="test1")
plt.plot([3,2,1], label="test2")

# Place a legend to the right of this smaller subplot.
plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)

plt.show()

---

**Multiple legends on the same Axes**

Sometimes it is more clear to split legend entries across multiple legends. Whilst the instinctive approach
to doing this might be to call the `legend()` function multiple times, you will find that only one legend ever
exists on the Axes. This has been done so that it is possible to call `legend()` repeatedly to update the legend
to the latest handles on the Axes, so to persist old legend instances, we must add them manually to the Axes:

```python
import matplotlib.pyplot as plt

line1, = plt.plot([1,2,3], label="Line 1", linestyle='--')
line2, = plt.plot([3,2,1], label="Line 2", linewidth=4)

# Create a legend for the first line.
first_legend = plt.legend(handles=[line1], loc=1)
```

---

3.2. Intermediate
# Add the legend manually to the current Axes.
ax = plt.gca().add_artist(first_legend)

# Create another legend for the second line.
plt.legend(handles=[line2], loc=4)

plt.show()

Legend Handlers

In order to create legend entries, handles are given as an argument to an appropriate `HandlerBase` subclass. The choice of handler subclass is determined by the following rules:

1. Update `get_legend_handler_map()` with the value in the `handler_map` keyword.
2. Check if the handle is in the newly created `handler_map`.
3. Check if the type of handle is in the newly created `handler_map`.
4. Check if any of the types in the handle's mro is in the newly created `handler_map`.

For completeness, this logic is mostly implemented in `get_legend_handler()`.
All of this flexibility means that we have the necessary hooks to implement custom handlers for our own type of legend key.

The simplest example of using custom handlers is to instantiate one of the existing `HandlerBase` subclasses. For the sake of simplicity, let’s choose `matplotlib.legend_handler.HandlerLine2D` which accepts a `numpoints` argument (note `numpoints` is a keyword on the `legend()` function for convenience). We can then pass the mapping of instance to Handler as a keyword to `legend`.

```python
import matplotlib.pyplot as plt
from matplotlib.legend_handler import HandlerLine2D

line1, = plt.plot([3,2,1], marker='o', label='Line 1')
line2, = plt.plot([1,2,3], marker='o', label='Line 2')

plt.legend(handler_map={line1: HandlerLine2D(numpoints=4)})
```

As you can see, “Line 1” now has 4 marker points, where “Line 2” has 2 (the default). Try the above code, only change the map’s key from `line1` to `type(line1)`. Notice how now both `Line2D` instances get 4 markers.

Along with handlers for complex plot types such as errorbars, stem plots and histograms, the default `handler_map` has a special `tuple` handler (`HandlerTuple`) which simply plots the handles on top of one another for each item in the given tuple. The following example demonstrates combining two legend keys on top of one another:

```python
import matplotlib.pyplot as plt
from matplotlib.legend_handler import HandlerLine2D

line1, = plt.plot([3,2,1], marker='o', label='Line 1')
line2, = plt.plot([1,2,3], marker='o', label='Line 2')

plt.legend(handler_map={line1: HandlerLine2D(numpoints=4), line2: HandlerLine2D(numpoints=4)})
```
import matplotlib.pyplot as plt
from numpy.random import randn

z = randn(10)

red_dot, = plt.plot(z, "ro", markersize=15)
# Put a white cross over some of the data.
white_cross, = plt.plot(z[:5], "w+", markeredgewidth=3, markersize=15)

plt.legend([red_dot, (red_dot, white_cross)], ["Attr A", "Attr A+B"])

Implementing a custom legend handler

A custom handler can be implemented to turn any handle into a legend key (handles don’t necessarily need to be matplotlib artists). The handler must implement a "legend_artist" method which returns a single artist for the legend to use. Signature details about the "legend_artist" are documented at legend_artist().

import matplotlib.pyplot as plt
import matplotlib.patches as mpatches

class AnyObject(object):
    pass
```python
class AnyObjectHandler(object):
    def legend_artist(self, legend, orig_handle, fontsize, handlebox):
        x0, y0 = handlebox.xdescent, handlebox.ydescent
        width, height = handlebox.width, handlebox.height
        patch = mpatches.Rectangle([x0, y0], width, height, facecolor='red',
                                   edgecolor='black', hatch='xx', lw=3,
                                   transform=handlebox.get_transform())
        handlebox.add_artist(patch)
        return patch

plt.legend([AnyObject()], ['My first handler'],
            handler_map={AnyObject: AnyObjectHandler()})
```

Alternatively, had we wanted to globally accept AnyObject instances without needing to manually set the handler_map keyword all the time, we could have registered the new handler with:

```python
from matplotlib.legend import Legend
Legend.update_default_handler_map({AnyObject: AnyObjectHandler()})
```

Whilst the power here is clear, remember that there are already many handlers implemented and what you want to achieve may already be easily possible with existing classes. For example, to produce elliptical legend keys, rather than rectangular ones:

```python
from matplotlib.legend_handler import HandlerPatch
import matplotlib.pyplot as plt
```

3.2. Intermediate
import matplotlib.patches as mpatches

class HandlerEllipse(HandlerPatch):
    def create_artists(self, legend, orig_handle, xdescent, ydescent, width, height, fontsize, trans):
        center = 0.5 * width - 0.5 * xdescent, 0.5 * height - 0.5 * ydescent
        p = mpatches.Ellipse(xy=center, width=width + xdescent, height=height + ydescent)
        self.update_prop(p, orig_handle, legend)
        p.set_transform(trans)
        return [p]

c = mpatches.Circle((0.5, 0.5), 0.25, facecolor="green", edgecolor="red", linewidth=3)
plt.gca().add_patch(c)
plt.legend([c], ["An ellipse, not a rectangle"], handler_map={mpatches.Circle: HandlerEllipse()})

Known examples of using legend

Here is a non-exhaustive list of the examples available involving legend being used in various ways:
• lines_bars_and_markers example code: scatter_with_legend.py
• api example code: legend_demo.py
• pylab_examples example code: contourf_hatching.py
• pylab_examples example code: figlegend_demo.py
• pylab_examples example code: scatter_symbol.py

3.3 Advanced

3.3.1 Transformations Tutorial

Like any graphics packages, matplotlib is built on top of a transformation framework to easily move between coordinate systems, the userland data coordinate system, the axes coordinate system, the figure coordinate system, and the display coordinate system. In 95% of your plotting, you won’t need to think about this, as it happens under the hood, but as you push the limits of custom figure generation, it helps to have an understanding of these objects so you can reuse the existing transformations matplotlib makes available to you, or create your own (see matplotlib.transforms). The table below summarizes the existing coordinate systems, the transformation object you should use to work in that coordinate system, and the description of that system. In the Transformation Object column, ax is a Axes instance, and fig is a Figure instance.

<table>
<thead>
<tr>
<th>Coordinate</th>
<th>Transformation Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>ax.transData</td>
<td>The userland data coordinate system, controlled by the xlim and ylim</td>
</tr>
<tr>
<td>axes</td>
<td>ax.transAxes</td>
<td>The coordinate system of the Axes; (0,0) is bottom left of the axes, and (1,1) is top right of the axes.</td>
</tr>
<tr>
<td>figure</td>
<td>fig.transFigure</td>
<td>The coordinate system of the Figure; (0,0) is bottom left of the figure, and (1,1) is top right of the figure.</td>
</tr>
<tr>
<td>display</td>
<td>None</td>
<td>This is the pixel coordinate system of the display; (0,0) is the bottom left of the display, and (width, height) is the top right of the display in pixels. Alternatively, the identity transform (matplotlib.transforms.IdentityTransform()) may be used instead of None.</td>
</tr>
</tbody>
</table>

All of the transformation objects in the table above take inputs in their coordinate system, and transform the input to the display coordinate system. That is why the display coordinate system has None for the Transformation Object column – it already is in display coordinates. The transformations also know how to invert themselves, to go from display back to the native coordinate system. This is particularly useful when processing events from the user interface, which typically occur in display space, and you want to know where the mouse click or key-press occurred in your data coordinate system.
Data coordinates

Let’s start with the most commonly used coordinate, the data coordinate system. Whenever you add data to the axes, matplotlib updates the datalimits, most commonly updated with the `set_xlim()` and `set_ylim()` methods. For example, in the figure below, the data limits stretch from 0 to 10 on the x-axis, and -1 to 1 on the y-axis.

```python
import numpy as np
import matplotlib.pyplot as plt

x = np.arange(0, 10, 0.005)
y = np.exp(-x/2.) * np.sin(2*np.pi*x)

fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(x, y)
ax.set_xlim(0, 10)
ax.set_ylim(-1, 1)
plt.show()
```

You can use the `ax.transData` instance to transform from your data to your display coordinate system, either a single point or a sequence of points as shown below:
In [14]: type(ax.transData)
Out[14]: <class 'matplotlib.transforms.CompositeGenericTransform'>

In [15]: ax.transData.transform((5, 0))
Out[15]: array([ 335.175,  247. ])

In [16]: ax.transData.transform(((5, 0), (1,2)))
Out[16]: array([[ 335.175,  247. ],
             [ 132.435,  642.2 ]])

You can use the `inverted()` method to create a transform which will take you from display to data coordinates:

In [41]: inv = ax.transData.inverted()

In [42]: type(inv)
Out[42]: <class 'matplotlib.transforms.CompositeGenericTransform'>

In [43]: inv.transform((335.175, 247.))
Out[43]: array([ 5.,  0.])

If you are typing along with this tutorial, the exact values of the display coordinates may differ if you have a different window size or dpi setting. Likewise, in the figure below, the display labeled points are probably not the same as in the ipython session because the documentation figure size defaults are different.
Note: If you run the source code in the example above in a GUI backend, you may also find that the two arrows for the data and display annotations do not point to exactly the same point. This is because the display point was computed before the figure was displayed, and the GUI backend may slightly resize the figure when it is created. The effect is more pronounced if you resize the figure yourself. This is one good reason why you rarely want to work in display space, but you can connect to the 'on_draw' Event to update figure coordinates on figure draws; see Event handling and picking.

When you change the x or y limits of your axes, the data limits are updated so the transformation yields a new display point. Note that when we just change the ylim, only the y-display coordinate is altered, and when we change the xlim too, both are altered. More on this later when we talk about the Bbox.

In [54]: ax.transData.transform((5, 0))
Out[54]: array([ 335.175,  247.  ])

In [55]: ax.set_ylim(-1,2)
Out[55]: (-1, 2)

In [56]: ax.transData.transform((5, 0))
Out[56]: array([ 335.175 , 181.13333333])

In [57]: ax.set_xlim(10,20)
Out[57]: (10, 20)

In [58]: ax.transData.transform((5, 0))
Out[58]: array([-171.675 , 181.13333333])

Axes coordinates

After the data coordinate system, axes is probably the second most useful coordinate system. Here the point (0,0) is the bottom left of your axes or subplot, (0.5, 0.5) is the center, and (1.0, 1.0) is the top right. You can also refer to points outside the range, so (-0.1, 1.1) is to the left and above your axes. This coordinate system is extremely useful when placing text in your axes, because you often want a text bubble in a fixed, location, e.g., the upper left of the axes pane, and have that location remain fixed when you pan or zoom. Here is a simple example that creates four panels and labels them ‘A’, ‘B’, ‘C’, ‘D’ as you often see in journals.

import numpy as np
import matplotlib.pyplot as plt

fig = plt.figure()
for i, label in enumerate(('A', 'B', 'C', 'D')):
    ax = fig.add_subplot(2,2,i+1)
    ax.text(0.05, 0.95, label, transform=ax.transAxes,
            fontsize=16, fontweight='bold', va='top')

plt.show()
You can also make lines or patches in the axes coordinate system, but this is less useful in my experience than using `ax.transAxes` for placing text. Nonetheless, here is a silly example which plots some random dots in data space, and overlays a semi-transparent `Circle` centered in the middle of the axes with a radius one quarter of the axes – if your axes does not preserve aspect ratio (see `set_aspect()`, this will look like an ellipse. Use the pan/zoom tool to move around, or manually change the data xlim and ylim, and you will see the data move, but the circle will remain fixed because it is not in data coordinates and will always remain at the center of the axes.

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches

fig = plt.figure()
ax = fig.add_subplot(111)
x, y = 10*rand(1000)
ax.plot(x, y, 'go')  # plot some data in data coordinates

circ = patches.Circle((0.5, 0.5), 0.25, transform=ax.transAxes,
                       facecolor='yellow', alpha=0.5)
ax.add_patch(circ)

plt.show()
```

3.3. Advanced 81
Blended transformations

Drawing in blended coordinate spaces which mix axes with data coordinates is extremely useful, for example to create a horizontal span which highlights some region of the y-data but spans across the x-axis regardless of the data limits, pan or zoom level, etc. In fact these blended lines and spans are so useful, we have built in functions to make them easy to plot (see `axhline()`, `axvline()`, `axhspan()`, `axvspan()`), but for didactic purposes we will implement the horizontal span here using a blended transformation. This trick only works for separable transformations, like you see in normal Cartesian coordinate systems, but not on inseparable transformations like the `PolarTransform`.

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import matplotlib.transforms as transforms

fig = plt.figure()
ax = fig.add_subplot(111)

x = np.random.randn(1000)
ax.hist(x, 30)
ax.set_title(r'$\sigma=1 \ldots \sigma=2$', fontsize=16)

# the x coords of this transformation are data, and the
# y coord are axes
trans = transforms.blended_transform_factory(ax.transData, ax.transAxes)

# highlight the 1..2 stddev region with a span.
# We want x to be in data coordinates and y to
# span from 0..1 in axes coords
rect = patches.Rectangle((1,0), width=1, height=1,
    transform=trans, color='yellow',
    alpha=0.5)

ax.add_patch(rect)
plt.show()

Note: The blended transformations where x is in data coords and y in axes coordinates is so useful that we have helper methods to return the versions mpl uses internally for drawing ticks, ticklabels, etc. The methods are matplotlib.axes.Axes.get_xaxis_transform() and matplotlib.axes.Axes.get_yaxis_transform(). So in the example above, the call to blended_transform_factory() can be replaced by get_xaxis_transform:

trans = ax.get_xaxis_transform()
**Using offset transforms to create a shadow effect**

One use of transformations is to create a new transformation that is offset from another transformation, e.g., to place one object shifted a bit relative to another object. Typically you want the shift to be in some physical dimension, like points or inches rather than in data coordinates, so that the shift effect is constant at different zoom levels and dpi settings.

One use for an offset is to create a shadow effect, where you draw one object identical to the first just to the right of it, and just below it, adjusting the zorder to make sure the shadow is drawn first and then the object it is shadowing above it. The transforms module has a helper transformation `ScaledTranslation`. It is instantiated with:

```python
trans = ScaledTranslation(xt, yt, scale_trans)
```

where `xt` and `yt` are the translation offsets, and `scale_trans` is a transformation which scales `xt` and `yt` at transformation time before applying the offsets. A typical use case is to use the figure `fig.dpi_scale_trans` transformation for the `scale_trans` argument, to first scale `xt` and `yt` specified in points to display space before doing the final offset. The dpi and inches offset is a common-enough use case that we have a special helper function to create it in `matplotlib.transforms.offset_copy()`, which returns a new transform with an added offset. But in the example below, we’ll create the offset transform ourselves. Note the use of the plus operator in:

```python
offset = transforms.ScaledTranslation(dx, dy, fig.dpi_scale_trans)
shadow_transform = ax.transData + offset
```

showing that can chain transformations using the addition operator. This code says: first apply the data transformation `ax.transData` and then translate the data by `dx` and `dy` points. In typography, a'point' is 1/72 inches, and by specifying your offsets in points, your figure will look the same regardless of the dpi resolution it is saved in.

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import matplotlib.transforms as transforms

fig = plt.figure()
ax = fig.add_subplot(111)

# make a simple sine wave
x = np.arange(0., 2., 0.01)
y = np.sin(2*np.pi*x)
line, = ax.plot(x, y, lw=3, color='blue')

# shift the object over 2 points, and down 2 points
dx, dy = 2/72., -2/72.
offset = transforms.ScaledTranslation(dx, dy, fig.dpi_scale_trans)
shadow_transform = ax.transData + offset

# now plot the same data with our offset transform;
# use the zorder to make sure we are below the line
```
The transformation pipeline

The `ax.transData` transform we have been working with in this tutorial is a composite of three different transformations that comprise the transformation pipeline from data -> display coordinates. Michael Droettboom implemented the transformations framework, taking care to provide a clean API that segregated the nonlinear projections and scales that happen in polar and logarithmic plots, from the linear affine transformations that happen when you pan and zoom. There is an efficiency here, because you can pan and zoom in your axes which affects the affine transformation, but you may not need to compute the potentially expensive nonlinear scales or projections on simple navigation events. It is also possible to multiply affine transformation matrices together, and then apply them to coordinates in one step. This is not true of all possible transformations.

Here is how the `ax.transData` instance is defined in the basic separable axis `Axes` class:

```
self.transData = self.transScale + (self.transLimits + self.transAxes)
```
We’ve been introduced to the `transAxes` instance above in *Axes coordinates*, which maps the (0,0), (1,1) corners of the axes or subplot bounding box to display space, so let’s look at these other two pieces.

`self.transLimits` is the transformation that takes you from *data* to *axes* coordinates; i.e., it maps your view `xlim` and `ylim` to the unit space of the axes (and `transAxes` then takes that unit space to display space). We can see this in action here:

```
In [80]: ax = subplot(111)
In [81]: ax.set_xlim(0, 10)
Out[81]: (0, 10)
In [82]: ax.set_ylim(-1,1)
Out[82]: (-1, 1)
In [84]: ax.transLimits.transform((0,-1))
Out[84]: array([ 0., 0.])
In [85]: ax.transLimits.transform((10,-1))
Out[85]: array([ 1., 0.])
In [86]: ax.transLimits.transform((10,1))
Out[86]: array([ 1., 1.])
In [87]: ax.transLimits.transform((5,0))
Out[87]: array([ 0.5, 0.5])
```

and we can use this same inverted transformation to go from the unit *axes* coordinates back to *data* coordinates.

```
In [90]: inv.transform((0.25, 0.25))
Out[90]: array([ 2.5, -0.5])
```

The final piece is the `self.transScale` attribute, which is responsible for the optional non-linear scaling of the data, e.g., for logarithmic axes. When an Axes is initially setup, this is just set to the identity transform, since the basic matplotlib axes has linear scale, but when you call a logarithmic scaling function like `semilogx()` or explicitly set the scale to logarithmic with `set_xscale()`, then the `ax.transScale` attribute is set to handle the nonlinear projection. The scales transforms are properties of the respective `xaxis` and `yaxis` `Axis` instances. For example, when you call `ax.set_xscale('log')`, the xaxis updates its scale to a `matplotlib.scale.LogScale` instance.

For non-separable axes the `PolarAxes`, there is one more piece to consider, the projection transformation. The `transData` `matplotlib.projections.polar.PolarAxes` is similar to that for the typical separable matplotlib Axes, with one additional piece `transProjection`:

```
```

`transProjection` handles the projection from the space, e.g., latitude and longitude for map data, or radius and theta for polar data, to a separable Cartesian coordinate system. There are several projection examples in the `matplotlib.projections` package, and the best way to learn more is to open the source for those packages and see how to make your own, since matplotlib supports extensible axes and projections. Michael
Droettboom has provided a nice tutorial example of creating a hammer projection axes; see *api example code: custom_projection_example.py*.

### 3.3.2 Path Tutorial

The object underlying all of the `matplotlib.patch` objects is the *Path*, which supports the standard set of moveto, lineto, curveto commands to draw simple and compound outlines consisting of line segments and splines. The *Path* is instantiated with a (N,2) array of (x,y) vertices, and a N-length array of path codes. For example to draw the unit rectangle from (0,0) to (1,1), we could use this code

```python
import matplotlib.pyplot as plt
from matplotlib.path import Path
import matplotlib.patches as patches

verts = [
    (0., 0.), # left, bottom
    (0., 1.), # left, top
    (1., 1.), # right, top
    (1., 0.), # right, bottom
    (0., 0.), # ignored
]

codes = [Path.MOVETO,
    Path.LINETO,
    Path.LINETO,
    Path.LINETO,
    Path.CLOSEPOLY,
]

path = Path(verts, codes)

fig = plt.figure()
ax = fig.add_subplot(111)
patch = patches.PathPatch(path, facecolor='orange', lw=2)
ax.add_patch(patch)
ax.set_xlim(-2,2)
ax.set_ylim(-2,2)
plt.show()
```
The following path codes are recognized

<table>
<thead>
<tr>
<th>Code</th>
<th>Vertices</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STOP</td>
<td>1 (ignored)</td>
<td>A marker for the end of the entire path (currently not required and ignored)</td>
</tr>
<tr>
<td>MOVETO</td>
<td>1</td>
<td>Pick up the pen and move to the given vertex.</td>
</tr>
<tr>
<td>LINETO</td>
<td>1</td>
<td>Draw a line from the current position to the given vertex.</td>
</tr>
<tr>
<td>CURVE3</td>
<td>2 (1 control point, 1 endpoint)</td>
<td>Draw a quadratic Bézier curve from the current position, with the given control point, to the given end point.</td>
</tr>
<tr>
<td>CURVE4</td>
<td>3 (2 control points, 1 endpoint)</td>
<td>Draw a cubic Bézier curve from the current position, with the given control points, to the given end point.</td>
</tr>
<tr>
<td>CLOSEPOLY</td>
<td>point itself is ignored</td>
<td>Draw a line segment to the start point of the current polyline.</td>
</tr>
</tbody>
</table>

**Bézier example**

Some of the path components require multiple vertices to specify them: for example CURVE 3 is a bézier curve with one control point and one end point, and CURVE4 has three vertices for the two control points and the end point. The example below shows a CURVE4 Bézier spline – the bézier curve will be contained in the convex hull of the start point, the two control points, and the end point.

```python
import matplotlib.pyplot as plt
from matplotlib.path import Path
```
import matplotlib.patches as patches

verts = [
    (0., 0.), # P0
    (0.2, 1.), # P1
    (1., 0.8), # P2
    (0.8, 0.), # P3
]

codes = [Path.MOVETO,
    Path.CURVE4,
    Path.CURVE4,
    Path.CURVE4,
]

path = Path(verts, codes)

fig = plt.figure()
ax = fig.add_subplot(111)
patch = patches.PathPatch(path, facecolor='none', lw=2)
ax.add_patch(patch)

xs, ys = zip(*verts)
ax.plot(xs, ys, 'x--', lw=2, color='black', ms=10)

ax.text(-0.05, -0.05, 'P0')
ax.text(0.15, 1.05, 'P1')
ax.text(1.05, 0.85, 'P2')
ax.text(0.85, -0.05, 'P3')

ax.set_xlim(-0.1, 1.1)
ax.set_ylim(-0.1, 1.1)
plt.show()
Compound paths

All of the simple patch primitives in matplotlib, Rectangle, Circle, Polygon, etc, are implemented with simple path. Plotting functions like `hist()` and `bar()`, which create a number of primitives, e.g., a bunch of Rectangles, can usually be implemented more efficiently using a compound path. The reason `bar()` creates a list of rectangles and not a compound path is largely historical: the `Path` code is comparatively new and `bar()` predates it. While we could change it now, it would break old code, so here we will cover how to create compound paths, replacing the functionality in `bar`, in case you need to do so in your own code for efficiency reasons, e.g., you are creating an animated bar plot.

We will make the histogram chart by creating a series of rectangles for each histogram bar: the rectangle width is the bin width and the rectangle height is the number of datapoints in that bin. First we'll create some random normally distributed data and compute the histogram. Because numpy returns the bin edges and not centers, the length of `bins` is 1 greater than the length of `n` in the example below:

```python
# histogram our data with numpy
data = np.random.randn(1000)
n, bins = np.histogram(data, 100)
```

We'll now extract the corners of the rectangles. Each of the `left`, `bottom`, etc, arrays below is `len(n)`, where `n` is the array of counts for each histogram bar:
# get the corners of the rectangles for the histogram
left = np.array(bins[:-1])
right = np.array(bins[1:])
bottom = np.zeros(len(left))
top = bottom + n

Now we have to construct our compound path, which will consist of a series of MOVETO, LINETO and CLOSEPOLY for each rectangle. For each rectangle, we need 5 vertices: 1 for the MOVETO, 3 for the LINETO, and 1 for the CLOSEPOLY. As indicated in the table above, the vertex for the closepoly is ignored but we still need it to keep the codes aligned with the vertices:

```python
nverts = nrects*(1+3+1)
verts = np.zeros((nverts, 2))
codes = np.ones(nverts, int) * path.Path.LINETO
codes[0::5] = path.Path.MOVETO
codes[4::5] = path.Path.CLOSEPOLY
verts[0::5,0] = left
verts[0::5,1] = bottom
verts[1::5,0] = left
verts[1::5,1] = top
verts[2::5,0] = right
verts[2::5,1] = top
verts[3::5,0] = right
verts[3::5,1] = bottom
```

All that remains is to create the path, attach it to a PathPatch, and add it to our axes:

```python
barpath = path.Path(verts, codes)
patch = patches.PathPatch(barpath, facecolor='green',
    edgecolor='yellow', alpha=0.5)
ax.add_patch(patch)
```

Here is the result
3.3.3 Path effects guide

Matplotlib’s `path_effects` module provides functionality to apply a multiple draw stage to any Artist which can be rendered via a `Path`.

Artists which can have a path effect applied to them include `Patch`, `Line2D`, `Collection` and even `Text`. Each artist’s path effects can be controlled via the `set_path_effects` method (`set_path_effects`), which takes an iterable of `AbstractPathEffect` instances.

The simplest path effect is the `Normal` effect, which simply draws the artist without any effect:

```python
import matplotlib.pyplot as plt
import matplotlib.path as path
fig = plt.figure(figsize=(5, 1.5))
text = fig.text(0.5, 0.5, 'Hello path effects world!
This is the normal path effect.
Pretty dull, huh?',
                ha='center', va='center', size=20)
text.set_path_effects([path.PathEffect.Normal()])
plt.show()
```
Hello path effects world!
This is the normal path effect.
Pretty dull, huh?

Whilst the plot doesn’t look any different to what you would expect without any path effects, the drawing of the text now been changed to use the path effects framework, opening up the possibilities for more interesting examples.

Adding a shadow

A far more interesting path effect than Normal is the drop-shadow, which we can apply to any of our path based artists. The classes SimplePatchShadow and SimpleLineShadow do precisely this by drawing either a filled patch or a line patch below the original artist:

```python
import matplotlib.pyplot as plt
import matplotlib.pathEffects as path_effects

text = plt.text(0.5, 0.5, 'Hello path effects world!',
                        path_effects=[path_effects.withSimplePatchShadow()])
plt.plot([0, 3, 2, 5], linewidth=5, color='blue',
                        path_effects=[path_effects.SimpleLineShadow(),
                                      path_effects.Normal()])
plt.show()
```
Notice the two approaches to setting the path effects in this example. The first uses the `with*` classes to include the desired functionality automatically followed with the “normal” effect, whereas the latter explicitly defines the two path effects to draw.

**Making an artist stand out**

One nice way of making artists visually stand out is to draw an outline in a bold color below the actual artist. The *Stroke* path effect makes this a relatively simple task:

```python
import matplotlib.pyplot as plt
import matplotlib.pathEffects as path_effects

fig = plt.figure(figsize=(7, 1))

text = fig.text(0.5, 0.5, 'This text stands out because of its black border.', color='white', ha='center', va='center', size=30)

text.set_path_effects([path_effects.Stroke(linewidth=3, foreground='black'), path_effects.Normal()])

plt.show()
```
This text stands out because of its black border.

It is important to note that this effect only works because we have drawn the text path twice; once with a thick black line, and then once with the original text path on top.

You may have noticed that the keywords to Stroke and SimplePatchShadow and SimpleLineShadow are not the usual Artist keywords (such as facecolor and edgecolor etc.). This is because with these path effects we are operating at lower level of matplotlib. In fact, the keywords which are accepted are those for a matplotlib.backend_bases.GraphicsContextBase instance, which have been designed for making it easy to create new backends - and not for its user interface.

**Greater control of the path effect artist**

As already mentioned, some of the path effects operate at a lower level than most users will be used to, meaning that setting keywords such as facecolor and edgecolor raise an AttributeError. Luckily there is a generic PathPatchEffect path effect which creates a PathPatch class with the original path. The keywords to this effect are identical to those of PathPatch:

```python
import matplotlib.pyplot as plt
import matplotlib.patheffects as path_effects

fig = plt.figure(figsize=(8, 1))
t = fig.text(0.02, 0.5, 'Hatch shadow', fontsize=75, weight=1000, va='center')
t.set_path_effects([path_effects.PathPatchEffect(offset=(4, -4), hatch='xxxx', facecolor='gray'), path_effects.PathPatchEffect(edgecolor='white', linewidth=1.1, facecolor='black')])

plt.show()
```

Hatch shadow
4.1 Text introduction

matplotlib has extensive text support, including support for mathematical expressions, truetype support for raster and vector outputs, newline separated text with arbitrary rotations, and unicode support. Because it embeds fonts directly in output documents, e.g., for postscript or PDF, what you see on the screen is what you get in the hardcopy. FreeType support produces very nice, antialiased fonts, that look good even at small raster sizes. matplotlib includes its own matplotlib.font_manager (thanks to Paul Barrett), which implements a cross platform, W3C compliant font finding algorithm.

The user has a great deal of control over text properties (font size, font weight, text location and color, etc.) with sensible defaults set in the rc file. And significantly, for those interested in mathematical or scientific figures, matplotlib implements a large number of TeX math symbols and commands, supporting mathematical expressions anywhere in your figure.

4.2 Basic text commands

The following commands are used to create text in the pyplot interface

- `text()` - add text at an arbitrary location to the Axes; matplotlib.axes.Axes.text() in the API.
- `xlabel()` - add a label to the x-axis; matplotlib.axes.Axes.set_xlabel() in the API.
- `ylabel()` - add a label to the y-axis; matplotlib.axes.Axes.set_ylabel() in the API.
- `title()` - add a title to the Axes; matplotlib.axes.Axes.set_title() in the API.
- `figtext()` - add text at an arbitrary location to the Figure; matplotlib.figure.Figure.text() in the API.
- `suptitle()` - add a title to the Figure; matplotlib.figure.Figure.suptitle() in the API.
- `annotate()` - add an annotation, with optional arrow, to the Axes; matplotlib.axes.Axes.annotate() in the API.

All of these functions create and return a matplotlib.text.Text() instance, which can be configured with a variety of font and other properties. The example below shows all of these commands in action.
import matplotlib.pyplot as plt

fig = plt.figure()
fig.suptitle('bold figure suptitle', fontsize=14, fontweight='bold')

ax = fig.add_subplot(111)
fig.subplots_adjust(top=0.85)
ax.set_title('axes title')
ax.set_xlabel('xlabel')
ax.set_ylabel('ylabel')

ax.text(3, 8, 'boxed italics text in data coords', style='italic',
        bbox={'facecolor':'red', 'alpha':0.5, 'pad':10})

ax.text(2, 6, r'an equation: $E=mc^2$', fontsize=15)

ax.text(3, 2, u'unicode: Institut f\374r Festk\366rperphysik')

ax.text(0.95, 0.01, 'colored text in axes coords',
        verticalalignment='bottom', horizontalalignment='right',
        transform=ax.transAxes,
        color='green', fontsize=15)

ax.plot([2], [1], 'o')
ax.annotate('annotate', xy=(2, 1), xytext=(3, 4),
            arrowprops=dict(facecolor='black', shrink=0.05))

ax.axis([0, 10, 0, 10])

plt.show()
4.3 Text properties and layout

The `matplotlib.text.Text` instances have a variety of properties which can be configured via keyword arguments to the text commands (e.g., `title()`, `xlabel()` and `text()`).
You can lay out text with the alignment arguments horizontalalignment, verticalalignment, and multialignment. horizontalalignment controls whether the x positional argument for the text indicates the left, center or right side of the text bounding box. verticalalignment controls whether the y positional argument for the text indicates the bottom, center or top side of the text bounding box. multialignment, for newline separated strings only, controls whether the different lines are left, center or right justified. Here is an example which uses the text() command to show the various alignment possibilities. The use of transform=ax.transAxes throughout the code indicates that the coordinates are given relative to the axes bounding box, with 0,0 being the lower left of the axes and 1,1 the upper right.

```python
import matplotlib.pyplot as plt
import matplotlib.patches as patches

# build a rectangle in axes coords
left, width = .25, .5
bottom, height = .25, .5
right = left + width
```
top = bottom + height

fig = plt.figure()
ax = fig.add_axes([0, 0, 1, 1])

# axes coordinates are 0,0 is bottom left and 1,1 is upper right
p = patches.Rectangle(
    (left, bottom), width, height,
    fill=False, transform=ax.transAxes, clip_on=False)

ax.add_patch(p)
ax.text(left, bottom, 'left top',
        horizontalalignment='left',
        verticalalignment='top',
        transform=ax.transAxes)

ax.text(left, bottom, 'left bottom',
        horizontalalignment='left',
        verticalalignment='bottom',
        transform=ax.transAxes)

ax.text(right, top, 'right bottom',
        horizontalalignment='right',
        verticalalignment='bottom',
        transform=ax.transAxes)

ax.text(right, top, 'right top',
        horizontalalignment='right',
        verticalalignment='top',
        transform=ax.transAxes)

ax.text(right, bottom, 'center top',
        horizontalalignment='center',
        verticalalignment='top',
        transform=ax.transAxes)

ax.text(left, 0.5*(bottom+top), 'right center',
        horizontalalignment='right',
        verticalalignment='center',
        rotation='vertical',
        transform=ax.transAxes)

ax.text(left, 0.5*(bottom+top), 'left center',
        horizontalalignment='left',
        verticalalignment='center',
        rotation='vertical',
        transform=ax.transAxes)

ax.text(0.5*(left+right), 0.5*(bottom+top), 'middle',
        horizontalalignment='center',
        verticalalignment='center',

4.3. Text properties and layout
4.4 Default Font

The base default font is controlled by a set of rcParams:
The mapping between the family aliases ("cursive", "fantasy", "monospace", "sans", "sans serif", "sans-serif", "serif") and actual font names is controlled by the following rcParams:

<table>
<thead>
<tr>
<th>family alias</th>
<th>rcParam with mappings</th>
</tr>
</thead>
<tbody>
<tr>
<td>'serif'</td>
<td>'font.serif'</td>
</tr>
<tr>
<td>'monospace'</td>
<td>'font.monospace'</td>
</tr>
<tr>
<td>'fantasy'</td>
<td>'font.fantasy'</td>
</tr>
<tr>
<td>'cursive'</td>
<td>'font.cursive'</td>
</tr>
<tr>
<td>{&quot;sans&quot;, &quot;sans serif&quot;, &quot;sans-serif&quot;}</td>
<td>'font.sans-serif'</td>
</tr>
</tbody>
</table>

which are lists of font names.

### 4.4.1 Text with non-latin glyphs

As of v2.0 the default font contains glyphs for many western alphabets, but still does not cover all of the glyphs that may be required by mpl users. For example, DejaVu has no coverage of Chinese, Korean, or Japanese.

To set the default font to be one that supports the code points you need, prepend the font name to 'font.family' or the desired alias lists

```python
matplotlib.rcParams['font.sans-serif'] = ['Source Han Sans TW', 'sans-serif']
```

or set it in your .matplotlibrc file:

```ini
font.sans-serif: Source Han Sans TW, Arial, sans-serif
```

To control the font used on per-artist basis use the 'name', 'fontname' or 'fontproperties' kwargs documented above.

On linux, fc-list can be a useful tool to discover the font name; for example

```
$ fc-list :lang=zh family
Noto to Sans Mono CJK TC,Noto Sans Mono CJK TC Bold
Noto Sans CJK TC,Noto Sans CJK TC Medium
Noto Sans CJK TC,Noto Sans CJK TC DemiLight
```
lists all of the fonts that support Chinese.

### 4.5 Annotation

#### Table of Contents

- Annotation
  - Basic annotation
  - Advanced Annotation
    * Annotating with Text with Box
    * Annotating with Arrow
    * Placing Artist at the anchored location of the Axes
    * Using Complex Coordinates with Annotations
    * Using ConnectorPatch
    * Zoom effect between Axes
    * Define Custom BoxStyle

#### 4.5.1 Basic annotation

The uses of the basic `text()` will place text at an arbitrary position on the Axes. A common use case of text is to annotate some feature of the plot, and the `annotate()` method provides helper functionality to make annotations easy. In an annotation, there are two points to consider: the location being annotated represented by the argument `xy` and the location of the text `xytext`. Both of these arguments are `(x, y)` tuples.

```python
import numpy as np
import matplotlib.pyplot as plt

fig = plt.figure()
ax = fig.add_subplot(111)

t = np.arange(0.0, 5.0, 0.01)
s = np.cos(2*np.pi*t)
line, = ax.plot(t, s, lw=2)

ax.annotate('local max', xy=(2, 1), xytext=(3, 1.5),
            arrowprops=dict(facecolor='black', shrink=0.05),
            )
```
In this example, both the \textit{xy} (arrow tip) and \textit{xytext} locations (text location) are in data coordinates. There are a variety of other coordinate systems one can choose – you can specify the coordinate system of \textit{xy} and \textit{xytext} with one of the following strings for \textit{xycoords} and \textit{textcoords} (default is ‘data’)

<table>
<thead>
<tr>
<th>argument</th>
<th>coordinate system</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘figure points’</td>
<td>points from the lower left corner of the figure</td>
</tr>
<tr>
<td>‘figure pixels’</td>
<td>pixels from the lower left corner of the figure</td>
</tr>
<tr>
<td>‘figure fraction’</td>
<td>0,0 is lower left of figure and 1,1 is upper right</td>
</tr>
<tr>
<td>‘axes points’</td>
<td>points from lower left corner of axes</td>
</tr>
<tr>
<td>‘axes pixels’</td>
<td>pixels from lower left corner of axes</td>
</tr>
<tr>
<td>‘axes fraction’</td>
<td>0,0 is lower left of axes and 1,1 is upper right</td>
</tr>
<tr>
<td>‘data’</td>
<td>use the axes data coordinate system</td>
</tr>
</tbody>
</table>

For example to place the text coordinates in fractional axes coordinates, one could do:

\begin{verbatim}
ax.annotate('local max', xy=(3, 1), xycoords='data',
            xytext=(0.8, 0.95), textcoords='axes fraction',
            arrowprops=dict(facecolor='black', shrink=0.05),
            horizontalalignment='right', verticalalignment='top',
        )
\end{verbatim}
For physical coordinate systems (points or pixels) the origin is the bottom-left of the figure or axes.

Optionally, you can enable drawing of an arrow from the text to the annotated point by giving a dictionary of arrow properties in the optional keyword argument `arrowprops`.

<table>
<thead>
<tr>
<th><code>arrowprops</code> key</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>width</td>
<td>the width of the arrow in points</td>
</tr>
<tr>
<td>frac</td>
<td>the fraction of the arrow length occupied by the head</td>
</tr>
<tr>
<td>headwidth</td>
<td>the width of the base of the arrow head in points</td>
</tr>
<tr>
<td>shrink</td>
<td>move the tip and base some percent away from the annotated point and text</td>
</tr>
<tr>
<td>**kwargs</td>
<td>any key for <code>matplotlib.patches.Polygon</code>, e.g., <code>facecolor</code></td>
</tr>
</tbody>
</table>

In the example below, the `xy` point is in native coordinates (`xycoords` defaults to `‘data’`). For a polar axes, this is in (theta, radius) space. The text in this example is placed in the fractional figure coordinate system. `matplotlib.text.Text` keyword args like `horizontalalignment`, `verticalalignment` and `fontsize` are passed from `annotate` to the `Text` instance.

```python
import numpy as np
import matplotlib.pyplot as plt

fig = plt.figure()
ax = fig.add_subplot(111, polar=True)
r = np.arange(0,1,0.001)
theta = 2*2*np.pi*r
line, = ax.plot(theta, r, color='#ee8d18', lw=3)

ind = 800
thisr, thistheta = r[ind], theta[ind]
ax.plot([thistheta], [thisr], 'o')
ax.annotate('a polar annotation',
            xy=(thistheta, thisr),  # theta, radius
            xytext=(0.05, 0.05),   # fraction, fraction
            textcoords='figure fraction',
            arrowprops=dict(facecolor='black', shrink=0.05),
            horizontalalignment='left',
            verticalalignment='bottom',
)
plt.show()
```
For more on all the wild and wonderful things you can do with annotations, including fancy arrows, see *Advanced Annotation* and *pylab_examples example code: annotation_demo.py*.

Do not proceed unless you have already read *Basic annotation*, `text()` and `annotate()`!

### 4.5.2 Advanced Annotation

**Annotating with Text with Box**

Let’s start with a simple example.
The `text()` function in the pyplot module (or text method of the Axes class) takes bbox keyword argument, and when given, a box around the text is drawn.

```python
bbox_props = dict(boxstyle="rarrow", pad=0.3, fc="cyan", ec="b", lw=2)
t = ax.text(0, 0, "Direction", ha="center", va="center", rotation=45,
            size=15,
            bbox=bbox_props)
```

The patch object associated with the text can be accessed by:

```python
bb = t.get_bbox_patch()
```

The return value is an instance of FancyBboxPatch and the patch properties like facecolor, edgewidth, etc. can be accessed and modified as usual. To change the shape of the box, use the `set_boxstyle` method.

```python
bb.set_boxstyle("rarrow", pad=0.6)
```

The arguments are the name of the box style with its attributes as keyword arguments. Currently, following box styles are implemented.
<table>
<thead>
<tr>
<th>Class</th>
<th>Name</th>
<th>Attrs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Circle</td>
<td>circle</td>
<td>pad=0.3</td>
</tr>
<tr>
<td>DArrow</td>
<td>darrow</td>
<td>pad=0.3</td>
</tr>
<tr>
<td>LArrow</td>
<td>larrow</td>
<td>pad=0.3</td>
</tr>
<tr>
<td>RArrow</td>
<td>rarrow</td>
<td>pad=0.3</td>
</tr>
<tr>
<td>Round</td>
<td>round</td>
<td>pad=0.3, rounding_size=None</td>
</tr>
<tr>
<td>Round4</td>
<td>round4</td>
<td>pad=0.3, rounding_size=None</td>
</tr>
<tr>
<td>Roundtooth</td>
<td>roundtooth</td>
<td>pad=0.3, tooth_size=None</td>
</tr>
<tr>
<td>Sawtooth</td>
<td>sawtooth</td>
<td>pad=0.3, tooth_size=None</td>
</tr>
<tr>
<td>Square</td>
<td>square</td>
<td>pad=0.3</td>
</tr>
</tbody>
</table>
Note that the attribute arguments can be specified within the style name with separating comma (this form can be used as “boxstyle” value of bbox argument when initializing the text instance)

```
bb.set_boxstyle("rarrow,pad=0.6")
```
Annotating with Arrow

The `annotate()` function in the pyplot module (or annotate method of the Axes class) is used to draw an arrow connecting two points on the plot.

```python
ax.annotate("Annotation",
    xy=(x1, y1), xycoords='data',
    xytext=(x2, y2), textcoords='offset points',
)
```

This annotates a point at `xy` in the given coordinate (`xycoords`) with the text at `xytext` given in `textcoords`. Often, the annotated point is specified in the `data` coordinate and the annotating text in `offset points`. See `annotate()` for available coordinate systems.

An arrow connecting two points (`xy` & `xytext`) can be optionally drawn by specifying the `arrowprops` argument. To draw only an arrow, use empty string as the first argument.

```python
ax.annotate("",
    xy=(0.2, 0.2), xycoords='data',
    xytext=(0.8, 0.8), textcoords='data',
    arrowprops=dict(arrowstyle="->",
                    connectionstyle="arc3"),
)
```

The arrow drawing takes a few steps.

1. A connecting path between two points are created. This is controlled by `connectionstyle` key value.
2. If patch object is given (`patchA` & `patchB`), the path is clipped to avoid the patch.
3. The path is further shrunk by given amount of pixels (`shrinkA` & `shrinkB`)
4. The path is transmuted to arrow patch, which is controlled by the `arrowstyle` key value.
The creation of the connecting path between two points is controlled by `connectionstyle` key and the following styles are available:

<table>
<thead>
<tr>
<th>Name</th>
<th>Attrs</th>
</tr>
</thead>
<tbody>
<tr>
<td>angle</td>
<td>angleA=90,angleB=0,rad=0.0</td>
</tr>
<tr>
<td>angle3</td>
<td>angleA=90,angleB=0</td>
</tr>
<tr>
<td>arc</td>
<td>angleA=0,angleB=0,armA=None,armB=None,rad=0.0</td>
</tr>
<tr>
<td>arc3</td>
<td>rad=0.0</td>
</tr>
<tr>
<td>bar</td>
<td>armA=0.0,armB=0.0,fraction=0.3,angle=None</td>
</tr>
</tbody>
</table>

Note that “3” in `angle3` and `arc3` is meant to indicate that the resulting path is a quadratic spline segment (three control points). As will be discussed below, some arrow style options can only be used when the connecting path is a quadratic spline.

The behavior of each connection style is (limitedly) demonstrated in the example below. (Warning: The behavior of the bar style is currently not well defined, it may be changed in the future).
The connecting path (after clipping and shrinking) is then mutated to an arrow patch, according to the given
arrowstyle.

<table>
<thead>
<tr>
<th>Name</th>
<th>Attrs</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>None</td>
</tr>
<tr>
<td>-&gt;</td>
<td>head_length=0.4,head_width=0.2</td>
</tr>
<tr>
<td>-[</td>
<td>widthB=1.0,lengthB=0.2,angleB=None</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>&gt;</td>
</tr>
<tr>
<td>&lt;-</td>
<td>head_length=0.4,head_width=0.2</td>
</tr>
<tr>
<td>&lt;</td>
<td></td>
</tr>
<tr>
<td>&lt;</td>
<td>-&gt;</td>
</tr>
<tr>
<td>fancy</td>
<td>head_length=0.4,head_width=0.4,tail_width=0.4</td>
</tr>
<tr>
<td>simple</td>
<td>head_length=0.5,head_width=0.5,tail_width=0.2</td>
</tr>
<tr>
<td>wedge</td>
<td>tail_width=0.3,shrink_factor=0.5</td>
</tr>
</tbody>
</table>
Some arrow styles only work with connection styles that generate a quadratic-spline segment. They are fancy, simple, and wedge. For these arrow styles, you must use the “angle3” or “arc3” connection style.

If the annotation string is given, the patchA is set to the bbox patch of the text by default.
As in the text command, a box around the text can be drawn using the bbox argument.

By default, the starting point is set to the center of the text extent. This can be adjusted with relpos key value. The values are normalized to the extent of the text. For example, (0,0) means lower-left corner and (1,1) means top-right.

**Placing Artist at the anchored location of the Axes**

There are classes of artists that can be placed at an anchored location in the Axes. A common example is the legend. This type of artist can be created by using the OffsetBox class. A few predefined classes are available in mpl_toolkits.axes_grid.anchored_artists.

```python
from mpl_toolkits.axes_grid.anchored_artists import AnchoredText
at = AnchoredText("Figure 1a",
```

The `loc` keyword has the same meaning as in the `legend` command.

A simple application is when the size of the artist (or collection of artists) is known in pixel size during the time of creation. For example, if you want to draw a circle with fixed size of 20 pixel x 20 pixel (radius = 10 pixel), you can utilize `AnchoredDrawingArea`. The instance is created with a size of the drawing area (in pixels), and arbitrary artists can be added to the drawing area. Note that the extents of the artists that are added to the drawing area are not related to the placement of the drawing area itself. Only the initial size matters.

```python
from mpl_toolkits.axes_grid.anchored_artists import AnchoredDrawingArea

da = AnchoredDrawingArea(20, 20, 0, 0,
                         loc=1, pad=0., frameon=False)
p1 = Circle((10, 10), 10)
da.drawing_area.add_artist(p1)
p2 = Circle((30, 10), 5, fc="r")
da.drawing_area.add_artist(p2)
```

The artists that are added to the drawing area should not have a transform set (it will be overridden) and the dimensions of those artists are interpreted as a pixel coordinate, i.e., the radius of the circles in above example are 10 pixels and 5 pixels, respectively.
Sometimes, you want your artists to scale with the data coordinate (or coordinates other than canvas pixels). You can use `AnchoredAuxTransformBox` class. This is similar to `AnchoredDrawingArea` except that the extent of the artist is determined during the drawing time respecting the specified transform.

```python
from mpl_toolkits.axes_grid.anchored_artists import AnchoredAuxTransformBox

box = AnchoredAuxTransformBox(ax.transData, loc=2)
el = Ellipse((0, 0), width=0.1, height=0.4, angle=30)  # in data coordinates!
box.drawing_area.add_artist(el)
```

The ellipse in the above example will have width and height corresponding to 0.1 and 0.4 in data coordinateing and will be automatically scaled when the view limits of the axes change.

As in the legend, the `bbox_to_anchor` argument can be set. Using the HPacker and VPacker, you can have an arrangement(?) of artist as in the legend (as a matter of fact, this is how the legend is created).

4.5. Annotation
Note that unlike the legend, the `bbox_transform` is set to `IdentityTransform` by default.

**Using Complex Coordinates with Annotations**

The Annotation in matplotlib supports several types of coordinates as described in *Basic annotation*. For an advanced user who wants more control, it supports a few other options.

1. *Transform* instance. For example,

   ```python
   ax.annotate("Test", xy=(0.5, 0.5), xycoords=ax.transAxes)
   ```

   is identical to

   ```python
   ax.annotate("Test", xy=(0.5, 0.5), xycoords="axes fraction")
   ```

   With this, you can annotate a point in other axes.

   ```python
   ax1, ax2 = subplot(121), subplot(122)
   ax2.annotate("Test", xy=(0.5, 0.5), xycoords=ax1.transData,
                   xytext=(0.5, 0.5), textcoords=ax2.transData,
                   arrowprops=dict(arrowstyle="->"))
   ```

2. *Artist* instance. The `xy` value (or `xytext`) is interpreted as a fractional coordinate of the `bbox` (return value of `get_window_extent`) of the artist.

   ```python
   an1 = ax.annotate("Test 1", xy=(0.5, 0.5), xycoords="data",
                     va="center", ha="center",
                     bbox=dict(boxstyle="round", fc="w"))
   an2 = ax.annotate("Test 2", xy=(1, 0.5), xycoords=an1, # (1,0.5) of the an1's bbox
                     xytext=(30,0), textcoords="offset points",
                     va="center", ha="left",
                     bbox=dict(boxstyle="round", fc="w"),
                     arrowprops=dict(arrowstyle="->"))
   ```
3. A callable object that returns an instance of either `BboxBase` or `Transform`. If a transform is returned, it is the same as 1 and if a box is returned, it is the same as 2. The callable object should take a single argument of the renderer instance. For example, the following two commands give identical results:

```python
an2 = ax.annotate("Test 2", xy=(1, 0.5), xycoords=an1,
                   xytext=(30,0), textcoords="offset points")
an2 = ax.annotate("Test 2", xy=(1, 0.5), xycoords=an1.get_window_extent,
                   xytext=(30,0), textcoords="offset points")
```

4. A tuple of two coordinate specifications. The first item is for the x-coordinate and the second is for the y-coordinate. For example,

```python
annotate("Test", xy=(0.5, 1), xycoords=("data", "axes fraction"))
```

0.5 is in data coordinates, and 1 is in normalized axes coordinates. You may use an artist or transform as with a tuple. For example,

```python
import matplotlib.pyplot as plt
plt.figure(figsize=(3,2))
a1=plt.axes([0.1, 0.1, 0.8, 0.7])
an1 = ax.annotate("Test 1", xy=(0.5, 0.5), xycoords="data",
                   va="center", ha="center",
                   bbox=dict(boxstyle="round", fc="w"))

an2 = ax.annotate("Test 2", xy=(0.5, 1.), xycoords=an1,
                   xytext=(0.5,1.1), textcoords=(an1, "axes fraction"),
                   va="bottom", ha="center",
                   bbox=dict(boxstyle="round", fc="w"),
                   arrowprops=dict(arrowstyle="->"))
plt.show()
```
5. Sometimes, you want your annotation with some “offset points”, not from the annotated point but from some other point. OffsetFrom is a helper class for such cases.

```python
import matplotlib.pyplot as plt

plt.figure(figsize=(3, 2))
ax = plt.axes([0.1, 0.1, 0.8, 0.7])
an1 = ax.annotate("Test 1", xy=(0.5, 0.5), xycoords="data",
va="center", ha="center",
bbox=dict(boxstyle="round", fc="w"))

from matplotlib.text import OffsetFrom
offset_from = OffsetFrom(an1, (0.5, 0))
an2 = ax.annotate("Test 2", xy=(0.1, 0.1), xycoords="data",
xytext=(0, -10), textcoords=offset_from,
# xytext is offset points from "xy=(0.5, 0), xycoords=an1"
va="top", ha="center",
bbox=dict(boxstyle="round", fc="w"),
arowprops=dict(arrowstyle="->"))

plt.show()
```

You may take a look at this example `pylab_examples example code: annotation_demo3.py`.

**Using ConnectorPatch**

The ConnectorPatch is like an annotation without text. While the annotate function is recommended in most situations, the ConnectorPatch is useful when you want to connect points in different axes.
from matplotlib.patches import ConnectionPatch
xy = (0.2, 0.2)
con = ConnectionPatch(xyA=xy, xyB=xy, coordsA="data", coordsB="data",
axesA=ax1, axesB=ax2)
ax2.add_artist(con)

The above code connects point xy in the data coordinates of ax1 to point xy in the data coordinates of ax2. Here is a simple example.

While the ConnectorPatch instance can be added to any axes, you may want to add it to the axes that is latest in drawing order to prevent overlap by other axes.

Advanced Topics

Zoom effect between Axes

mpl_toolkits.axes_grid.inset_locator defines some patch classes useful for interconnecting two axes. Understanding the code requires some knowledge of how mpl’s transform works. But, utilizing it will be straightforward.
Define Custom BoxStyle

You can use a custom box style. The value for the boxstyle can be a callable object in the following forms:

```python
def __call__(self, x0, y0, width, height, mutation_size, aspect_ratio=1.):
    """
    Given the location and size of the box, return the path of
    the box around it.
    
    - *x0*, *y0*, *width*, *height* : location and size of the box
    - *mutation_size* : a reference scale for the mutation.
    - *aspect_ratio* : aspect-ratio for the mutation.
    """
    path = ...
    return path
```

Here is a complete example.
However, it is recommended that you derive from the matplotlib.patches.BoxStyle._Base as demonstrated below.

```python
from matplotlib.path import Path
from matplotlib.patches import BoxStyle
import matplotlib.pyplot as plt

# we may derive from matplotlib.patches.BoxStyle._Base class.
# You need to override transmute method in this case.

class MyStyle(BoxStyle._Base):
    
    """
    A simple box.
    """

    def __init__(self, pad=0.3):
        """
        The arguments need to be floating numbers and need to have
default values.
        """
        *pad*
        amount of padding

        self.pad = pad
        super(MyStyle, self).__init__()

    def transmute(self, x0, y0, width, height, mutation_size):
        """
        Given the location and size of the box, return the path of
the box around it.

        - *x0*, *y0*, *width*, *height* : location and size of the box
        - *mutation_size* : a reference scale for the mutation.
```

4.5. Annotation
Often, the "mutation_size" is the font size of the text. You don't need to worry about the rotation as it is automatically taken care of.

```
# padding
pad = mutation_size * self.pad

# width and height with padding added.
width, height = width + 2.*pad, 
                height + 2.*pad,

# boundary of the padded box
x0, y0 = x0-pad, y0-pad,
x1, y1 = x0+width, y0 + height

cp = [(x0, y0),
      (x1, y0), (x1, y1), (x0, y1),
      (x0-pad, (y0+y1)/2.), (x0, y0),
      (x0, y0)]

com = [Path.MOVETO, 
       Path.LINETO, Path.LINETO, Path.LINETO, 
       Path.LINETO, Path.LINETO, 
       Path.CLOSEPOLY]

path = Path(cp, com)

return path
```

# register the custom style
BoxStyle._style_list["angled"] = MyStyle

plt.figure(1, figsize=(3,3))
ax = plt.subplot(111)
ax.text(0.5, 0.5, "Test", size=30, va="center", ha="center", rotation=30, 
        bbox=dict(boxstyle="angled,pad=0.5", alpha=0.2))

del BoxStyle._style_list["angled"]

plt.show()
Similarly, you can define a custom ConnectionStyle and a custom ArrowStyle. See the source code of lib/matplotlib/patches.py and check how each style class is defined.

## 4.6 Writing mathematical expressions

You can use a subset TeX markup in any matplotlib text string by placing it inside a pair of dollar signs ($). Note that you do not need to have TeX installed, since matplotlib ships its own TeX expression parser, layout engine and fonts. The layout engine is a fairly direct adaptation of the layout algorithms in Donald Knuth’s TeX, so the quality is quite good (matplotlib also provides a `usetex` option for those who do want to call out to TeX to generate their text (see *Text rendering With LaTeX*).

Any text element can use math text. You should use raw strings (precede the quotes with an 'r'), and surround the math text with dollar signs ($), as in TeX. Regular text and mathtext can be interleaved within the same string. Mathtext can use DejaVu Sans (default), DejaVu Serif, the Computer Modern fonts (from (La)TeX), STIX fonts (with are designed to blend well with Times), or a Unicode font that you provide. The mathtext font can be selected with the customization variable `mathtext.fontset` (see *Customizing matplotlib*).

**Note:** On “narrow” builds of Python, if you use the STIX fonts you should also set `ps.fonttype` and `pdf.fonttype` to 3 (the default), not 42. Otherwise some characters will not be visible.

Here is a simple example:

```python
# plain text
plt.title('alpha > beta')
```

produces “alpha > beta”.

Whereas this:
# math text
plt.title(r'$\alpha > \beta$')

produces “$\alpha > \beta$”.

**Note:** Mathtext should be placed between a pair of dollar signs ($). To make it easy to display monetary values, e.g., “$100.00”, if a single dollar sign is present in the entire string, it will be displayed verbatim as a dollar sign. This is a small change from regular TeX, where the dollar sign in non-math text would have to be escaped (“\$”).

**Note:** While the syntax inside the pair of dollar signs ($) aims to be TeX-like, the text outside does not. In particular, characters such as:

```
# $ % & ~ _ ^ \ { } \( \) \\
```

have special meaning outside of math mode in TeX. Therefore, these characters will behave differently depending on the rcParam text.usetex flag. See the [usetex tutorial](#) for more information.

### 4.6.1 Subscripts and superscripts

To make subscripts and superscripts, use the '_-' and '^' symbols:

```
r'$\alpha_i > \beta_i$'
```

\[\alpha_i > \beta_i\] (4.1)

Some symbols automatically put their sub/superscripts under and over the operator. For example, to write the sum of $x_i$ from 0 to $\infty$, you could do:

```
r'$\sum_{i=0}^{\infty} x_i$'
```

\[
\sum_{i=0}^{\infty} x_i\] (4.2)

### 4.6.2 Fractions, binomials and stacked numbers

Fractions, binomials and stacked numbers can be created with the \frac{}, \binom{} and \stackrel{} commands, respectively:

```
r'$\frac{3}{4} \binom{3}{4} \stackrel{3}{4}$'
```

produces

\[
\frac{3}{4} \binom{3}{4} \stackrel{3}{4}\] (4.3)

Fractions can be arbitrarily nested:
$\frac{5 - \frac{1}{x}}{4}$

produces

$$\frac{5 - \frac{1}{x}}{4}$$ (4.4)

Note that special care needs to be taken to place parentheses and brackets around fractions. Doing things the obvious way produces brackets that are too small:

$$(\frac{5 - \frac{1}{x}}{4})$$

$$\left(\frac{5 - \frac{1}{x}}{4}\right)$$ (4.5)

The solution is to precede the bracket with \left and \right to inform the parser that those brackets encompass the entire object:

$$\left(\frac{5 - \frac{1}{x}}{4}\right)$$

4.6.3 Radicals

Radicals can be produced with the $\sqrt{}$ command. For example:

$$\sqrt{2}$$ (4.7)

Any base can (optionally) be provided inside square brackets. Note that the base must be a simple expression, and can not contain layout commands such as fractions or sub/superscripts:

$$\sqrt[3]{x}$$ (4.8)

4.6.4 Fonts

The default font is italics for mathematical symbols.

Note: This default can be changed using the mathtext.default rcParam. This is useful, for example, to use the same font as regular non-math text for math text, by setting it to regular.

To change fonts, e.g., to write “sin” in a Roman font, enclose the text in a font command:

$$s(t) = \mathcal{A}\mathrm{sin}(2 \omega t)$$

4.6. Writing mathematical expressions
More conveniently, many commonly used function names that are typeset in a Roman font have shortcuts. So the expression above could be written as follows:

```
s(t) = \mathcal{A} \sin(2 \omega t)
```

(4.10)

Here “s” and “t” are variable in italics font (default), “sin” is in Roman font, and the amplitude “A” is in calligraphy font. Note in the example above the calligraphy A is squished into the sin. You can use a spacing command to add a little whitespace between them:

```
s(t) = \mathcal{A} \sin(2 \omega t)
```

(4.11)

The choices available with all fonts are:

<table>
<thead>
<tr>
<th>Command</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textsc{Roman}</td>
<td>Roman</td>
</tr>
<tr>
<td>\textit{Italic}</td>
<td>Italic</td>
</tr>
<tr>
<td>\texttt{Typewriter}</td>
<td>Typewriter</td>
</tr>
<tr>
<td>\textcal{CALLIGRAPHY}</td>
<td>CALLIGRAPHY</td>
</tr>
</tbody>
</table>

When using the STIX fonts, you also have the choice of:

<table>
<thead>
<tr>
<th>Command</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textbb{blackboard}</td>
<td>$\mathbb{blackboard}$</td>
</tr>
<tr>
<td>\textsc{\textbb{blackboard}}</td>
<td>$\mathbb{blackboard}$</td>
</tr>
<tr>
<td>\textfrak{Fraktur}</td>
<td>Fraktur</td>
</tr>
<tr>
<td>\textsf{sansserif}</td>
<td>sansserif</td>
</tr>
<tr>
<td>\textsc{\textsf{sansserif}}</td>
<td>sansserif</td>
</tr>
</tbody>
</table>

There are also three global “font sets” to choose from, which are selected using the mathtext.fontset parameter in `matplotlibrc`.

**cm**: Computer Modern (TeX)

\[
\mathcal{R} \prod_{i=\alpha_{i+1}}^{\infty} a_i \sin(2\pi f x_i)
\]

**stix**: STIX (designed to blend well with Times)

\[
\mathcal{R} \prod_{i=\alpha_{i+1}}^{\infty} a_i \sin(2\pi f x_i)
\]

**stixsans**: STIX sans-serif
Additionally, you can use `\textdefault{...}` or its alias `\textregular{...}` to use the font used for regular text outside of math. There are a number of limitations to this approach, most notably that far fewer symbols will be available, but it can be useful to make math expressions blend well with other text in the plot.

### Custom fonts

mathtext also provides a way to use custom fonts for math. This method is fairly tricky to use, and should be considered an experimental feature for patient users only. By setting the rcParam `mathtext.fontset` to `custom`, you can then set the following parameters, which control which font file to use for a particular set of math characters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Corresponds to</th>
</tr>
</thead>
<tbody>
<tr>
<td>mathtext.it</td>
<td><code>\textit{}</code> or default italic</td>
</tr>
<tr>
<td>mathtext.rm</td>
<td><code>\textsc{}</code> Roman (upright)</td>
</tr>
<tr>
<td>mathtext.tt</td>
<td><code>\texttt{}</code> Typewriter (monospace)</td>
</tr>
<tr>
<td>mathtext.bf</td>
<td><code>\textbf{}</code> bold italic</td>
</tr>
<tr>
<td>mathtext.cal</td>
<td><code>\textcal{}</code> calligraphic</td>
</tr>
<tr>
<td>mathtext.sf</td>
<td><code>\textsf{}</code> sans-serif</td>
</tr>
</tbody>
</table>

Each parameter should be set to a fontconfig font descriptor (as defined in the yet-to-be-written font chapter).

The fonts used should have a Unicode mapping in order to find any non-Latin characters, such as Greek. If you want to use a math symbol that is not contained in your custom fonts, you can set the rcParam `mathtext.fallback_to_cm` to True which will cause the mathtext system to use characters from the default Computer Modern fonts whenever a particular character can not be found in the custom font.

Note that the math glyphs specified in Unicode have evolved over time, and many fonts may not have glyphs in the correct place for mathtext.

#### 4.6.5 Accents

An accent command may precede any symbol to add an accent above it. There are long and short forms for some of them.
In addition, there are two special accents that automatically adjust to the width of the symbols below:

<table>
<thead>
<tr>
<th>Command</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>\widehat{xyz}</td>
<td>̂xyz</td>
</tr>
<tr>
<td>\widetilde{xyz}</td>
<td>̃xyz</td>
</tr>
</tbody>
</table>

Care should be taken when putting accents on lower-case i’s and j’s. Note that in the following \imath is used to avoid the extra dot over the i:

```
r"$\hat i \backslash \hat \imath$"
```

\[ \hat i \, \hat \imath \] (4.12)

### 4.6.6 Symbols

You can also use a large number of the TeX symbols, as in \infty, \leftarrow, \sum, \int.

#### Lower-case Greek

<table>
<thead>
<tr>
<th>α , \alpha</th>
<th>β , \beta</th>
<th>χ , \chi</th>
<th>δ , \delta</th>
<th>F , \digamma</th>
</tr>
</thead>
<tbody>
<tr>
<td>ε , \epsilon</td>
<td>η , \eta</td>
<td>γ , \gamma</td>
<td>i , \iota</td>
<td>κ , \kappa</td>
</tr>
<tr>
<td>λ , \lambda</td>
<td>μ , \mu</td>
<td>ν , \nu</td>
<td>ω , \omega</td>
<td>φ , \phi</td>
</tr>
<tr>
<td>π , \pi</td>
<td>ψ , \psi</td>
<td>ρ , \rho</td>
<td>σ , \sigma</td>
<td>τ , \tau</td>
</tr>
<tr>
<td>θ , \theta</td>
<td>υ , \upsilon</td>
<td>ε , \varepsilon</td>
<td>x , \varkappa</td>
<td>ϕ , \varphi</td>
</tr>
<tr>
<td>σ , \sigma</td>
<td>ω , \varrho</td>
<td>ς , \varsigma</td>
<td>θ , \vartheta</td>
<td>ξ , \xi</td>
</tr>
<tr>
<td>ζ , \zeta</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Upper-case Greek

<table>
<thead>
<tr>
<th>Α , \Delta</th>
<th>Γ , \Gamma</th>
<th>Λ , \Lambda</th>
<th>Ω , \Omega</th>
<th>Φ , \Phi</th>
<th>Π , \Pi</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ψ , \Psi</td>
<td>Σ , \Sigma</td>
<td>Θ , \Theta</td>
<td>Τ , \Upsilon</td>
<td>Ξ , \Xi</td>
<td>Ω , \mho</td>
</tr>
<tr>
<td>Ν , \nabla</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Hebrew

| א \, \aleph | ב \, \beth | ד \, \daleth | ג \, \gimel |

#### Delimiters
Big symbols

\begin{tabular}{|c|c|c|c|c|c|}
\hline
\bigcap & \bigcup & \bigodot & \bigoplus & \bigotimes & \biguplus \\
\hline
\Pr & \arccos & \arcsin & \arctan & \arg & \cos \\
\cot & \csc & \cosh & \cot & \deg & \det \\
\dim & \exp & \gcd & \hom & \hom & \int \\
\inf & \ker & \lg & \lim & \ln & \log \\
\lim & \lim & \lim & \lim & \ln & \log \\
\max & \min & \sec & \sec & \sin & \sin \\
\sinh & \sup & \tan & \tan & \tanh & \\
\hline
\end{tabular}

Standard function names

\begin{tabular}{|c|c|c|c|c|c|}
\hline
Pr & \Pr & \arccos & \arcsin & \arctan & \arg \\
\cos & \cosh & \cot & \deg & \det & \hom \\
\csc & \csc & \cosh & \cot & \deg & \det \\
\exp & \exp & \gcd & \hom & \hom & \\
\ker & \ker & \lg & \lim & \ln & \\
\lim & \lim & \lim & \lim & \ln & \log \\
\min & \min & \sec & \sec & \sin & \sin \\
\sup & \tan & \tan & \tanh & \\
\hline
\end{tabular}

Binary operation and relation symbols

\begin{tabular}{|c|c|c|c|}
\hline
\Bumpeq & \Cap & \cup & \\
\Doteq & \Join & \Subset & \\
\Supset & \Vdash & \vdash & \\
\approx & \approx & \approx & \\
\asymp & \backepsilon & \backsimeq & \\
\between & \bigcirc & \div & \\
\bowtie & \blacktriangleleft & \blacktriangleright & \\
\bigcirc & \circ & \circ & \\
\cong & \cup & \cup & \\
\curlyeqprec & \curlyeqsucc & \curlyvee & \\
\curlywedge & \dag & \div & \\
\divideontimes & \doteq & \doteqdot & \\
\dotplus & \euro & \equiv & \\
\eqcirc & \eqcolon & \eqsim & \\
\eqslantless & \fallingdotseq & \\
\hline
\end{tabular}
### Arrow symbols

| \Downarrow | \Leftarrow | \Rightarrow |
| \Leftrightarrow | \Longleftarrow | \Longrightarrow |
| \Nearrow | \Nwarrow | \Rrightarrow |
| \Swarrow | \Uparrow | \multimap |

### Miscellaneous symbols

| \longleftarrow | \leftrightarrow | \twoheadleftarrow |
| \looparrowleft | \looparrowright | \uparrow |
| \mapsto | \multimap | \twoheadrightarrow |
| \nLeftarrow | \nLeftrightarrow | \nearrow |
| \nLeftarrow | \nLeftrightarrow | \swarrow |
| \nRightarrow | \nLeftarrow | \searrow |
| \nLeftarrow | \nLeftrightarrow | \nearrow |
| \nRightarrow | \nLeftarrow | \swarrow |

#### 4.6. Writing mathematical expressions
If a particular symbol does not have a name (as is true of many of the more obscure symbols in the STIX fonts), Unicode characters can also be used:

```
ur'$\u23ce$
```

### 4.6.7 Example

Here is an example illustrating many of these features in context.

```python
import numpy as np
import matplotlib.pyplot as plt
t = np.arange(0.0, 2.0, 0.01)
s = np.sin(2*np.pi*t)
plt.plot(t, s)
plt.title(r'$\alpha_i > \beta_i$', fontsize=20)
plt.text(1, -0.6, r'$\sum_{i=0}^\infty x_i$', fontsize=20)
plt.text(0.6, 0.6, r'$\mathcal{A}\mathrm{sin}(2 \omega t)$',
          fontsize=20)
plt.xlabel('time (s)')
plt.ylabel('volts (mV)')
plt.show()```
4.7 Text rendering With LaTeX

Matplotlib has the option to use LaTeX to manage all text layout. This option is available with the following backends:

- Agg
- PS
- PDF

The LaTeX option is activated by setting `text.usetex : True` in your rc settings. Text handling with matplotlib's LaTeX support is slower than matplotlib's very capable mathtext, but is more flexible, since different LaTeX packages (font packages, math packages, etc.) can be used. The results can be striking, especially when you take care to use the same fonts in your figures as in the main document.

Matplotlib's LaTeX support requires a working LaTeX installation, dvipng (which may be included with your LaTeX installation), and Ghostscript (GPL Ghostscript 8.60 or later is recommended). The executables for these external dependencies must all be located on your `PATH`.

There are a couple of options to mention, which can be changed using `rc settings`. Here is an example matplotlibrc file:
The first valid font in each family is the one that will be loaded. If the fonts are not specified, the Computer Modern fonts are used by default. All of the other fonts are Adobe fonts. Times and Palatino each have their own accompanying math fonts, while the other Adobe serif fonts make use of the Computer Modern math fonts. See the PSNFSS documentation for more details.

To use LaTeX and select Helvetica as the default font, without editing matplotlibrc use:

```python
from matplotlib import rc
rc('font', **{'family': 'sans-serif', 'sans-serif': ['Helvetica']})
```

## for Palatino and other serif fonts use:
```python
#rc('font', **{'family': 'serif', 'serif': ['Palatino']})
rc('text', usetex=True)
```

Here is the standard example, `tex_demo.py`:

```python
import numpy as np
import matplotlib.pyplot as plt

# Example data
m = np.arange(0.0, 1.0 + 0.01, 0.01)
s = np.cos(4 * np.pi * m) + 2

plt.rc('text', usetex=True)
plt.rc('font', family='serif')
plt.plot(m, s)
plt.xlabel(r'$\textbf{time} (s)$', fontsize=16)
plt.ylabel(r'$\textit{voltage} (mV)$', fontsize=16)
plt.title(r'$\TeX$ is Number $\sum_{n=1}^{\infty} \frac{-e^{i\pi}}{2^n}$!
```

"""
Demo of TeX rendering.

You can use TeX to render all of your matplotlib text if the rc parameter `text.usetex` is set. This works currently on the agg and ps backends, and requires that you have tex and the other dependencies described at http://matplotlib.org/users/usetex.html properly installed on your system. The first time you run a script you will see a lot of output from tex and associated tools. The next time, the run may be silent, as a lot of the information is cached in ~/.tex.cache

"""
import numpy as np
import matplotlib.pyplot as plt

# Example data
m = np.arange(0.0, 1.0 + 0.01, 0.01)
s = np.cos(4 * np.pi * m) + 2

plt.rc('text', usetex=True)
plt.rc('font', family='serif')
plt.plot(m, s)
plt.xlabel(r'$\textbf{time} (s)$')
plt.ylabel(r'$\textit{voltage} (mV)$', fontsize=16)
plt.title(r'$\TeX$ is Number $\sum_{n=1}^{\infty} \frac{-e^{i\pi}}{2^n}$!
```
Note that display math mode ($$ e=mc^2 $$) is not supported, but adding the command \displaystyle, as in tex_demo.py, will produce the same results.

**Note:** Certain characters require special escaping in TeX, such as:

```latex
# $ \% \& \sim \_ \^ \{ \} \( \) \[ \]
```

Therefore, these characters will behave differently depending on the rcParam `text.usetex` flag.

### 4.7.1 usetex with unicode

It is also possible to use unicode strings with the LaTeX text manager, here is an example taken from tex_unicode_demo.py:
This demo is tex_demo.py modified to have unicode. See that file for more information.

```python
# -*- coding: utf-8 -*-

from __future__ import unicode_literals
import numpy as np
import matplotlib
matplotlib.rcParams['text.usetex'] = True
matplotlib.rcParams['text.latex.unicode'] = True
import matplotlib.pyplot as plt

plt.figure(1, figsize=(6, 4))
ax = plt.axes([0.1, 0.1, 0.8, 0.7])
t = np.arange(0.0, 1.0 + 0.01, 0.01)
s = np.cos(2*2*np.pi*t) + 2
plt.plot(t, s)
plt.xlabel(r'\textbf{time (s)}')
plt.ylabel(r'\textit{Velocity (\u00B0/sec)}', fontsize=16)
plt.title(r'\TeX\ is Number $\sum_{n=1}^{\infty} \frac{-e^{i\pi}}{2^n}$!', fontsize=16, color='r')
plt.grid(True)
plt.show()
```

\[ \text{\TeX\ is Number} \sum_{n=1}^{\infty} \frac{-e^{i\pi}}{2^n} \]
4.7.2 Postscript options

In order to produce encapsulated postscript files that can be embedded in a new LaTeX document, the default behavior of matplotlib is to distill the output, which removes some postscript operators used by LaTeX that are illegal in an eps file. This step produces results which may be unacceptable to some users, because the text is coarsely rasterized and converted to bitmaps, which are not scalable like standard postscript, and the text is not searchable. One workaround is to set `ps.distiller.res` to a higher value (perhaps 6000) in your rc settings, which will produce larger files but may look better and scale reasonably. A better workaround, which requires Poppler or Xpdf, can be activated by changing the `ps.usedistiller` rc setting to `xpdf`. This alternative produces postscript without rasterizing text, so it scales properly, can be edited in Adobe Illustrator, and searched text in pdf documents.

4.7.3 Possible hangups

- On Windows, the `PATH` environment variable may need to be modified to include the directories containing the latex, dvipng and ghostscript executables. See `Environment Variables` and `Setting environment variables in windows` for details.
- Using MiKTeX with Computer Modern fonts, if you get odd *Agg and PNG results, go to MiKTeX/Options and update your format files.
- The fonts look terrible on screen. You are probably running Mac OS, and there is some funny business with older versions of dvipng on the mac. Set `text.dvipnghack` : `True` in your matplotlibrc file.
- On Ubuntu and Gentoo, the base texlive install does not ship with the type1cm package. You may need to install some of the extra packages to get all the goodies that come bundled with other latex distributions.
- Some progress has been made so matplotlib uses the dvi files directly for text layout. This allows latex to be used for text layout with the pdf and svg backends, as well as the *Agg and PS backends. In the future, a latex installation may be the only external dependency.

4.7.4 Troubleshooting

- Try deleting your `.matplotlib/tex.cache` directory. If you don’t know where to find `.matplotlib`, see `matplotlib configuration and cache directory locations`.
- Make sure LaTeX, dvipng and ghostscript are each working and on your `PATH`.
- Make sure what you are trying to do is possible in a LaTeX document, that your LaTeX syntax is valid and that you are using raw strings if necessary to avoid unintended escape sequences.
- Most problems reported on the mailing list have been cleared up by upgrading Ghostscript. If possible, please try upgrading to the latest release before reporting problems to the list.
- The `text.latex.preamble` rc setting is not officially supported. This option provides lots of flexibility, and lots of ways to cause problems. Please disable this option before reporting problems to the mailing list.
- If you still need help, please see `Getting help`
4.8 Typesetting With XeLaTeX/LuaLaTeX

Using the \texttt{pgf} backend, matplotlib can export figures as pgf drawing commands that can be processed with \texttt{pdflatex}, \texttt{xelatex} or \texttt{lualatex}. XeLaTeX and LuaLaTeX have full unicode support and can use any font that is installed in the operating system, making use of advanced typographic features of OpenType, AAT and Graphite. Pgf pictures created by \texttt{plt.savefig('figure.pgf')} can be embedded as raw commands in LaTeX documents. Figures can also be directly compiled and saved to PDF with \texttt{plt.savefig('figure.pdf')} by either switching to the backend

\begin{verbatim}
matplotlib.use('pgf')
\end{verbatim}

or registering it for handling pdf output

\begin{verbatim}
from matplotlib.backends.backend_pgf import FigureCanvasPgf
matplotlib.backend_bases.register_backend('pdf', FigureCanvasPgf)
\end{verbatim}

The second method allows you to keep using regular interactive backends and to save xelatex, lualatex or pdflatex compiled PDF files from the graphical user interface.

Matplotlib’s pgf support requires a recent \LaTeX installation that includes the TikZ/PGF packages (such as \TeXLive), preferably with XeLaTeX or LuaLaTeX installed. If either pdftocairo or ghostscript is present on your system, figures can optionally be saved to PNG images as well. The executables for all applications must be located on your \texttt{PATH}.

Rc parameters that control the behavior of the pgf backend:

\begin{verbatim}
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Documentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>pgf.preamble</td>
<td>Lines to be included in the LaTeX preamble</td>
</tr>
<tr>
<td>pgf.rcfonts</td>
<td>Setup fonts from rc params using the fontspec package</td>
</tr>
<tr>
<td>pgf.texsystem</td>
<td>Either “xelatex” (default), “lualatex” or “pdflatex”</td>
</tr>
</tbody>
</table>
\end{verbatim}

\textbf{Note:} \LaTeX defines a set of special characters, such as:

\begin{verbatim}
# $ % & ~ _ ^ \ { }
\end{verbatim}

Generally, these characters must be escaped correctly. For convenience, some characters (_,^,%) are automatically escaped outside of math environments.

4.8.1 Font specification

The fonts used for obtaining the size of text elements or when compiling figures to PDF are usually defined in the matplotlib rc parameters. You can also use the \LaTeX default Computer Modern fonts by clearing the lists for \texttt{font.serif}, \texttt{font.sans-serif} or \texttt{font.monospace}. Please note that the glyph coverage of these fonts is very limited. If you want to keep the Computer Modern font face but require extended unicode support, consider installing the \texttt{Computer Modern Unicode} fonts \texttt{CMU Serif}, \texttt{CMU Sans Serif}, etc.

When saving to \texttt{.pgf}, the font configuration matplotlib used for the layout of the figure is included in the header of the text file.
4.8.2 Custom preamble

Full customization is possible by adding your own commands to the preamble. Use the `pgf.preamble` parameter if you want to configure the math fonts, using `unicode-math` for example, or for loading additional packages. Also, if you want to do the font configuration yourself instead of using the fonts specified in the rc parameters, make sure to disable `pgf.rcfonts`.

```python
# -*- coding: utf-8 -*-

from __future__ import (absolute_import, division, print_function,
                         unicode_literals)

import six
```
import matplotlib as mpl
mpl.use("pgf")
pgf_with_custom_preamble = {
    "font.family": "serif", # use serif/main font for text elements
    "text.usetex": True,    # use inline math for ticks
    "pgf.rcfonts": False,  # don’t setup fonts from rc parameters
    "pgf.preamble": [
        r"\usepackage{units}",   # load additional packages
        r"\usepackage{metahpgo}",
        r"\usepackage{unicode-math}",  # unicode math setup
        r"\setmathfont{xits-math.otf}",
        r"\setmainfont{DejaVu Serif}"  # serif font via preamble
    ]
}
mpl.rcParams.update(pgf_with_custom_preamble)

4.8.3 Choosing the TeX system

The TeX system to be used by matplotlib is chosen by the pgf.texsystem parameter. Possible values are 'xelatex' (default), 'lualatex' and 'pdflatex'. Please note that when selecting pdflatex the fonts and unicode handling must be configured in the preamble.

# -*- coding: utf-8 -*-
import matplotlib as mpl
mpl.use("pgf")
pgf_with_pdflatex = {
    "pgf.texsystem": "pdflatex",
    "pgf.preamble": [
        r"\usepackage[utf8x]{inputenc}",
        r"\usepackage[T1]{fontenc}",
        r"\usepackage{cambright}",
    ]
}
mpl.rcParams.update(pgf_with_pdflatex)

```python
import matplotlib.pyplot as plt
plt.figure(figsize=(4.5, 2.5))
plt.plot(range(5))
plt.text(0.5, 3., "serif", family="serif")
plt.text(0.5, 2., "monospace", family="monospace")
plt.text(2.5, 2., "sans-serif", family="sans-serif")
plt.xlabel(u"\mu is not $\mu$")
plt.tight_layout(.5)
```

4.8.4 Troubleshooting

- Please note that the TeX packages found in some Linux distributions and MiKTeX installations are dramatically outdated. Make sure to update your package catalog and upgrade or install a recent TeX distribution.

- On Windows, the `PATH` environment variable may need to be modified to include the directories containing the `latex`, `dvipng` and `ghostscript` executables. See Environment Variables and Setting environment variables in windows for details.

- A limitation on Windows causes the backend to keep file handles that have been opened by your application open. As a result, it may not be possible to delete the corresponding files until the application closes (see #1324).

- Sometimes the font rendering in figures that are saved to png images is very bad. This happens when the `pdftocairo` tool is not available and `ghostscript` is used for the pdf to png conversion.

- Make sure what you are trying to do is possible in a LaTeX document, that your LaTeX syntax is valid and that you are using raw strings if necessary to avoid unintended escape sequences.

- The `pgf.preamble` rc setting provides lots of flexibility, and lots of ways to cause problems. When experiencing problems, try to minimalize or disable the custom preamble.
• Configuring an unicode-math environment can be a bit tricky. The TeXLive distribution for example provides a set of math fonts which are usually not installed system-wide. XeTeX, unlike LuaLatex, cannot find these fonts by their name, which is why you might have to specify \setmathfont{xits-math.otf} instead of \setmathfont{XITS Math} or alternatively make the fonts available to your OS. See this tex.stackexchange.com question for more details.

• If the font configuration used by matplotlib differs from the font setting in your LaTeX document, the alignment of text elements in imported figures may be off. Check the header of your .pgf file if you are unsure about the fonts matplotlib used for the layout.

• Vector images and hence .pgf files can become bloated if there are a lot of objects in the graph. This can be the case for image processing or very big scatter graphs. In an extreme case this can cause TeX to run out of memory: “TeX capacity exceeded, sorry” You can configure latex to increase the amount of memory available to generate the .pdf image as discussed on tex.stackexchange.com. Another way would be to “rasterize” parts of the graph causing problems using either the rasterized=\text{True} keyword, or .set_rasterized(\text{True}) as per this example.

• If you still need help, please see Getting help
5.1 Specifying Colors

In almost all places in matplotlib where a color can be specified by the user it can be provided as:

- an RGB or RGBA tuple of float values in \([0, 1]\) (e.g., \((0.1, 0.2, 0.5)\) or \((0.1, 0.2, 0.5, 0.3)\))
- a hex RGB or RGBA string (e.g., '\#0F0F0F' or '\#0F0F0F0F')
- a string representation of a float value in \([0, 1]\) inclusive for gray level (e.g., '0.5')
- one of {'b', 'g', 'r', 'c', 'm', 'y', 'k', 'w'}
- a X11/CSS4 color name
- a name from the xkcd color survey prefixed with 'xkcd:' (e.g., 'xkcd:sky blue')
- one of {'C0', 'C1', 'C2', 'C3', 'C4', 'C5', 'C6', 'C7', 'C8', 'C9'}
- one of {'tab:blue', 'tab:orange', 'tab:green', 'tab:red', 'tab:purple', 'tab:brown', 'tab:pink', 'tab:gray', 'tab:olive', 'tab:cyan'} which are the Tableau Colors from the ‘T10’ categorical palette (which is the default color cycle).

All string specifications of color are case-insensitive.

5.1.1 'CN' color selection

Color can be specified by a string matching the regex C[0-9]. This can be passed any place that a color is currently accepted and can be used as a ‘single character color’ in format-string to matplotlib.Axes.plot.

The single digit is the index into the default property cycle (matplotlib.rcParams['axes.prop_cycle']). If the property cycle does not include 'color' then black is returned. The color is evaluated when the artist is created. For example,

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
th = np.linspace(0, 2*np.pi, 128)
```
```python
def demo(sty):
    mpl.style.use(sty)
    fig, ax = plt.subplots(figsize=(3, 3))
    ax.set_title('style: {!r}'.format(sty), color='C0')
    ax.plot(th, np.cos(th), 'C1', label='C1')
    ax.plot(th, np.sin(th), 'C2', label='C2')
    ax.legend()

demo('default')
demo('seaborn')
```

will use the first color for the title and then plot using the second and third colors of each style’s mpl.
rcParams['axes.prop_cycle'].
5.1.2 xkcd v X11/CSS4

The xkcd colors are derived from a user survey conducted by the webcomic xkcd. Details of the survey are available on the xkcd blog.

Out of 148 colors in the CSS color list, there are 95 name collisions between the X11/CSS4 names and the xkcd names, all but 3 of which have different hex values. For example 'blue' maps to '#0000FF' whereas 'xkcd:blue' maps to '#0343DF'. Due to these name collisions all of the xkcd colors have 'xkcd:' prefixed. As noted in the blog post, while it might be interesting to re-define the X11/CSS4 names based on such a survey, we do not do so unilaterally.

The name collisions are shown in the table below; the color names where the hex values agree are shown in bold.
<table>
<thead>
<tr>
<th>X11</th>
<th>xkcd</th>
</tr>
</thead>
<tbody>
<tr>
<td>yellowgreen</td>
<td>aqua</td>
</tr>
<tr>
<td>yellow</td>
<td>aquamarine</td>
</tr>
<tr>
<td>white</td>
<td>azure</td>
</tr>
<tr>
<td>wheat</td>
<td>beige</td>
</tr>
<tr>
<td>violet</td>
<td>black</td>
</tr>
<tr>
<td>turquoise</td>
<td>blue</td>
</tr>
<tr>
<td>tomato</td>
<td>brown</td>
</tr>
<tr>
<td>teal</td>
<td>chartreuse</td>
</tr>
<tr>
<td>tan</td>
<td>chocolate</td>
</tr>
<tr>
<td>silver</td>
<td>coral</td>
</tr>
<tr>
<td>sienna</td>
<td>crimson</td>
</tr>
<tr>
<td>salmon</td>
<td>cyan</td>
</tr>
<tr>
<td>red</td>
<td>darkblue</td>
</tr>
<tr>
<td>purple</td>
<td>darkgreen</td>
</tr>
<tr>
<td>plum</td>
<td>fuchsia</td>
</tr>
<tr>
<td>pink</td>
<td>gold</td>
</tr>
<tr>
<td>orchid</td>
<td>goldenrod</td>
</tr>
<tr>
<td>orange</td>
<td>green</td>
</tr>
<tr>
<td>olive</td>
<td>grey</td>
</tr>
<tr>
<td>navy</td>
<td>indigo</td>
</tr>
<tr>
<td>maroon</td>
<td>ivory</td>
</tr>
<tr>
<td>magenta</td>
<td>khaki</td>
</tr>
<tr>
<td>lime</td>
<td>lavender</td>
</tr>
<tr>
<td>lightgreen</td>
<td>lightblue</td>
</tr>
<tr>
<td>lightblue</td>
<td>lightgreen</td>
</tr>
<tr>
<td>lime</td>
<td>lightgreen</td>
</tr>
<tr>
<td>magenta</td>
<td>lime</td>
</tr>
<tr>
<td>maroon</td>
<td>magenta</td>
</tr>
<tr>
<td>navy</td>
<td>magenta</td>
</tr>
<tr>
<td>olive</td>
<td>magenta</td>
</tr>
</tbody>
</table>
5.2 Choosing Colormaps

5.2.1 Overview

The idea behind choosing a good colormap is to find a good representation in 3D colorspace for your data set. The best colormap for any given data set depends on many things including:

- Whether representing form or metric data ([Ware])
- Your knowledge of the data set (e.g., is there a critical value from which the other values deviate?)
- If there is an intuitive color scheme for the parameter you are plotting
- If there is a standard in the field the audience may be expecting

For many applications, a perceptually uniform colormap is the best choice — one in which equal steps in data are perceived as equal steps in the color space. Researchers have found that the human brain perceives changes in the lightness parameter as changes in the data much better than, for example, changes in hue. Therefore, colormaps which have monotonically increasing lightness through the colormap will be better interpreted by the viewer. A wonderful example of perceptually uniform colormaps is [colorcet].

Color can be represented in 3D space in various ways. One way to represent color is using CIELAB. In CIELAB, color space is represented by lightness, $L^*$; red-green, $a^*$; and yellow-blue, $b^*$. The lightness parameter $L^*$ can then be used to learn more about how the matplotlib colormaps will be perceived by viewers.

An excellent starting resource for learning about human perception of colormaps is from [IBM].

5.2.2 Classes of colormaps

Colormaps are often split into several categories based on their function (see, e.g., [Moreland]):

1. Sequential: change in lightness and often saturation of color incrementally, often using a single hue; should be used for representing information that has ordering.

2. Diverging: change in lightness and possibly saturation of two different colors that meet in the middle at an unsaturated color; should be used when the information being plotted has a critical middle value, such as topography or when the data deviates around zero.

3. Qualitative: often are miscellaneous colors; should be used to represent information which does not have ordering or relationships.

5.2.3 Lightness of matplotlib colormaps

Here we examine the lightness values of the matplotlib colormaps. Note that some documentation on the colormaps is available ([list-colormaps]).
Sequential

For the Sequential plots, the lightness value increases monotonically through the colormaps. This is good. Some of the \( L^* \) values in the colormaps span from 0 to 100 (binary and the other grayscale), and others start around \( L^* = 20 \). Those that have a smaller range of \( L^* \) will accordingly have a smaller perceptual range. Note also that the \( L^* \) function varies amongst the colormaps: some are approximately linear in \( L^* \) and others are more curved.

Sequential2

Many of the \( L^* \) values from the Sequential2 plots are monotonically increasing, but some (autumn, cool, spring, and winter) plateau or even go both up and down in \( L^* \) space. Others (afmhot, copper, gist_heat, and hot) have kinks in the \( L^* \) functions. Data that is being represented in a region of the colormap that is at a plateau or kink will lead to a perception of banding of the data in those values in the colormap (see [mycarta-banding] for an excellent example of this).

Diverging

For the Diverging maps, we want to have monotonically increasing \( L^* \) values up to a maximum, which should be close to \( L^* = 100 \), followed by monotonically decreasing \( L^* \) values. We are looking for approximately equal minimum \( L^* \) values at opposite ends of the colormap. By these measures, BrBG and RdBu are good options. coolwarm is a good option, but it doesn’t span a wide range of \( L^* \) values (see grayscale section below).

Qualitative

Qualitative colormaps are not aimed at being perceptual maps, but looking at the lightness parameter can verify that for us. The \( L^* \) values move all over the place throughout the colormap, and are clearly not monotonically increasing. These would not be good options for use as perceptual colormaps.

Miscellaneous

Some of the miscellaneous colormaps have particular uses for which they have been created. For example, gist_earth, ocean, and terrain all seem to be created for plotting topography (green/brown) and water depths (blue) together. We would expect to see a divergence in these colormaps, then, but multiple kinks may not be ideal, such as in gist_earth and terrain. CMRmap was created to convert well to grayscale, though it does appear to have some small kinks in \( L^* \). cubehelix was created to vary smoothly in both lightness and hue, but appears to have a small hump in the green hue area.

The often-used jet colormap is included in this set of colormaps. We can see that the \( L^* \) values vary widely throughout the colormap, making it a poor choice for representing data for viewers to see perceptually. See an extension on this idea at [mycarta-jet].
Perceptually Uniform Sequential colormaps

5.2. Choosing Colormaps
Sequential colormaps

Lightness $L^*$

Sequential colormaps

Chapter 5. Colors
5.2. Choosing Colormaps

Sequential (2) colormaps
Diverging colormaps
5.2. Choosing Colormaps
5.2.4 Grayscale conversion

It is important to pay attention to conversion to grayscale for color plots, since they may be printed on black and white printers. If not carefully considered, your readers may end up with indecipherable plots because the grayscale changes unpredictably through the colormap.

Conversion to grayscale is done in many different ways [bw]. Some of the better ones use a linear combination of the rgb values of a pixel, but weighted according to how we perceive color intensity. A nonlinear
method of conversion to grayscale is to use the $L^*$ values of the pixels. In general, similar principles apply for this question as they do for presenting one’s information perceptually; that is, if a colormap is chosen that is monotonically increasing in $L^*$ values, it will print in a reasonable manner to grayscale.

With this in mind, we see that the Sequential colormaps have reasonable representations in grayscale. Some of the Sequential2 colormaps have decent enough grayscale representations, though some (autumn, spring, summer, winter) have very little grayscale change. If a colormap like this was used in a plot and then the plot was printed to grayscale, a lot of the information may map to the same gray values. The Diverging colormaps mostly vary from darker gray on the outer edges to white in the middle. Some (PuOr and seismic) have noticeably darker gray on one side than the other and therefore are not very symmetric. coolwarm has little range of gray scale and would print to a more uniform plot, losing a lot of detail. Note that overlaid, labeled contours could help differentiate between one side of the colormap vs. the other since color cannot be used once a plot is printed to grayscale. Many of the Qualitative and Miscellaneous colormaps, such as Accent, hsv, and jet, change from darker to lighter and back to darker gray throughout the colormap. This would make it impossible for a viewer to interpret the information in a plot once it is printed in grayscale.

Perceptually Uniform Sequential colormaps

<table>
<thead>
<tr>
<th>viridis</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>plasma</td>
<td></td>
</tr>
<tr>
<td>inferno</td>
<td></td>
</tr>
<tr>
<td>magma</td>
<td></td>
</tr>
</tbody>
</table>
Sequential colormaps

Greys
Purples
Blues
Greens
Oranges
Reds
YlOrBr
YlOrRd
OrRd
PuRd
RdPu
BuPu
GnBu
PuBu
YlGnBu
PuBuGn
BuGn
YlGn
5.2. Choosing Colormaps
Diverging colormaps

- PiYG
- PRGn
- BrBG
- PuOr
- RdGy
- RdBu
- RdYlBu
- RdYlGn
- Spectral
- coolwarm
- bwr
- seismic
Qualitative colormaps

- Pastel1
- Pastel2
- Paired
- Accent
- Dark2
- Set1
- Set2
- Set3
- tab10
- tab20
- tab20b
- tab20c

5.2. Choosing Colormaps
5.2.5 Color vision deficiencies

There is a lot of information available about color blindness (e.g., [colorblindness]). Additionally, there are tools available to convert images to how they look for different types of color vision deficiencies (e.g., [vischeck]).

The most common form of color vision deficiency involves differentiating between red and green. Thus, avoiding colormaps with both red and green will avoid many problems in general.

5.2.6 References

5.3 Colormap Normalization

Objects that use colormaps by default linearly map the colors in the colormap from data values vmin to vmax. For example:

```python
pcm = ax.pcolormesh(x, y, Z, vmin=-1., vmax=1., cmap='RdBu_r')
```

will map the data in Z linearly from -1 to +1, so Z=0 will give a color at the center of the colormap RdBu_r (white in this case).
Matplotlib does this mapping in two steps, with a normalization from [0,1] occurring first, and then mapping onto the indices in the colormap. Normalizations are classes defined in the `matplotlib.colors()` module. The default, linear normalization is `matplotlib.colors.Normalize()`.

Artists that map data to color pass the arguments `vmin` and `vmax` to construct a `matplotlib.colors.Normalize()` instance, then call it:

```python
In [1]: import matplotlib as mpl
In [2]: norm = mpl.colors.Normalize(vmin=-1., vmax=1.)
In [3]: norm(0.)
Out[3]: 0.5
```

However, there are sometimes cases where it is useful to map data to colormaps in a non-linear fashion.

### 5.3.1 Logarithmic

One of the most common transformations is to plot data by taking its logarithm (to the base-10). This transformation is useful to display changes across disparate scales. Using `colors.LogNorm()` normalizes the data via $\log_{10}$. In the example below, there are two bumps, one much smaller than the other. Using `colors.LogNorm()`, the shape and location of each bump can clearly be seen:

```python
""
Demonstration of using norm to map colormaps onto data in non-linear ways.
""

```import numpy as np
import matplotlib.pyplot as plt
import matplotlib.colors as colors
from matplotlib.mlab import bivariate_normal

""
Lognorm: Instead of pcolor log10(Z1) you can have colorbars that have the exponential labels using a norm.
""

N = 100
X, Y = np.mgrid[-3:3:complex(0, N), -2:2:complex(0, N)]

# A low hump with a spike coming out of the top right. Needs to have # z/colour axis on a log scale so we see both hump and spike. linear # scale only shows the spike.
Z1 = bivariate_normal(X, Y, 0.1, 0.2, 1.0, 1.0) + \
     0.1 * bivariate_normal(X, Y, 1.0, 1.0, 0.0, 0.0)

fig, ax = plt.subplots(2, 1)
pcm = ax[0].pcolor(X, Y, Z1,

   norm=colors.LogNorm(vmin=Z1.min(), vmax=Z1.max()),
   cmap='PuBu_r')
fig.colorbar(pcm, ax=ax[0], extend='max')
```

5.3. Colormap Normalization
5.3.2 Symmetric logarithmic

Similarly, it sometimes happens that there is data that is positive and negative, but we would still like a logarithmic scaling applied to both. In this case, the negative numbers are also scaled logarithmically, and mapped to smaller numbers; e.g., if \( v_{\text{min}} = -v_{\text{max}} \), then they the negative numbers are mapped from 0 to 0.5 and the positive from 0.5 to 1.

Since the logarithm of values close to zero tends toward infinity, a small range around zero needs to be mapped linearly. The parameter \( \text{linthresh} \) allows the user to specify the size of this range (-\( \text{linthresh} \), \( \text{linthresh} \)). The size of this range in the colormap is set by \( \text{linscale} \). When \( \text{linscale} == 1.0 \) (the default), the space used for the positive and negative halves of the linear range will be equal to one decade in the logarithmic range.

```python
import numpy as np
import matplotlib.pyplot as plt

pcm = ax[1].pcolor(X, Y, Z1, cmap='PuBu_r')
fig.colorbar(pcm, ax=ax[1], extend='max')
fig.show()
```

Demonstration of using norm to map colormaps onto data in non-linear ways.

```python
import numpy as np
import matplotlib.pyplot as plt
```
Matplotlib, Release 2.0.2

```python
import matplotlib.colors as colors
from matplotlib.mlab import bivariate_normal

# SymLogNorm: two humps, one negative and one positive, The positive with 5-times the amplitude. Linearly, you cannot see detail in the negative hump. Here we logarithmically scale the positive and negative data separately.

N=100
X, Y = np.mgrid[-3:3:complex(0, N), -2:2:complex(0, N)]
Z1 = (bivariate_normal(X, Y, 1., 1., 1.0, 1.0))**2
    - 0.4 * (bivariate_normal(X, Y, 1.0, 1.0, -1.0, 0.0))**2
Z1 = Z1/0.03

fig, ax = plt.subplots(2, 1)

pcm = ax[0].pcolormesh(X, Y, Z1,
    norm=colors.SymLogNorm(linthresh=0.03, linscale=0.03,
    vmin=-1.0, vmax=1.0),
    cmap='RdBu_r')
fig.colorbar(pcm, ax=ax[0], extend='both')

pcm = ax[1].pcolormesh(X, Y, Z1, cmap='RdBu_r', vmin=-np.max(Z1))
fig.colorbar(pcm, ax=ax[1], extend='both')
fig.show()
```

5.3. Colormap Normalization
5.3.3 Power-law

Sometimes it is useful to remap the colors onto a power-law relationship (i.e. $y = x^\gamma$, where $\gamma$ is the power). For this we use the `colors.PowerNorm()`. It takes as an argument `gamma` (`gamma == 1.0` will just yield the default linear normalization):

**Note:** There should probably be a good reason for plotting the data using this type of transformation. Technical viewers are used to linear and logarithmic axes and data transformations. Power laws are less common, and viewers should explicitly be made aware that they have been used.

```python
Demonstration of using norm to map colormaps onto data in non-linear ways.

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.colors as colors
from matplotlib.mlab import bivariate_normal

N = 100
X, Y = np.mgrid[-3:3:complex(0, N), -2:2:complex(0, N)]
```
PowerNorm: Here a power-law trend in X partially obscures a rectified sine wave in Y. We can remove the power law using a PowerNorm.

```python
X, Y = np.mgrid[0:3:complex(0, N), 0:2:complex(0, N)]
Z1 = (1 + np.sin(Y * 10.)) * X**(2.)

fig, ax = plt.subplots(2, 1)
pcm = ax[0].pcolormesh(X, Y, Z1, norm=colors.PowerNorm(gamma=1./2.), cmap='PuBu_r')
fig.colorbar(pcm, ax=ax[0], extend='max')

pcm = ax[1].pcolormesh(X, Y, Z1, cmap='PuBu_r')
fig.colorbar(pcm, ax=ax[1], extend='max')
fig.show()
```

5.3.4 Discrete bounds

Another normalization that comes with matplotlib is `colors.BoundaryNorm()`. In addition to `vmin` and `vmax`, this takes as arguments boundaries between which data is to be mapped. The colors are then linearly distributed between these “bounds”. For instance:
In [4]: import matplotlib.colors as colors
In [5]: bounds = np.array([-0.25, -0.125, 0, 0.5, 1])
In [6]: norm = colors.BoundaryNorm(boundaries=bounds, ncolors=4)
In [7]: print(norm([-0.2,-0.15,-0.02, 0.3, 0.8, 0.99]))
[0 0 1 2 3 3]

Note unlike the other norms, this norm returns values from 0 to \( ncolors-1 \).

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.colors as colors
from matplotlib.mlab import bivariate_normal

N = 100
X, Y = np.mgrid[-3:3:complex(0, N), -2:2:complex(0, N)]
Z1 = (bivariate_normal(X, Y, 1., 1., 1.0, 1.0))**2 \
   - 0.4 * (bivariate_normal(X, Y, 1.0, 1.0, -1.0, 0.0))**2
Z1 = Z1/0.03

BoundaryNorm: For this one you provide the boundaries for your colors, and the Norm puts the first color in between the first pair, the second color between the second pair, etc.

fig, ax = plt.subplots(3, 1, figsize=(8, 8))
ax = ax.flatten()
# even bounds gives a contour-like effect
bounds = np.linspace(-1, 1, 10)
norm = colors.BoundaryNorm(boundaries=bounds, ncolors=256)
pcm = ax[0].pcolormesh(X, Y, Z1, norm=norm, cmap='RdBu_r')
fig.colorbar(pcm, ax=ax[0], extend='both', orientation='vertical')

# uneven bounds changes the colormapping:
bounds = np.array([-0.25, -0.125, 0, 0.5, 1])
norm = colors.BoundaryNorm(boundaries=bounds, ncolors=256)
pcm = ax[1].pcolormesh(X, Y, Z1, norm=norm, cmap='RdBu_r')
fig.colorbar(pcm, ax=ax[1], extend='both', orientation='vertical')

pcm = ax[2].pcolormesh(X, Y, Z1, cmap='RdBu_r', vmin=-np.max(Z1))
fig.colorbar(pcm, ax=ax[2], extend='both', orientation='vertical')
fig.show()
```
5.3.5 Custom normalization: Two linear ranges

It is possible to define your own normalization. In the following example, we modify colors:SymLogNorm() to use different linear maps for the negative data values and the positive. (Note that this example is simple, and does not validate inputs or account for complex cases such as masked data)

Note: This may appear soon as colors.OffsetNorm().

As above, non-symmetric mapping of data to color is non-standard practice for quantitative data, and should only be used advisedly. A practical example is having an ocean/land colormap where the land and ocean
Demonstration of using norm to map colormaps onto data in non-linear ways.

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.colors as colors
from matplotlib.mlab import bivariate_normal

N = 100

X, Y = np.mgrid[-3:3:complex(0, N), -2:2:complex(0, N)]
Z1 = (bivariate_normal(X, Y, 1., 1., 1.0, 1.0))**2 - 0.4 * (bivariate_normal(X, Y, 1.0, 1.0, -1.0, 0.0))**2
Z1 = Z1/0.03

class MidpointNormalize(colors.Normalize):
    def __init__(self, vmin=None, vmax=None, midpoint=None, clip=False):
        self.midpoint = midpoint
        colors.Normalize.__init__(self, vmin, vmax, clip)

    def __call__(self, value, clip=None):
        # I'm ignoring masked values and all kinds of edge cases to make a
        # simple example...
        x, y = [self.vmin, self.midpoint, self.vmax], [0, 0.5, 1]
        return np.ma.masked_array(np.interp(value, x, y))

fig, ax = plt.subplots(2, 1)
pcm = ax[0].pcolormesh(X, Y, Z1, norm=MidpointNormalize(midpoint=0.), cmap='RdBu_r')
fig.colorbar(pcm, ax=ax[0], extend='both')

pcm = ax[1].pcolormesh(X, Y, Z1, cmap='RdBu_r', vmin=-np.max(Z1))
fig.colorbar(pcm, ax=ax[1], extend='both')
fig.show()
```

Custom Norm: An example with a customized normalization. This one uses the example above, and normalizes the negative data differently from the positive.

# Example of making your own norm. Also see matplotlib.colors.
# From Joe Kington: This one gives two different linear ramps:
5.3. Colormap Normalization
6.1 Using style sheets

The style package adds support for easy-to-switch plotting “styles” with the same parameters as a matplotlibrc file (which is read at startup to configure matplotlib).

There are a number of pre-defined styles provided by matplotlib. For example, there’s a pre-defined style called “ggplot”, which emulates the aesthetics of ggplot (a popular plotting package for R). To use this style, just add:

```python
>>> import matplotlib.pyplot as plt
>>> plt.style.use('ggplot')
```

To list all available styles, use:

```python
>>> print(plt.style.available)
```

6.1.1 Defining your own style

You can create custom styles and use them by calling style.use with the path or URL to the style sheet. Additionally, if you add your `<style-name>.mplstyle` file to mpl_configdir/stylelib, you can reuse your custom style sheet with a call to style.use(<style-name>). By default mpl_configdir should be `~/.config/matplotlib`, but you can check where yours is with matplotlib.get_configdir(); you may need to create this directory. You also can change the directory where matplotlib looks for the stylelib/ folder by setting the MPLCONFIGDIR environment variable, see matplotlib configuration and cache directory locations.

Note that a custom style sheet in mpl_configdir/stylelib will override a style sheet defined by matplotlib if the styles have the same name.

For example, you might want to create mpl_configdir/stylelib/presentation.mplstyle with the following:

```plaintext
axes.titlesize : 24
axes.labelsize : 20
lines.linewidth : 3
lines.markersize : 10
```
Then, when you want to adapt a plot designed for a paper to one that looks good in a presentation, you can just add:

```python
>>> import matplotlib.pyplot as plt
>>> plt.style.use('presentation')
```

### 6.1.2 Composing styles

Style sheets are designed to be composed together. So you can have a style sheet that customizes colors and a separate style sheet that alters element sizes for presentations. These styles can easily be combined by passing a list of styles:

```python
>>> import matplotlib.pyplot as plt
>>> plt.style.use(['dark_background', 'presentation'])
```

Note that styles further to the right will overwrite values that are already defined by styles on the left.

### 6.1.3 Temporary styling

If you only want to use a style for a specific block of code but don’t want to change the global styling, the style package provides a context manager for limiting your changes to a specific scope. To isolate your styling changes, you can write something like the following:

```python
>>> import numpy as np
>>> import matplotlib.pyplot as plt

>>> with plt.style.context(['dark_background']):
    plt.plot(np.sin(np.linspace(0, 2 * np.pi)), 'r-o')

>>> # Some plotting code with the default style
>>> plt.show()
```

### 6.2 matplotlib rcParams

#### 6.2.1 Dynamic rc settings

You can also dynamically change the default rc settings in a python script or interactively from the python shell. All of the rc settings are stored in a dictionary-like variable called `matplotlib.rcParams`, which is global to the matplotlib package. `rcParams` can be modified directly, for example:
import matplotlib as mpl
mpl.rcParams['lines.linewidth'] = 2
mpl.rcParams['lines.color'] = 'r'

Matplotlib also provides a couple of convenience functions for modifying rc settings. The `matplotlib.rc()` command can be used to modify multiple settings in a single group at once, using keyword arguments:

```python
import matplotlib as mpl
mpl.rc('lines', linewidth=2, color='r')
```

The `matplotlib.rcdefaults()` command will restore the standard matplotlib default settings.

There is some degree of validation when setting the values of rcParams, see `matplotlib.rcsetup` for details.

### 6.2.2 The `matplotlibrc` File

Matplotlib uses `matplotlibrc` configuration files to customize all kinds of properties, which we call rc settings or rc parameters. You can control the defaults of almost every property in matplotlib: figure size and dpi, line width, color and style, axes, axis and grid properties, text and font properties and so on. Matplotlib looks for `matplotlibrc` in four locations, in the following order:

1. `matplotlibrc` in the current working directory, usually used for specific customizations that you do not want to apply elsewhere.
2. `$MATPLOTLIBRC/matplotlibrc`.
3. It next looks in a user-specific place, depending on your platform:
   - On Linux and FreeBSD, it looks in `.config/matplotlib/matplotlibrc` (or `$XDG_CONFIG_HOME/matplotlib/matplotlibrc`) if you've customized your environment.
   - On other platforms, it looks in `.matplotlib/matplotlibrc`.

See `matplotlibrc` configuration and cache directory locations.

4. `INSTALL/matplotlib/mpl-data/matplotlibrc`, where `INSTALL` is something like `/usr/lib/python3.5/site-packages` on Linux, and maybe `C:\Python35\Lib\site-packages` on Windows. Every time you install matplotlib, this file will be overwritten, so if you want your customizations to be saved, please move this file to your user-specific matplotlib directory.

To display where the currently active `matplotlibrc` file was loaded from, one can do the following:

```python
>>> import matplotlib
>>> matplotlib.matplotlib_fname()
'/home/foo/.config/matplotlib/matplotlibrc'
```

See below for a sample `matplotlibrc file`.

### 6.2. `matplotlib rcParams`

---
Matplotlib, Release 2.0.2

A sample matplotlibrc file

```plaintext
### MATPLOTLIBRC FORMAT

# This is a sample matplotlib configuration file - you can find a copy
# of it on your system in
# site-packages/matplotlib/mpl-data/matplotlibrc. If you edit it
# there, please note that it will be overwritten in your next install.
# If you want to keep a permanent local copy that will not be
# overwritten, place it in the following location:
# unix/linux:
#   $HOME/.config/matplotlib/matplotlibrc or
#   $XDG_CONFIG_HOME/matplotlib/matplotlibrc (if $XDG_CONFIG_HOME is set)
# other platforms:
#   $HOME/.matplotlib/matplotlibrc
#
# See http://matplotlib.org/users/customizing.html#the-matplotlibrc-file for
# more details on the paths which are checked for the configuration file.
#
# This file is best viewed in a editor which supports python mode
# syntax highlighting. Blank lines, or lines starting with a comment
# symbol, are ignored, as are trailing comments. Other lines must
# have the format
#   key : val # optional comment
#
# Colors: for the color values below, you can either use - a
# matplotlib color string, such as r, k, or b - an rgb tuple, such as
# (1.0, 0.5, 0.0) - a hex string, such as ff00ff - a scalar
# grayscale intensity such as 0.75 - a legal html color name, e.g., red,
# blue, darkslategray

#### CONFIGURATION BEGINS HERE

# The default backend; one of GTK GTKAgg GTKCairo GTK3Agg GTK3Cairo
# MacOSX Qt4Agg Qt5Agg TkAgg WX WXAgg Agg Cairo GDK PS PDF SVG
# Template.
# You can also deploy your own backend outside of matplotlib by
# referring to the module name (which must be in the PYTHONPATH) as
# 'module://my_backend'.
backend : TkAgg

# If you are using the Qt4Agg backend, you can choose here
# to use the PyQt4 bindings or the newer PySide bindings to
# the underlying Qt4 toolkit.
#backend.qt4 : PyQt4  # PyQt4 | PySide

# Note that this can be overridden by the environment variable
# QT_API used by Enthought Tool Suite (ETS); valid values are
# "pyqt" and "pyside". The "pyqt" setting has the side effect of
# forcing the use of Version 2 API for QString and QVariant.

# The port to use for the web server in the WebAgg backend.
webagg.port : 8888
```

176 Chapter 6. Customizing matplotlib
# If webagg.port is unavailable, a number of other random ports will
# be tried until one that is available is found.
# webagg.port_retries : 50

# When True, open the webbrowser to the plot that is shown
# webagg.open_in_browser : True

# When True, the figures rendered in the nbagg backend are created with
# a transparent background.
# nbagg.transparent : False

# if you are running pyplot inside a GUI and your backend choice
# conflicts, we will automatically try to find a compatible one for
# you if backend_fallback is True
# backend_fallback: True

#interactive : False
#toolbar : toolbar2  # None | toolbar2 ("classic" is deprecated)
#timezone : UTC     # a pytz timezone string, e.g., US/Central or Europe/Paris

# Where your matplotlib data lives if you installed to a non-default
# location. This is where the matplotlib fonts, bitmaps, etc reside
#datapath : /home/jdhunter/mpldata

### LINES
# See http://matplotlib.org/api/artist_api.html#module-matplotlib.lines for more
# information on line properties.
#lines.linewidth : 1.5    # line width in points
#lines.linestyle : -     # solid line
#lines.color : C0        # has no affect on plot(); see axes.prop_cycle
#lines.marker : None     # the default marker
#lines.markeredgewidth : 1.0  # the line width around the marker symbol
#lines.markersize : 6     # markersize, in points
#lines.dash_joinstyle : miter  # miter|round|bevel
#lines.dash_capstyle : butt # butt|round|projecting
#lines.solid_joinstyle : miter  # miter|round|bevel
#lines.solid_capstyle : projecting # butt|round|projecting
#lines.antialiased : True  # render lines in antialiased (no jaggies)

# The three standard dash patterns. These are scaled by the linewidth.
#lines.dashed_pattern : 2.8, 1.2
#lines.dashdot_pattern : 4.8, 1.2, 0.8, 1.2
#lines.dotted_pattern : 1.1, 1.1
#lines.scale_dashes : True

#markers.fillstyle: full # full|left|right|bottom|top|none

### PATCHES
# Patches are graphical objects that fill 2D space, like polygons or
# circles. See
# http://matplotlib.org/api/artist_api.html#module-matplotlib.patches

6.2. matplotlib rcParams
# information on patch properties
#patch.linewidth : 1  # edge width in points.
#patch.facecolor : C0
#patch.edgecolor : black  # if forced, or patch is not filled
#patch.force_edgecolor : False  # True to always use edgecolor
#patch.antialiased : True  # render patches in antialiased (no jaggies)

### HATCHES
#hatch.color : k
#hatch.linewidth : 1.0

### Boxplot
#boxplot.notch : False
#boxplot.vertical : True
#boxplot.whiskers : 1.5
#boxplot.bootstrap : None
#boxplot.patchartist : False
#boxplot.showmeans : False
#boxplot.showcaps : True
#boxplot.showbox : True
#boxplot.showfliers : True
#boxplot.meanline : False

#boxplot.flierprops.color : 'k'
#boxplot.flierprops.marker : 'o'
#boxplot.flierprops.markersize : 6
#boxplot.flierprops.markerfacecolor : 'none'
#boxplot.flierprops.markeredgecolor : 'k'
#boxplot.flierprops.linestyle : 'none'
#boxplot.flierprops.linewidth : 1.0

#boxplot.boxprops.color : 'k'
#boxplot.boxprops.linewidth : 1.0
#boxplot.boxprops.linestyle : '-'

#boxplot.whiskerprops.color : 'k'
#boxplot.whiskerprops.linewidth : 1.0
#boxplot.whiskerprops.linestyle : '-'

#boxplot.capprops.color : 'k'
#boxplot.capprops.linewidth : 1.0
#boxplot.capprops.linestyle : '-'

#boxplot.medianprops.color : 'C1'
#boxplot.medianprops.linewidth : 1.0
#boxplot.medianprops.linestyle : '-'

#boxplot.meanprops.color : 'C2'
#boxplot.meanprops.marker : '^'
#boxplot.meanprops.markersize : 6
#boxplot.meanprops.markerfacecolor : 'C2'
#boxplot.meanprops.markeredgecolor : 'C2'
#boxplot.meanprops.markersize : 6
#boxplot.meanprops.linestyle : 'none'
# boxplot.meanprops.linewidth : 1.0

### FONT
#
# font properties used by text.Text. See
# http://matplotlib.org/api/font_manager_api.html for more
# information on font properties. The 6 font properties used for font
# matching are given below with their default values.
#
# The font.family property has five values: 'serif' (e.g., Times),
# 'sans-serif' (e.g., Helvetica), 'cursive' (e.g., Zapf-Chancery),
# 'fantasy' (e.g., Western), and 'monospace' (e.g., Courier). Each of
# these font families has a default list of font names in decreasing
# order of priority associated with them. When text.usetex is False,
# font.family may also be one or more concrete font names.
#
# The font.style property has three values: normal (or roman), italic
# or oblique. The oblique style will be used for italic, if it is not
# present.
#
# The font.variant property has two values: normal or small-caps. For
# TrueType fonts, which are scalable fonts, small-caps is equivalent
# to using a font size of 'smaller', or about 83% of the current font
# size.
#
# The font.weight property has effectively 13 values: normal, bold,
# bolder, lighter, 100, 200, 300, ..., 900. Normal is the same as
# 400, and bold is 700. bolder and lighter are relative values with
# respect to the current weight.
#
# The font.stretch property has 11 values: ultra-condensed,
# extra-condensed, condensed, semi-condensed, normal, semi-expanded,
# expanded, extra-expanded, ultra-expanded, wider, and narrower. This
# property is not currently implemented.
#
# The font.size property is the default font size for text, given in pts.
# 10 pt is the standard value.
#
# font.family : sans-serif
# font.style : normal
# font.variant : normal
# font.weight : medium
# font.stretch : normal
# note that font.size controls default text sizes. To configure
# special text sizes tick labels, axes, labels, title, etc, see the rc
# settings for axes and ticks. Special text sizes can be defined
# relative to font.size, using the following values: xx-small, x-small,
# small, medium, large, x-large, xx-large, larger, or smaller
# font.size : 10.0
# font.serif : DejaVu Serif, Bitstream Vera Serif, New Century Schoolbook,
# --Century Schoolbook L, Utopia, ITC Bookman, Bookman, Nimbus Roman No9 L, Times New,
# --Roman, Times, Palatino, Charter, serif
# font.sans-serif : DejaVu Sans, Bitstream Vera Sans, Lucida Grande, Verdana, Geneva,
# --Lucid, Arial, Helvetica, Avant Garde, sans serif

6.2. matplotlib rcParams
### LaTeX customizations. See [http://wiki.scipy.org/Cookbook/Matplotlib/UsingTex](http://wiki.scipy.org/Cookbook/Matplotlib/UsingTex)

```python
#text.usetex : False # use latex for all text handling. The following fonts
# are supported through the usual rc parameter settings:
# new century schoolbook, bookman, times, palatino,
# zapf chancery, charter, serif, sans-serif, helvetica,
# avant garde, courier, monospace, computer modern roman,
# computer modern sans serif, computer modern typewriter
# If another font is desired which can loaded using the
# LaTeX \usepackage command, please inquire at the
# matplotlib mailing list
```

```python
#text.latex.unicode : False # use "ucs" and "inputenc" LaTeX packages for handling
# unicode strings.
```

```python
#text.latex.preamble : # IMPROPER USE OF THIS FEATURE WILL LEAD TO LATEX FAILURES
# AND IS THEREFORE UNSUPPORTED. PLEASE DO NOT ASK FOR HELP
# IF THIS FEATURE DOES NOT DO WHAT YOU EXPECT IT TO.
# preamble is a comma separated list of LaTeX statements
# that are included in the LaTeX document preamble.
# An example:
# text.latex.preamble : \usepackage{bm},\usepackage{euler}
# The following packages are always loaded with usetex, so
# beware of package collisions: color, geometry, graphicx,
# type1cm, textcomp. Adobe Postscript (PSSNFS) font packages
# may also be loaded, depending on your font settings
```

```python
#text.dvipnghack : None # some versions of dvipng don't handle alpha
# channel properly. Use True to correct
# and flush ~/.matplotlib/tex.cache
# before testing and False to force
# correction off. None will try and
# guess based on your dvipng version
```

```python
#text.hinting : auto # May be one of the following:
# 'none': Perform no hinting
# 'auto': Use FreeType's autohinter
# 'native': Use the hinting information in the
# font file, if available, and if your
# FreeType library supports it
# 'either': Use the native hinting information,
# or the autohinter if none is available.
```
# For backward compatibility, this value may also be True === 'auto' or False === 'none'.

#text.hinting_factor : 8 # Specifies the amount of softness for hinting in the horizontal direction. A value of 1 will hint to full pixels. A value of 2 will hint to half pixels etc.

#text.antialiased : True # If True (default), the text will be antialiased. This only affects the Agg backend.

# The following settings allow you to select the fonts in math mode. They map from a TeX font name to a fontconfig font pattern. These settings are only used if mathtext.fontset is 'custom'. Note that this "custom" mode is unsupported and may go away in the future.

#mathtext.cal : cursive
#mathtext.rm : serif
#mathtext.tt : monospace
#mathtext.it : serif:italic
#mathtext.bf : serif:bold
#mathtext.sf : sans
#mathtext.fontset : dejavusans # Should be 'dejavusans' (default), 'dejavuserif', 'cm' (Computer Modern), 'stix', 'stixsans' or 'custom'

#mathtext.fallback_to_cm : True # When True, use symbols from the Computer Modern fonts when a symbol can not be found in one of the custom math fonts.

#mathtext.default : it # The default font to use for math. Can be any of the LaTeX font names, including the special name "regular" for the same font used in regular text.

### AXES

# default face and edge color, default tick sizes, default font sizes for ticklabels, and so on. See http://matplotlib.org/api/axes_api.html#module-matplotlib.axes

#axes.facecolor : white # axes background color
#axes.edgecolor : black # axes edge color
#axes.linewidth : 0.8 # edge linewidth
#axes.grid : False # display grid or not
#axes.titlepad : 6.0 # pad between axes and title in points
#axes.labelpad : 4.0 # space between label and axis
#axes.labelweight : normal # weight of the x and y labels
#axes.labelcolor : black
#axes.axisbelow : 'line' # draw axis gridlines and ticks below # patches (True); above patches but below # lines ('line'); or above all (False)

#axes.formatter.limits : -7, 7 # use scientific notation if log10 of the axis range is smaller than the first or larger than the second
#axes.formatter.use_locale : False # When True, format tick labels
    # according to the user's locale.
    # For example, use ',' as a decimal
    # separator in the fr_FR locale.
#axes.formatter.use_mathtext : False # When True, use mathtext for scientific
    # notation.
#axes.formatter.useoffset : True # If True, the tick label formatter
    # will default to labeling ticks relative
    # to an offset when the data range is
    # small compared to the minimum absolute
    # value of the data.
#axes.formatter.offset_threshold : 4 # When useoffset is True, the offset
    # will be used when it can remove
    # at least this number of significant
    # digits from tick labels.

# axes.spines.left : True # display axis spines
# axes.spines.bottom : True
# axes.spines.top : True
# axes.spines.right : True

#axes.unicode_minus : True # use unicode for the minus symbol
    # rather than hyphen. See
    # http://en.wikipedia.org/wiki/Plus_and_minus_signs

#axes.prop_cycle : cycler('color',
    #   ["0072B2", "DDCC78", "1E73C3", "CF0050", "A63600", "F46D61",
    #    "61A3B5", "4DAED2", "9E81B0", "FC8D62", "7468CB", "E5C494",
    #    "BBB465", "000000"])
    # color cycle for plot lines
    # as list of string colorspecs:
    # single letter, long name, or
    # web-style hex

#axes.autolimit_mode : data # How to scale axes limits to the data.
    # Use "data" to use data limits, plus some margin
    # Use "round_number" move to the nearest "round" number
#axes.xmargin : .05 # x margin. See `axes.Axes.margins`
#axes.ymargin : .05 # y margin See `axes.Axes.margins`

#polaraxes.grid : True # display grid on polar axes
#axes3d.grid : True # display grid on 3d axes

### DATES
# These control the default format strings used in AutoDateFormatter.
# Any valid format datetime format string can be used (see the python
# `datetime` for details). For example using `%Y` will use the locale date representation
# `%X` will use the locale time representation and `%C` will use the full locale datetime
# representation.
# These values map to the scales:
#     `{year': 365, 'month': 30, 'day': 1, 'hour': 1/24, 'minute': 1 / (24 * 60)}`

# date.autoformatter.year : %Y
# date.autoformatter.month : %Y-%m
# date.autoformatter.day : %Y-%m-%d
# date.autoformatter.hour : %H:%M
# date.autoformatter.minute : %d %H:%M
# date.autoformatter.second : %H:%M:%S
# date.autoformatter.microsecond : %M:%S.%f

### TICKS
# see http://matplotlib.org/api/axis_api.html#matplotlib.axis.Tick
#xtick.top : False # draw ticks on the top side
#xtick.bottom : True # draw ticks on the bottom side
#xtick.major.size : 3.5 # major tick size in points
#xtick.minor.size : 2 # minor tick size in points
#xtick.major.width : 0.8 # major tick width in points
#xtick.minor.width : 0.6 # minor tick width in points
#xtick.major.pad : 3.5 # distance to major tick label in points
#xtick.minor.pad : 3.4 # distance to the minor tick label in points
#xtick.color : k # color of the tick labels
#xtick.labelsize : medium # fontsize of the tick labels
#xtick.direction : out # direction: in, out, or inout
#xtick.minor.visible : False # visibility of minor ticks on x-axis
#xtick.major.top : True # draw x axis top major ticks
#xtick.major.bottom : True # draw x axis bottom major ticks
#xtick.minor.top : True # draw x axis top minor ticks
#xtick.minor.bottom : True # draw x axis bottom minor ticks

#ytick.left : True # draw ticks on the left side
#ytick.right : False # draw ticks on the right side
#ytick.major.size : 3.5 # major tick size in points
#ytick.minor.size : 2 # minor tick size in points
#ytick.major.width : 0.8 # major tick width in points
#ytick.minor.width : 0.6 # minor tick width in points
#ytick.major.pad : 3.5 # distance to major tick label in points
#ytick.minor.pad : 3.4 # distance to the minor tick label in points
#ytick.color : k # color of the tick labels
#ytick.labelsize : medium # fontsize of the tick labels
#ytick.direction : out # direction: in, out, or inout
#ytick.minor.visible : False # visibility of minor ticks on y-axis
#ytick.major.left : True # draw y axis left major ticks
#ytick.major.right : True # draw y axis right major ticks
#ytick.minor.left : True # draw y axis left minor ticks
#ytick.minor.right : True # draw y axis right minor ticks

### GRIDS
#grid.color : b0b0b0 # grid color
#grid.linestyle : - # solid
#grid.linewidth : 0.8 # in points
#grid.alpha : 1.0 # transparency, between 0.0 and 1.0

### Legend
#legend.loc : best
#legend.frameon : True # if True, draw the legend on a background patch

6.2. matplotlib rcParams
#legend.framealpha : 0.8  # legend patch transparency
#legend.facecolor : inherit  # inherit from axes.facecolor; or color spec
#legend.edgecolor : 0.8  # background patch boundary color
#legend.fancybox : True  # if True, use a rounded box for the
                      # legend background, else a rectangle
#legend.shadow : False  # if True, give background a shadow effect
#legend.numpoints : 1  # the number of marker points in the legend line
#legend.scatterpoints : 1  # number of scatter points
#legend.markerscale : 1.0  # the relative size of legend markers vs. original
#legend.fontsize : medium

# Dimensions as fraction of fontsize:
#legend.borderpad : 0.4  # border whitespace
#legend.labelsacing : 0.5  # the vertical space between the legend entries
#legend.handlelength : 2.0  # the length of the legend lines
#legend.handleheight : 0.7  # the height of the legend handle
#legend.handletextpad : 0.8  # the space between the legend line and legend text
#legend.borderaxespad : 0.5  # the border between the axes and legend edge
#legend.columnspacing : 2.0  # column separation

### FIGURE

# See http://matplotlib.org/api/figure_api.html#matplotlib.figure.Figure
#figure.titlesize : large  # size of the figure title (Figure.suptitle())
#figure.titleweight : normal  # weight of the figure title
#figure.figsize : 6.4, 4.8  # figure size in inches
#figure.dpi : 100  # figure dots per inch
#figure.facecolor : white  # figure facecolor; 0.75 is scalar gray
#figure.edgecolor : white  # figure edgecolor
#figure.autolayout : False  # When True, automatically adjust subplot
                      # parameters to make the plot fit the figure
#figure.max_open_warning : 20  # The maximum number of figures to open through
                              # the pyplot interface before emitting a warning.
                              # If less than one this feature is disabled.

# The figure subplot parameters. All dimensions are a fraction of the
#figure.subplot.left : 0.125  # the left side of the subplots of the figure
#figure.subplot.right : 0.9  # the right side of the subplots of the figure
#figure.subplot.bottom : 0.11  # the bottom of the subplots of the figure
#figure.subplot.top : 0.88  # the top of the subplots of the figure
#figure.subplot.wspace : 0.2  # the amount of width reserved for blank space between
                             # subplots,
                             # expressed as a fraction of the average axis width
#figure.subplot.hspace : 0.2  # the amount of height reserved for white space between
                             # subplots,
                             # expressed as a fraction of the average axis height

### IMAGES
#image.aspect : equal  # equal | auto | a number
#image.interpolation : nearest  # see help(imshow) for options
#image.cmap : viridis  # A colormap name, gray etc...
#image.lut : 256  # the size of the colormap lookup table
#image.origin : upper  # lower | upper
#image.resample : True
#image.composite_image : True # When True, all the images on a set of axes are
# combined into a single composite image before
# saving a figure as a vector graphics file,
# such as a PDF.

### CONTOUR PLOTS
#contour.negative_linestyle : dashed # dashed | solid
#contour.corner_mask : True # True | False | legacy

### ERRORBAR PLOTS
#errorbar.capsize : 0 # length of end cap on error bars in pixels

### HISTOGRAM PLOTS
#hist.bins : 10 # The default number of histogram bins.
# If Numpy 1.11 or later is
# installed, may also be `auto`

### SCATTER PLOTS
#scatter.marker : o # The default marker type for scatter plots.

### Agg rendering
### Warning: experimental, 2008/10/10
#agg.path.chunksize : 0 # 0 to disable; values in the range
# 10000 to 100000 can improve speed slightly
# and prevent an Agg rendering failure
# when plotting very large data sets,
# especially if they are very gappy.
# It may cause minor artifacts, though.
# A value of 20000 is probably a good
# starting point.

### SAVING FIGURES
#path.simplify : True # When True, simplify paths by removing "invisible"
# points to reduce file size and increase rendering
# speed
#path.simplify_threshold : 0.1 # The threshold of similarity below which
# vertices will be removed in the simplification
# process
#path.snap : True # When True, rectilinear axis-aligned paths will be snapped to
# the nearest pixel when certain criteria are met. When False,
# paths will never be snapped.
#path.sketch : None # May be none, or a 3-tuple of the form (scale, length,
# randomness).
# *scale* is the amplitude of the wiggle
# perpendicular to the line (in pixels). *length*
# is the length of the wiggle along the line (in
# pixels). *randomness* is the factor by which
# the length is randomly scaled.

# the default savefig params can be different from the display params
# e.g., you may want a higher resolution, or to make the figure
# background white
#savefig.dpi : figure # figure dots per inch or 'figure'
#savefig.facecolor : white # figure facecolor when saving
# savefig.edgecolor : white  # figure edgecolor when saving
# savefig.format : png  # png, ps, pdf, svg
# savefig.bbox : standard  # 'tight' or 'standard'.
# 'tight' is incompatible with pipe-based animation
# backends but will work with temporary file based ones:
# e.g. setting animation.writer to ffmpeg will not work,
# use ffmpeg_file instead
# savefig.pad_inches : 0.1  # Padding to be used when bbox is set to 'tight'
# savefig.jpeg_quality : 95  # when a jpeg is saved, the default quality parameter.
# savefig.directory : ~  # default directory in savefig dialog box,
# leave empty to always use current working directory
# savefig.transparent : False  # setting that controls whether figures are saved with a
# transparent background by default

# tk backend params
# tk.window_focus : False  # Maintain shell focus for TkAgg

# ps backend params
# ps.papersize : letter  # auto, letter, legal, ledger, A0-A10, B0-B10
# ps.useafm : False  # use of afm fonts, results in small files
# ps.usedistiller : False  # can be: None, ghostscript or xpdf
# Experimental: may produce smaller files.
# xpdf intended for production of publication,
# but requires ghostscript, xpdf and ps2eps
# ps.distiller.res : 6000  # dpi
# ps.fonttype : 3  # Output Type 3 (Type3) or Type 42 (TrueType)

# pdf backend params
# pdf.compression : 6  # integer from 0 to 9
# 0 disables compression (good for debugging)
# pdf.fonttype : 3  # Output Type 3 (Type3) or Type 42 (TrueType)

# svg backend params
# svg.image_inline : True  # write raster image data directly into the svg file
# svg.fonttype : 'path'  # How to handle SVG fonts:
# 'none': Assume fonts are installed on the machine where the SVG will be viewed.
# 'path': Embed characters as paths -- supported by most SVG renderers
# 'svgfont': Embed characters as SVG fonts -- supported only by Chrome,
# Opera and Safari
# svg.hashsalt : None  # if not None, use this string as hash salt
# instead of uuid4

# docstring params
# docstring.hardcopy = False  # set this when you want to generate hardcopy docstring

# Set the verbose flags. This controls how much information
# matplotlib gives you at runtime and where it goes. The verbosity
# levels are: silent, helpful, debug, debug-annoying. Any level is
# inclusive of all the levels below it. If your setting is "debug",
# you'll get all the debug and helpful messages. When submitting
# problems to the mailing-list, please set verbose to "helpful" or "debug"
# and paste the output into your report.
# The "fileo" gives the destination for any calls to verbose.report.
# These objects can a filename, or a filehandle like sys.stdout.
#
# You can override the rc default verbosity from the command line by
# giving the flags --verbose-LEVEL where LEVEL is one of the legal
# levels, e.g., --verbose-helpful.
#
# You can access the verbose instance in your code
# from matplotlib import verbose.
#verbose.level : silent # one of silent, helpful, debug, debug-annoying
#verbose.fileo : sys.stdout # a log filename, sys.stdout or sys.stderr
#
# Event keys to interact with figures/plots via keyboard.
# Customize these settings according to your needs.
# Leave the field(s) empty if you don't need a key-map. (i.e., fullscreen :)

#keymap.fullscreen : f, ctrl+f # toggling
#keymap.home : h, r, home # home or reset mnemonic
#keymap.back : left, c, backspace # forward / backward keys to enable
#keymap.forward : right, v # left handed quick navigation
#keymap.pan : p # pan mnemonic
#keymap.zoom : o # zoom mnemonic
#keymap.save : s # saving current figure
#keymap.quit : ctrl+w, cmd+w # close the current figure
#keymap.grid : g # switching on/off a grid in current axes
#keymap.yscale : l # toggle scaling of y-axes (log/linear)
#keymap.xscale : L, k # toggle scaling of x-axes (log/linear)
#keymap.all_axes : a # enable all axes

# Control location of examples data files
#examples.directory : '' # directory to look in for custom installation

###ANIMATION settings
#animation.html : 'none' # How to display the animation as HTML in
# the IPython notebook. html5' uses
# HTML5 video tag.
#animation.writer : ffmpeg # MovieWriter 'backend' to use
#animation.codec : h264 # Codec to use for writing movie
#animation.bitrate : -1 # Controls size/quality tradeoff for movie.
# -1 implies let utility auto-determine
#animation.frame_format : 'png' # Controls frame format used by temp files
#animation.ffmpeg_path : 'ffmpeg' # Path to ffmpeg binary. Without full path
# $PATH is searched
#animation.ffmpeg_args : '' # Additional arguments to pass to ffmpeg
#animation.avconv_path : 'avconv' # Path to avconv binary. Without full path
# $PATH is searched
#animation.avconv_args : '' # Additional arguments to pass to avconv
#animation.mencoder_path : 'mencoder' # Path to mencoder binary. Without full path
# $PATH is searched
#animation.mencoder_args : '' # Additional arguments to pass to mencoder
#animation.convert_path : 'convert' # Path to ImageMagick's convert binary.

6.2. matplotlib rcParams
# On Windows use the full path since convert
# is also the name of a system tool.
7.1 Interactive navigation

All figure windows come with a navigation toolbar, which can be used to navigate through the data set. Here is a description of each of the buttons at the bottom of the toolbar.

The **Home, Forward and Back buttons** These are akin to a web browser’s home, forward and back controls. Forward and Back are used to navigate back and forth between previously defined views. They have no meaning unless you have already navigated somewhere else using the pan and zoom buttons. This is analogous to trying to click Back on your web browser before visiting a new page or Forward before you have gone back to a page – nothing happens. Home always takes you to the first, default view of your data. Again, all of these buttons should feel very familiar to any user of a web browser.
The Pan/Zoom button  This button has two modes: pan and zoom. Click the toolbar button to activate panning and zooming, then put your mouse somewhere over an axes. Press the left mouse button and hold it to pan the figure, dragging it to a new position. When you release it, the data under the point where you pressed will be moved to the point where you released. If you press ‘x’ or ‘y’ while panning the motion will be constrained to the x or y axis, respectively. Press the right mouse button to zoom, dragging it to a new position. The x axis will be zoomed in proportionately to the rightward movement and zoomed out proportionately to the leftward movement. The same is true for the y axis and up/down motions. The point under your mouse when you begin the zoom remains stationary, allowing you to zoom in or out around that point as much as you wish. You can use the modifier keys ‘x’, ‘y’ or ‘CONTROL’ to constrain the zoom to the x axis, the y axis, or aspect ratio preserve, respectively.

With polar plots, the pan and zoom functionality behaves differently. The radius axis labels can be dragged using the left mouse button. The radius scale can be zoomed in and out using the right mouse button.

The Zoom-to-rectangle button  Click this toolbar button to activate this mode. Put your mouse somewhere over an axes and press the left mouse button. Drag the mouse while holding the button to a new location and release. The axes view limits will be zoomed to the rectangle you have defined. There is also an experimental ‘zoom out to rectangle’ in this mode with the right button, which will place your entire axes in the region defined by the zoom out rectangle.

The Subplot-configuration button  Use this tool to configure the appearance of the subplot: you can stretch or compress the left, right, top, or bottom side of the subplot, or the space between the rows or space between the columns.
The **Save button** Click this button to launch a file save dialog. You can save files with the following extensions: png, ps, eps, svg and pdf.

### 7.1.1 Navigation Keyboard Shortcuts

The following table holds all the default keys, which can be overwritten by use of your matplotlibrc (#keymap.*).

<table>
<thead>
<tr>
<th>Command</th>
<th>Keyboard Shortcut(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Home/Reset</td>
<td>h or r or home</td>
</tr>
<tr>
<td>Back</td>
<td>e or left arrow or backspace</td>
</tr>
<tr>
<td>Forward</td>
<td>v or right arrow</td>
</tr>
<tr>
<td>Pan/Zoom</td>
<td>p</td>
</tr>
<tr>
<td>Zoom-to-rect</td>
<td>o</td>
</tr>
<tr>
<td>Save</td>
<td>ctrl + s</td>
</tr>
<tr>
<td>Toggle fullscreen</td>
<td>f or ctrl + f</td>
</tr>
<tr>
<td>Close plot</td>
<td>ctrl + w</td>
</tr>
<tr>
<td>Constrain pan/zoom to x axis</td>
<td>hold x when panning/zooming with mouse</td>
</tr>
<tr>
<td>Constrain pan/zoom to y axis</td>
<td>hold y when panning/zooming with mouse</td>
</tr>
<tr>
<td>Preserve aspect ratio</td>
<td>hold CONTROL when panning/zooming with mouse</td>
</tr>
<tr>
<td>Toggle grid</td>
<td>g when mouse is over an axes</td>
</tr>
<tr>
<td>Toggle x axis scale (log/linear)</td>
<td>L or k when mouse is over an axes</td>
</tr>
<tr>
<td>Toggle y axis scale (log/linear)</td>
<td>l when mouse is over an axes</td>
</tr>
</tbody>
</table>

If you are using matplotlib.pyplot the toolbar will be created automatically for every figure. If you are writing your own user interface code, you can add the toolbar as a widget. The exact syntax depends on your UI, but we have examples for every supported UI in the matplotlib/examples/user_interfaces directory. Here is some example code for GTK:

```python
import gtk

from matplotlib.figure import Figure
from matplotlib.backends.backend_gtkagg import FigureCanvasGTKAgg as FigureCanvas
from matplotlib.backends.backend_gtkagg import NavigationToolbar2GTKAgg as NavigationToolbar

win = gtk.Window()
win.connect("destroy", lambda x: gtk.main_quit())
win.set_default_size(400,300)
win.set_title("Embedding in GTK")

vbox = gtk.VBox()
win.add(vbox)

fig = Figure(figsize=(5,4), dpi=100)
ax = fig.add_subplot(111)
ax.plot([1,2,3])
canvas = FigureCanvas(fig)  # a gtk.DrawingArea
vbox.pack_start(canvas)
```

7.1. Interactive navigation
7.2 Using matplotlib in a python shell

**Warning:** This page is significantly out of date

By default, matplotlib defers drawing until the end of the script because drawing can be an expensive operation, and you may not want to update the plot every time a single property is changed, only once after all the properties have changed.

But when working from the python shell, you usually do want to update the plot with every command, e.g., after changing the `xlabel()`, or the marker style of a line. While this is simple in concept, in practice it can be tricky, because matplotlib is a graphical user interface application under the hood, and there are some tricks to make the applications work right in a python shell.

7.2.1 IPython to the rescue

**Note:** The mode described here still exists for historical reasons, but it is highly advised not to use. It pollutes namespaces with functions that will shadow python built-in and can lead to hard to track bugs. To get IPython integration without imports the use of the `%matplotlib` magic is preferred. See python documentation.

Fortunately, ipython, an enhanced interactive python shell, has figured out all of these tricks, and is matplotlib aware, so when you start ipython in the `pylab` mode.

```
johnh@flag:~> ipython
Python 2.4.5 (#4, Apr 12 2008, 09:09:16)
IPython 0.9.0 -- An enhanced Interactive Python.

In [1]: %pylab
```

Welcome to pylab, a matplotlib-based Python environment.
For more information, type 'help(pylab).

```
In [2]: x = randn(10000)
In [3]: hist(x, 100)
```

it sets everything up for you so interactive plotting works as you would expect it to. Call `figure()` and a figure window pops up, call `plot()` and your data appears in the figure window.
Note in the example above that we did not import any matplotlib names because in pylab mode, ipython will import them automatically. ipython also turns on interactive mode for you, which causes every pyplot command to trigger a figure update, and also provides a matplotlib aware run command to run matplotlib scripts efficiently. ipython will turn off interactive mode during a run command, and then restore the interactive state at the end of the run so you can continue tweaking the figure manually.

There has been a lot of recent work to embed ipython, with pylab support, into various GUI applications, so check on the ipython mailing list for the latest status.

### 7.2.2 Other python interpreters

If you can’t use ipython, and still want to use matplotlib/pylab from an interactive python shell, e.g., the plain-ole standard python interactive interpreter, you are going to need to understand what a matplotlib backend is. **What is a backend?**

With the TkAgg backend, which uses the Tkinter user interface toolkit, you can use matplotlib from an arbitrary non-gui python shell. Just set your backend : TkAgg and interactive : True in your matplotlibrc file (see Customizing matplotlib) and fire up python. Then:

```python
>>> from pylab import *
>>> plot([1,2,3])
>>> xlabel('hi mom')
```

should work out of the box. This is also likely to work with recent versions of the qt4agg and gtkagg backends, and with the macosx backend on the Macintosh. Note, in batch mode, i.e. when making figures from scripts, interactive mode can be slow since it redraws the figure with each command. So you may want to think carefully before making this the default behavior via the matplotlibrc file instead of using the functions listed in the next section.

Gui shells are at best problematic, because they have to run a mainloop, but interactive plotting also involves a mainloop. Ipython has sorted all this out for the primary matplotlib backends. There may be other shells and IDEs that also work with matplotlib in interactive mode, but one obvious candidate does not: the python IDLE IDE is a Tkinter gui app that does not support pylab interactive mode, regardless of backend.

### 7.2.3 Controlling interactive updating

The interactive property of the pyplot interface controls whether a figure canvas is drawn on every pyplot command. If interactive is False, then the figure state is updated on every plot command, but will only be drawn on explicit calls to draw(). When interactive is True, then every pyplot command triggers a draw.

The pyplot interface provides 4 commands that are useful for interactive control.

- `isinteractive()` returns the interactive setting True|False
- `ion()` turns interactive mode on
- `ioff()` turns interactive mode off
- `draw()` forces a figure redraw
When working with a big figure in which drawing is expensive, you may want to turn matplotlib’s interactive setting off temporarily to avoid the performance hit:

```
>>> # create big-expensive-figure
>>> ioff()  # turn updates off
>>> title('now how much would you pay?')
>>> xticklabels(fontsize=20, color='green')
>>> draw()  # force a draw
>>> savefig('alldone', dpi=300)
>>> close()
>>> ion()  # turn updating back on
>>> plot(rand(20), mfc='g', mec='r', ms=40, mew=4, ls='--', lw=3)
```

### 7.3 Event handling and picking

matplotlib works with a number of user interface toolkits (wxpython, tkinter, qt4, gtk, and macosx) and in order to support features like interactive panning and zooming of figures, it is helpful to the developers to have an API for interacting with the figure via key presses and mouse movements that is “GUI neutral” so we don’t have to repeat a lot of code across the different user interfaces. Although the event handling API is GUI neutral, it is based on the GTK model, which was the first user interface matplotlib supported. The events that are triggered are also a bit richer vis-a-vis matplotlib than standard GUI events, including information like which `matplotlib.axes.Axes` the event occurred in. The events also understand the matplotlib coordinate system, and report event locations in both pixel and data coordinates.

#### 7.3.1 Event connections

To receive events, you need to write a callback function and then connect your function to the event manager, which is part of the `FigureCanvasBase`. Here is a simple example that prints the location of the mouse click and which button was pressed:

```
fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(np.random.rand(10))
def onclick(event):
    print('button=%d, x=%d, y=%d, xdata=%f, ydata=%f'
          '(event.button, event.x, event.y, event.xdata, event.ydata))

cid = fig.canvas.mpl_connect('button_press_event', onclick)
``` 

The `FigureCanvas` method `mpl_connect()` returns a connection id which is simply an integer. When you want to disconnect the callback, just call:

```
fig.canvas.mpl_disconnect(cid)
```

**Note:** The canvas retains only weak references to the callbacks. Therefore if a callback is a method of
a class instance, you need to retain a reference to that instance. Otherwise the instance will be garbage-collected and the callback will vanish.

Here are the events that you can connect to, the class instances that are sent back to you when the event occurs, and the event descriptions:

<table>
<thead>
<tr>
<th>Event name</th>
<th>Class and description</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘button_press_event’</td>
<td>MouseEvent - mouse button is pressed</td>
</tr>
<tr>
<td>‘button_release_event’</td>
<td>MouseEvent - mouse button is released</td>
</tr>
<tr>
<td>‘draw_event’</td>
<td>DrawEvent - canvas draw</td>
</tr>
<tr>
<td>‘key_press_event’</td>
<td>KeyEvent - key is pressed</td>
</tr>
<tr>
<td>‘key_release_event’</td>
<td>KeyEvent - key is released</td>
</tr>
<tr>
<td>‘motion_notify_event’</td>
<td>MouseEvent - mouse motion</td>
</tr>
<tr>
<td>‘pick_event’</td>
<td>PickEvent - an object in the canvas is selected</td>
</tr>
<tr>
<td>‘resize_event’</td>
<td>ResizeEvent - figure canvas is resized</td>
</tr>
<tr>
<td>‘scroll_event’</td>
<td>MouseEvent - mouse scroll wheel is rolled</td>
</tr>
<tr>
<td>‘figure_enter_event’</td>
<td>LocationEvent - mouse enters a new figure</td>
</tr>
<tr>
<td>‘figure_leave_event’</td>
<td>LocationEvent - mouse leaves a figure</td>
</tr>
<tr>
<td>‘axes_enter_event’</td>
<td>LocationEvent - mouse enters a new axes</td>
</tr>
<tr>
<td>‘axes_leave_event’</td>
<td>LocationEvent - mouse leaves an axes</td>
</tr>
</tbody>
</table>

7.3.2 Event attributes

All matplotlib events inherit from the base class `matplotlib.backend_bases.Event`, which store the attributes:

- **name** the event name
- **canvas** the FigureCanvas instance generating the event
- **guiEvent** the GUI event that triggered the matplotlib event

The most common events that are the bread and butter of event handling are key press/release events and mouse press/release and movement events. The `KeyEvent` and `MouseEvent` classes that handle these events are both derived from the `LocationEvent`, which has the following attributes:

- **x** x position - pixels from left of canvas
- **y** y position - pixels from bottom of canvas
- **inaxes** the Axes instance if mouse is over axes
- **xdata** x coord of mouse in data coords
- **ydata** y coord of mouse in data coords

Let’s look a simple example of a canvas, where a simple line segment is created every time a mouse is pressed:

```python
from matplotlib import pyplot as plt

class LineBuilder:
```

7.3. Event handling and picking
```python
def __init__(self, line):
    self.line = line
    self.xs = list(line.get_xdata())
    self.ys = list(line.get_ydata())
    self.cid = line.figure.canvas.mpl_connect('button_press_event', self)

def __call__(self, event):
    print('click', event)
    if event.inaxes!=self.line.axes:
        return
    self.xs.append(event.xdata)
    self.ys.append(event.ydata)
    self.line.set_data(self.xs, self.ys)
    self.line.figure.canvas.draw()

fig = plt.figure()
ax = fig.add_subplot(111)
ax.set_title('click to build line segments')
line, = ax.plot([0], [0])  # empty line
linebuilder = LineBuilder(line)
plt.show()
```

The `MouseEvent` that we just used is a `LocationEvent`, so we have access to the data and pixel coordinates in `event.x` and `event.xdata`. In addition to the `LocationEvent` attributes, it has:

- **button**: button pressed None, 1, 2, 3, ‘up’, ‘down’ (up and down are used for scroll events)
- **key**: the key pressed: None, any character, ‘shift’, ‘win’, or ‘control’

**Draggable rectangle exercise**

Write draggable rectangle class that is initialized with a `Rectangle` instance but will move its x,y location when dragged. Hint: you will need to store the original xy location of the rectangle which is stored as `rect.xy` and connect to the press, motion and release mouse events. When the mouse is pressed, check to see if the click occurs over your rectangle (see `matplotlib.patches.Rectangle.contains()`) and if it does, store the rectangle xy and the location of the mouse click in data coords. In the motion event callback, compute the deltax and deltay of the mouse movement, and add those deltas to the origin of the rectangle you stored. The redraw the figure. On the button release event, just reset all the button press data you stored as None.

Here is the solution:

```python
import numpy as np
import matplotlib.pyplot as plt

class DraggableRectangle:
    def __init__(self, rect):
        self.rect = rect
        self.press = None

    def connect(self):
        'connect to all the events we need'
```

196 Chapter 7. Interactive plots
self.cidpress = self.rect.figure.canvas.mpl_connect('button_press_event', self.on_press)
self.cidrelease = self.rect.figure.canvas.mpl_connect('button_release_event', self.on_release)
self.cidmotion = self.rect.figure.canvas.mpl_connect('motion_notify_event', self.on_motion)

def on_press(self, event):
    'on button press we will see if the mouse is over us and store some data'
    if event.inaxes != self.rect.axes: return

    contains, attrd = self.rect.contains(event)
    if not contains: return
    print('event contains', self.rect.xy)
x0, y0 = self.rect.xy
self.press = x0, y0, event.xdata, event.ydata

def on_motion(self, event):
    'on motion we will move the rect if the mouse is over us'
    if self.press is None: return
    if event.inaxes != self.rect.axes: return
    x0, y0, xpress, ypress = self.press
    dx = event.xdata - xpress
    dy = event.ydata - ypress
    #print('%f, %f, %f, %f, %f, %f' %
    # (x0, xpress, event.xdata, dx, x0+dx))
    self.rect.set_x(x0+dx)
    self.rect.set_y(y0+dy)

    self.rect.figure.canvas.draw()

def on_release(self, event):
    'on release we reset the press data'
    self.press = None
    self.rect.figure.canvas.draw()

def disconnect(self):
    'disconnect all the stored connection ids'
    self.rect.figure.canvas.mpl_disconnect(self.cidpress)
    self.rect.figure.canvas.mpl_disconnect(self.cidrelease)
    self.rect.figure.canvas.mpl_disconnect(self.cidmotion)

fig = plt.figure()
ax = fig.add_subplot(111)
rects = ax.bar(range(10), 20*np.random.rand(10))
drs = []
for rect in rects:
    dr = DraggableRectangle(rect)
    dr.connect()
    drs.append(dr)

plt.show()
Extra credit: use the animation blit techniques discussed in the animations recipe to make the animated drawing faster and smoother.

Extra credit solution:

```python
# draggable rectangle with the animation blit techniques; see
# http://www.scipy.org/Cookbook/Matplotlib/Animations
import numpy as np
import matplotlib.pyplot as plt

class DraggableRectangle:
    lock = None  # only one can be animated at a time

    def __init__(self, rect):
        self.rect = rect
        self.press = None
        self.background = None

    def connect(self):
        'connect to all the events we need'
        self.cidpress = self.rect.figure.canvas.mpl_connect('button_press_event', self.on_press)
        self.cidrelease = self.rect.figure.canvas.mpl_connect('button_release_event', self.on_release)
        self.cidmotion = self.rect.figure.canvas.mpl_connect('motion_notify_event', self.on_motion)

    def on_press(self, event):
        'on button press we will see if the mouse is over us and store some data'
        if event.inaxes != self.rect.axes: return
        if DraggableRectangle.lock is not None: return
        contains, attrd = self.rect.contains(event)
        if not contains: return
        print('event contains', self.rect.xy)
        x0, y0 = self.rect.xy
        self.press = x0, y0, event.xdata, event.ydata
        DraggableRectangle.lock = self

        # draw everything but the selected rectangle and store the pixel buffer
        canvas = self.rect.figure.canvas
        axes = self.rect.axes
        self.rect.set_animated(True)
        canvas.draw()
        self.background = canvas.copy_from_bbox(self.rect.axes.bbox)
        # now redraw just the rectangle
        axes.draw_artist(self.rect)
        # and blit just the redrawn area
        canvas.blit(axes.bbox)

    def on_release(self, event):
        'button release to reset the press data'
        DraggableRectangle.lock = None

    def on_motion(self, event):
        'on motion we will move the rect if the mouse is over us'
        if DraggableRectangle.lock is not self:
            return

        if event.inaxes == self.rect.axes:
            x, y = self.rect.xy
            x+= event.xdata - self.press[2]
            y+= event.ydata - self.press[3]
            self.rect.set_x(x)
            self.rect.set_y(y)
```

Chapter 7. Interactive plots
if event.inaxes != self.rect.axes: return
x0, y0, xpress, ypress = self.press
dx = event.xdata - xpress
dy = event.ydata - ypress
self.rect.set_x(x0+dx)
self.rect.set_y(y0+dy)

canvas = self.rect.figure.canvas
axes = self.rect.axes
# restore the background region
canvas.restore_region(self.background)

# redraw just the current rectangle
axes.draw_artist(self.rect)

# blit just the redrawn area
canvas.blit(axes.bbox)

def on_release(self, event):
    'on release we reset the press data'
    if DraggableRectangle.lock is not self:
        return

    self.press = None
    DraggableRectangle.lock = None

    # turn off the rect animation property and reset the background
    self.rect.set_animated(False)
    self.background = None

    # redraw the full figure
    self.rect.figure.canvas.draw()

def disconnect(self):
    'disconnect all the stored connection ids'
    self.rect.figure.canvas.mpl_disconnect(self.cidpress)
    self.rect.figure.canvas.mpl_disconnect(self.cidrelease)
    self.rect.figure.canvas.mpl_disconnect(self.cidmotion)

fig = plt.figure()
ax = fig.add_subplot(111)
rects = ax.bar(range(10), 20*np.random.rand(10))
drs = []
for rect in rects:
    dr = DraggableRectangle(rect)
    dr.connect()
    drs.append(dr)

plt.show()
7.3.3 Mouse enter and leave

If you want to be notified when the mouse enters or leaves a figure or axes, you can connect to the figure/axes enter/leave events. Here is a simple example that changes the colors of the axes and figure background that the mouse is over:

```python
"""
Illustrate the figure and axes enter and leave events by changing the frame colors on enter and leave
"""
import matplotlib.pyplot as plt
def enter_axes(event):
    print('enter_axes', event.inaxes)
    event.inaxes.patch.set_facecolor('yellow')
    event.canvas.draw()
def leave_axes(event):
    print('leave_axes', event.inaxes)
    event.inaxes.patch.set_facecolor('white')
    event.canvas.draw()
def enter_figure(event):
    print('enter_figure', event.canvas.figure)
    event.canvas.figure.patch.set_facecolor('red')
    event.canvas.draw()
def leave_figure(event):
    print('leave_figure', event.canvas.figure)
    event.canvas.figure.patch.set_facecolor('grey')
    event.canvas.draw()

fig1 = plt.figure()
fig1.suptitle('mouse hover over figure or axes to trigger events')
ax1 = fig1.add_subplot(211)
ax2 = fig1.add_subplot(212)
fig1.canvas.mpl_connect('figure_enter_event', enter_figure)
fig1.canvas.mpl_connect('figure_leave_event', leave_figure)
fig1.canvas.mpl_connect('axes_enter_event', enter_axes)
fig1.canvas.mpl_connect('axes_leave_event', leave_axes)

fig2 = plt.figure()
fig2.suptitle('mouse hover over figure or axes to trigger events')
ax1 = fig2.add_subplot(211)
ax2 = fig2.add_subplot(212)
fig2.canvas.mpl_connect('figure_enter_event', enter_figure)
fig2.canvas.mpl_connect('figure_leave_event', leave_figure)
fig2.canvas.mpl_connect('axes_enter_event', enter_axes)
fig2.canvas.mpl_connect('axes_leave_event', leave_axes)

plt.show()
```
### 7.3.4 Object picking

You can enable picking by setting the `picker` property of an `Artist` (e.g., a matplotlib `Line2D`, `Text`, `Patch`, `Polygon`, `AxesImage`, etc...)

There are a variety of meanings of the `picker` property:

- **None** picking is disabled for this artist (default)
- **boolean** if True then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist
- **float** if picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if its data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event.
- **function** if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event. The signature is `hit, props = picker(artist, mouseevent)` to determine the hit test. If the mouse event is over the artist, return `hit=True` and `props` is a dictionary of properties you want added to the `PickEvent` attributes

After you have enabled an artist for picking by setting the `picker` property, you need to connect to the figure canvas pick_event to get pick callbacks on mouse press events. e.g.:

```python
def pick_handler(event):
    mouseevent = event.mouseevent
    artist = event.artist
    # now do something with this...
```

The `PickEvent` which is passed to your callback is always fired with two attributes:

- **mouseevent** the mouse event that generate the pick event. The mouse event in turn has attributes like `x` and `y` (the coords in display space, e.g., pixels from left, bottom) and `xdata`, `ydata` (the coords in data space). Additionally, you can get information about which buttons were pressed, which keys were pressed, which `Axes` the mouse is over, etc. See `matplotlib.backend_bases.MouseEvent` for details.

- **artist** the `Artist` that generated the pick event.

Additionally, certain artists like `Line2D` and `PatchCollection` may attach additional meta data like the indices into the data that meet the picker criteria (e.g., all the points in the line that are within the specified epsilon tolerance)

#### Simple picking example

In the example below, we set the line picker property to a scalar, so it represents a tolerance in points (72 points per inch). The onpick callback function will be called when the pick event it within the tolerance distance from the line, and has the indices of the data vertices that are within the pick distance tolerance. Our onpick callback function simply prints the data that are under the pick location. Different matplotlib Artists can attach different data to the `PickEvent`. For example, `Line2D` attaches the `ind` property, which are
the indices into the line data under the pick point. See `pick()` for details on the PickEvent properties of the line. Here is the code:

```python
import numpy as np
import matplotlib.pyplot as plt

fig = plt.figure()
ax = fig.add_subplot(111)
ax.set_title('click on points')

line, = ax.plot(np.random.rand(100), 'o', picker=5)  # 5 points tolerance

def onpick(event):
    thisline = event.artist
    xdata = thisline.get_xdata()
    ydata = thisline.get_ydata()
    ind = event.ind
    points = tuple(zip(xdata[ind], ydata[ind]))
    print('onpick points:', points)

fig.canvas.mpl_connect('pick_event', onpick)

plt.show()
```

**Picking exercise**

Create a data set of 100 arrays of 1000 Gaussian random numbers and compute the sample mean and standard deviation of each of them (hint: numpy arrays have a mean and std method) and make a xy marker plot of the 100 means vs the 100 standard deviations. Connect the line created by the plot command to the pick event, and plot the original time series of the data that generated the clicked on points. If more than one point is within the tolerance of the clicked on point, you can use multiple subplots to plot the multiple time series.

Exercise solution:

```python
""
compute the mean and stddev of 100 data sets and plot mean vs stddev.
When you click on one of the mu, sigma points, plot the raw data from
the dataset that generated the mean and stddev
""

import numpy as np
import matplotlib.pyplot as plt

X = np.random.rand(100, 1000)
x = np.mean(X, axis=1)
y = np.std(X, axis=1)

fig = plt.figure()
ax = fig.add_subplot(111)
ax.set_title('click on point to plot time series')
line, = ax.plot(x, y, 'o', picker=5)  # 5 points tolerance
```

202 Chapter 7. Interactive plots
```python
def onpick(event):
    if event.artist != line: return True

    N = len(event.ind)
    if not N: return True

    figi = plt.figure()
    for subplotnum, dataind in enumerate(event.ind):
        ax = figi.add_subplot(N,1,subplotnum+1)
        ax.plot(X[dataind])
        ax.text(0.05, 0.9, 'mu=%1.3f
        sigma=%1.3f'
        % (xs[dataind], ys[dataind]),
        transform=ax.transAxes, va='top')
        ax.set_ylim(-0.5, 1.5)
    figi.show()
    return True

figi.canvas.mpl_connect('pick_event', onpick)

plt.show()
```

7.3. Event handling and picking
8.1 Screenshots

Here you’ll find a host of example plots with the code that generated them.

8.1.1 Simple Plot

Here’s a very basic `plot()` with text labels:

```
0.00 0.25 0.50 0.75 1.00 1.25 1.50 1.75 2.00
time (s)
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
voltage (mV)
About as simple as it gets, folks
```
8.1.2 Subplot demo

Multiple axes (i.e. subplots) are created with the subplot() command:

![A tale of 2 subplots](image)

8.1.3 Histograms

The hist() command automatically generates histograms and returns the bin counts or probabilities:
8.1.4 Path demo

You can add arbitrary paths in matplotlib using the `matplotlib.path` module:
8.1.5 mplot3d

The mplot3d toolkit (see mplot3d tutorial and mplot3d Examples) has support for simple 3d graphs including surface, wireframe, scatter, and bar charts.
Thanks to John Porter, Jonathon Taylor, Reinier Heeres, and Ben Root for the mplot3d toolkit. This toolkit is included with all standard matplotlib installs.

### 8.1.6 Streamplot

The `streamplot()` function plots the streamlines of a vector field. In addition to simply plotting the streamlines, it allows you to map the colors and/or line widths of streamlines to a separate parameter, such as the speed or local intensity of the vector field.
This feature complements the `quiver()` function for plotting vector fields. Thanks to Tom Flannaghan and Tony Yu for adding the streamplot function.

### 8.1.7 Ellipses

In support of the Phoenix mission to Mars (which used matplotlib to display ground tracking of spacecraft), Michael Droettboom built on work by Charlie Moad to provide an extremely accurate 8-spline approximation to elliptical arcs (see `Arc`), which are insensitive to zoom level.
8.1.8 Bar charts

Bar charts are simple to create using the `bar()` command, which includes customizations such as error bars:
It’s also simple to create stacked bars (bar_stacked.py), or horizontal bar charts (barh_demo.py).

### 8.1.9 Pie charts

The `pie()` command allows you to easily create pie charts. Optional features include auto-labeling the percentage of area, exploding one or more wedges from the center of the pie, and a shadow effect. Take a close look at the attached code, which generates this figure in just a few lines of code.
8.1.10 Table demo

The `table()` command adds a text table to an axes.
### 8.1.11 Scatter demo

The `scatter()` command makes a scatter plot with (optional) size and color arguments. This example plots changes in Google’s stock price, with marker sizes reflecting the trading volume and colors varying with time. Here, the alpha attribute is used to make semitransparent circle markers.
8.1.12 Slider demo

Matplotlib has basic GUI widgets that are independent of the graphical user interface you are using, allowing you to write cross GUI figures and widgets. See matplotlib.widgets and the widget examples.
8.1.13 Fill demo

The `fill()` command lets you plot filled curves and polygons:
Thanks to Andrew Straw for adding this function.

### 8.1.14 Date demo

You can plot date data with major and minor ticks and custom tick formatters for both.
See `matplotlib.ticker` and `matplotlib.dates` for details and usage.

### 8.1.15 Log plots

The `semilogx()`, `semilogy()` and `loglog()` functions simplify the creation of logarithmic plots.
Thanks to Andrew Straw, Darren Dale and Gregory Lielens for contributions log-scaling infrastructure.

### 8.1.16 Polar plots

The `polar()` command generates polar plots.
8.1.17 Legends

The `legend()` command automatically generates figure legends, with MATLAB-compatible legend placement commands.
Thanks to Charles Twardy for input on the legend command.

### 8.1.18 Mathtext_examples

Below is a sampling of the many TeX expressions now supported by matplotlib’s internal mathtext engine. The mathtext module provides TeX style mathematical expressions using FreeType and the DejaVu, BaKoMa computer modern, or STIX fonts. See the `matplotlib.mathtext` module for additional details.
Matplotlib’s mathtext infrastructure is an independent implementation and does not require TeX or any external packages installed on your computer. See the tutorial at Writing mathematical expressions.

8.1.19 Native TeX rendering

Although matplotlib’s internal math rendering engine is quite powerful, sometimes you need TeX. Matplotlib supports external TeX rendering of strings with the `usetex` option.
8.1.20 EEG demo

You can embed matplotlib into pygtk, wx, Tk, or Qt applications. Here is a screenshot of an EEG viewer called pbrain.
The lower axes uses `specgram()` to plot the spectrogram of one of the EEG channels.

For examples of how to embed matplotlib in different toolkits, see:

- `user_interfaces example code: embedding_in_gtk2.py`
- `user_interfaces example code: embedding_in_wx2.py`
- `user_interfaces example code: mpl_with_glade.py`
- `user_interfaces example code: embedding_in_qt4.py`
- `user_interfaces example code: embedding_in_tk.py`

### 8.1.21 XKCD-style sketch plots

matplotlib supports plotting in the style of `xkcd`.

8.1. Screenshots
"Stove ownership" from XKCD by Randall Monroe
Here is a collection of short tutorials, examples and code snippets that illustrate some of the useful idioms and tricks to make snazzier figures and overcome some matplotlib warts.

### 8.2.1 Sharing axis limits and views

It’s common to make two or more plots which share an axis, e.g., two subplots with time as a common axis. When you pan and zoom around on one, you want the other to move around with you. To facilitate this, matplotlib Axes support a `sharex` and `sharey` attribute. When you create a `subplot()` or `axes()` instance, you can pass in a keyword indicating what axes you want to share with.

```
In [96]: t = np.arange(0, 10, 0.01)
In [97]: ax1 = plt.subplot(211)
In [98]: ax1.plot(t, np.sin(2*np.pi*t))
Out[98]: [<matplotlib.lines.Line2D object at 0x98719ec>]
In [99]: ax2 = plt.subplot(212, sharex=ax1)
```

---

8.2. Our Favorite Recipes

"The data so far" from XKCD by Randall Monroe
8.2.2 Easily creating subplots

In early versions of matplotlib, if you wanted to use the pythonic API and create a figure instance and from that create a grid of subplots, possibly with shared axes, it involved a fair amount of boilerplate code. e.g.

```python
# old style
fig = plt.figure()
ax1 = fig.add_subplot(221)
ax2 = fig.add_subplot(222, sharex=ax1, sharey=ax1)
ax3 = fig.add_subplot(223, sharex=ax1, sharey=ax1)
ax3 = fig.add_subplot(224, sharex=ax1, sharey=ax1)
```

Fernando Perez has provided a nice top level method to create in subplots() (note the “s” at the end) everything at once, and turn on x and y sharing for the whole bunch. You can either unpack the axes individually:

```python
# new style method 1; unpack the axes
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, sharex=True, sharey=True)
ax1.plot(x)
```

or get them back as a numrows x numcolumns object array which supports numpy indexing:

```python
# new style method 2; use an axes array
fig, axs = plt.subplots(2, 2, sharex=True, sharey=True)
asx[0,0].plot(x)
```

8.2.3 Fixing common date annoyances

matplotlib allows you to natively plots python datetime instances, and for the most part does a good job picking tick locations and string formats. There are a couple of things it does not handle so gracefully, and here are some tricks to help you work around them. We’ll load up some sample date data which contains datetime.date objects in a numpy record array:

```
In [63]: datafile = cbook.get_sample_data('goog.npy')
In [64]: r = np.load(datafile).view(np.recarray)
In [65]: r.dtype
Out[65]: dtype([('date', '|S4'), ('', '|V4'), ('open', '<f8'),
('high', '<f8'), ('low', '<f8'), ('close', '<f8'),
('volume', '<i8'), ('adj_close', '<f8'))]
In [66]: r.date
Out[66]: array(['2004-08-19', '2004-08-20', '2004-08-23', ...
'2008-10-10', '2008-10-13',
'2008-10-14'], dtype=object)
```
The dtype of the numpy record array for the field date is |O4 which means it is a 4-byte python object pointer; in this case the objects are datetime.date instances, which we can see when we print some samples in the ipython terminal window.

If you plot the data,

```python
In [67]: plot(r.date, r.close)
Out[67]: [<matplotlib.lines.Line2D object at 0x92a6b6c>]
```

you will see that the x tick labels are all squashed together.

![Default date handling can cause overlapping labels](image)

Another annoyance is that if you hover the mouse over the window and look in the lower right corner of the matplotlib toolbar (Interactive navigation) at the x and y coordinates, you see that the x locations are formatted the same way the tick labels are, e.g., “Dec 2004”. What we’d like is for the location in the toolbar to have a higher degree of precision, e.g., giving us the exact date out mouse is hovering over. To fix the first problem, we can use `matplotlib.figure.Figure.autofmt_xdate()` and to fix the second problem we can use the `ax.fmt_xdata` attribute which can be set to any function that takes a scalar and returns a string. matplotlib has a number of date formatters built in, so we’ll use one of those.

```python
plt.close('all')
fig, ax = plt.subplots(1)
ar.plot(r.date, r.close)

# rotate and align the tick labels so they look better
fig.autofmt_xdate()
```
# use a more precise date string for the x axis locations in the toolbar
import matplotlib.dates as mdates
ax.fmt_xdata = mdates.DateFormatter('%Y-%m-%d')
plt.title('fig.autofmt_xdate fixes the labels')

Now when you hover your mouse over the plotted data, you’ll see date format strings like 2004-12-01 in the toolbar.

8.2.4 Fill Between and Alpha

The fill_between() function generates a shaded region between a min and max boundary that is useful for illustrating ranges. It has a very handy where argument to combine filling with logical ranges, e.g., to just fill in a curve over some threshold value.

At its most basic level, fill_between can be use to enhance a graphs visual appearance. Let’s compare two graphs of a financial times with a simple line plot on the left and a filled line on the right.

import matplotlib.pyplot as plt
import numpy as np
import matplotlib.cbook as cbook
# load up some sample financial data
datafile = cbook.get_sample_data('goog.npy')

try:
    # Python3 cannot load python2 .npy files with datetime(object) arrays
    # unless the encoding is set to bytes. However this option was
    # not added until numpy 1.10 so this example will only work with
    # python 2 or with numpy 1.10 and later.
    r = np.load(datafile, encoding='bytes').view(np.recarray)
except TypeError:
    r = np.load(datafile).view(np.recarray)

# create two subplots with the shared x and y axes
fig, (ax1, ax2) = plt.subplots(1, 2, sharex=True, sharey=True)
pricemin = r.close.min()
ax1.plot(r.date, r.close, lw=2)
ax2.fill_between(r.date, pricemin, r.close, facecolor='blue', alpha=0.5)

for ax in ax1, ax2:
    ax.grid(True)
ax1.set_ylabel('price')
for label in ax2.get_yticklabels():
    label.set_visible(False)

fig.suptitle('Google (GOOG) daily closing price')
fig.autofmt_xdate()
The alpha channel is not necessary here, but it can be used to soften colors for more visually appealing plots. In other examples, as we’ll see below, the alpha channel is functionally useful as the shaded regions can overlap and alpha allows you to see both. Note that the postscript format does not support alpha (this is a postscript limitation, not a matplotlib limitation), so when using alpha save your figures in PNG, PDF or SVG.

Our next example computes two populations of random walkers with a different mean and standard deviation of the normal distributions from which the steps are drawn. We use shared regions to plot +/- one standard deviation of the mean position of the population. Here the alpha channel is useful, not just aesthetic.

```python
import matplotlib.pyplot as plt
import numpy as np

Nsteps, Nwalkers = 100, 250
x = np.arange(Nsteps)

# an (Nsteps x Nwalkers) array of random walk steps
S1 = 0.002 + 0.01*np.random.randn(Nsteps, Nwalkers)
S2 = 0.004 + 0.02*np.random.randn(Nsteps, Nwalkers)

# an (Nsteps x Nwalkers) array of random walker positions
X1 = S1.cumsum(axis=0)
X2 = S2.cumsum(axis=0)
```

![Google (GOOG) daily closing price](image)
# Nsteps length arrays empirical means and standard deviations of both populations over time
mu1 = X1.mean(axis=1)
sigma1 = X1.std(axis=1)
u2 = X2.mean(axis=1)
sigma2 = X2.std(axis=1)

# plot it!
fig, ax = plt.subplots(1)
ax.plot(t, mu1, lw=2, label='mean population 1', color='blue')
ax.plot(t, mu2, lw=2, label='mean population 2', color='yellow')
a.fill_between(t, mu1+sigma1, mu1-sigma1, facecolor='blue', alpha=0.5)
a.fill_between(t, mu2+sigma2, mu2-sigma2, facecolor='yellow', alpha=0.5)
a.set_title('random walkers empirical $\mu$ and $\pm \sigma$ interval')
a.legend(loc='upper left')
a.set_xlabel('num steps')
a.set_ylabel('position')
a.grid()

The *where* keyword argument is very handy for highlighting certain regions of the graph. *where* takes a boolean mask the same length as the x, ymin and ymax arguments, and only fills in the region where the boolean mask is True. In the example below, we simulate a single random walker and compute the analytic mean and standard deviation of the population positions. The population mean is shown as the black dashed line, and the plus/minus one sigma deviation from the mean is shown as the yellow filled region. We use the *where* mask $X>upper\_bound$ to find the region where the walker is above the one sigma boundary, and

8.2. Our Favorite Recipes 233
shade that region blue.

```python
np.random.seed(1234)
Nsteps = 500
t = np.arange(Nsteps)
mu = 0.002
sigma = 0.01

# the steps and position
S = mu + sigma*np.random.randn(Nsteps)
X = S.cumsum()

# the 1 sigma upper and lower analytic population bounds
lower_bound = mu*t - sigma*np.sqrt(t)
upper_bound = mu*t + sigma*np.sqrt(t)

fig, ax = plt.subplots(1)
ax.plot(t, X, lw=2, label='walker position', color='blue')
ax.plot(t, mu*t, lw=1, label='population mean', color='black', ls='--')
ax.fill_between(t, lower_bound, upper_bound, facecolor='yellow', alpha=0.5,
                label='1 sigma range')
ax.legend(loc='upper left')

# here we use the where argument to only fill the region where the
# walker is above the population 1 sigma boundary
ax.fill_between(t, upper_bound, X, where=X>upper_bound, facecolor='blue', alpha=0.5)
ax.set_xlabel('num steps')
ax.set_ylabel('position')
ax.grid()
```
Another handy use of filled regions is to highlight horizontal or vertical spans of an axes – for that matplotlib has some helper functions `axhspan()` and `axvspan()` and example `pylab_examples example code: axhspan_demo.py`.

### 8.2.5 Transparent, fancy legends

Sometimes you know what your data looks like before you plot it, and may know for instance that there won’t be much data in the upper right hand corner. Then you can safely create a legend that doesn’t overlay your data:

```python
ax.legend(loc='upper right')
```

Other times you don’t know where your data is, and `loc='best'` will try and place the legend:

```python
ax.legend(loc='best')
```

but still, your legend may overlap your data, and in these cases it’s nice to make the legend frame transparent.

```python
np.random.seed(1234)
fig, ax = plt.subplots(1)
ax.plot(np.random.randn(300), 'o-', label='normal distribution')
ax.plot(np.random.randn(300), 's-', label='uniform distribution')
ax.set_yscale('symlog')
ax.legend(loc='best', fancybox=True, framealpha=0.5)
```
8.2.6 Placing text boxes

When decorating axes with text boxes, two useful tricks are to place the text in axes coordinates (see Transformations Tutorial), so the text doesn’t move around with changes in x or y limits. You can also use the bbox property of text to surround the text with a Patch instance – the bbox keyword argument takes a dictionary with keys that are Patch properties.

```python
np.random.seed(1234)
fig, ax = plt.subplots(1)
x = 30*np.random.randn(10000)
mu = x.mean()
median = np.median(x)
sigma = x.std()
textstr = '\$
\mu=\%.2f$
\mathrm{median}=\%.2f
\sigma=\%.2f$''%(mu, median, sigma)
ax.hist(x, 50)
# these are matplotlib.patch.Patch properties
props = dict(boxstyle='round', facecolor='wheat', alpha=0.5)

# place a text box in upper left in axes coords
```

```python
ax.set_title('fancy, transparent legends')
```
\( \mu = 0.48 \)
\( \text{median} = 0.54 \)
\( \sigma = 29.86 \)
WHAT’S NEW IN MATPLOTLIB

For a list of all of the issues and pull requests since the last revision, see the GitHub Stats.

Table of Contents

• What’s new in matplotlib
  – New in matplotlib 2.0
    * Default style changes
    * Improved color conversion API and RGBA support
    * New Configuration (rcParams)
    * Qualitative colormaps
    * Axis offset label now responds to labelcolor
    * Improved offset text choice
    * Style parameter blacklist
    * Change in default font
    * Faster text rendering
    * Improvements for the Qt figure options editor
    * Improved image support
    * Support for HiDPI (Retina) displays in the NbAgg and WebAgg backends
    * Change in the default animation codec
    * Deprecated support for mencoder in animation
    * Boxplot Zorder Keyword Argument
    * Filled + and x markers
    * rcount andccount for plot_surface()
    * Streamplot Zorder Keyword Argument Changes
    * Extension to matplotlib.backend_bases.GraphicsContextBase
9.1 New in matplotlib 2.0

Note: matplotlib 2.0 supports Python 2.7, and 3.4+

9.1.1 Default style changes

The major changes in v2.0 are related to overhauling the default styles.

Changes to the default style

The most important changes in matplotlib 2.0 are the changes to the default style.
While it is impossible to select the best default for all cases, these are designed to work well in the most common cases.

A ‘classic’ style sheet is provided so reverting to the 1.x default values is a single line of python

\[
\text{mpl.style.use('classic')}
\]

See The matplotlibrc file for details about how to persistently and selectively revert many of these changes.

Table of Contents

- Colors, color cycles, and color maps
  - Colors in default property cycle
  - Colormap
  - Interactive figures
  - Grid lines
- Figure size, font size, and screen dpi
- Plotting functions
  - scatter
  - plot
  - errorbar
  - boxplot
  - fill_between and fill_betweenx
– Patch edges and color
– bar and barh

• Hatching

• Fonts
  – Normal text
  – Math text

• Legends

• Image
  – Interpolation
  – Colormapping pipeline
  – Shading

• Plot layout
  – Auto limits
  – Z-order
  – Ticks
  – Tick label formatting

• mplot3d

**Colors, color cycles, and color maps**

**Colors in default property cycle**

The colors in the default property cycle have been changed from ['b', 'g', 'r', 'c', 'm', 'y', 'k'] to the category10 color palette used by Vega and d3 originally developed at Tableau.
In addition to changing the colors, an additional method to specify colors was added. Previously, the default colors were the single character short-hand notations for red, green, blue, cyan, magenta, yellow, and black. This made them easy to type and usable in the abbreviated style string in plot, however the new default colors are only specified via hex values. To access these colors outside of the property cycling the notation for colors 'CN', where N takes values 0-9, was added to denote the first 10 colors in mpl.rcParams['axes.prop_cycle']. See Specifying Colors for more details.

To restore the old color cycle use

```python
from cycler import cycler
mpl.rcParams['axes.prop_cycle'] = cycler(color='bgrcmyk')
```

or set

```python
axes.prop_cycle : cycler('color', 'bgrcmyk')
```

in your matplotlibrc file.

**Colormap**

The new default color map used by `matplotlib.cm.ScalarMappable` instances is 'viridis' (aka option D).
For an introduction to color theory and how ‘viridis’ was generated watch Nathaniel Smith and Stéfan van der Walt’s talk from SciPy2015. See here for many more details about the other alternatives and the tools used to create the color map. For details on all of the color maps available in matplotlib see *Choosing Colormaps*.

The previous default can be restored using

```python
mpl.rcParams['image.cmap'] = 'jet'
```

or setting

```python
image.cmap : 'jet'
```

in your `matplotlibrc` file; however this is strongly discouraged.

### Interactive figures

The default interactive figure background color has changed from grey to white, which matches the default background color used when saving.

The previous defaults can be restored by

```python
mpl.rcParams['figure.facecolor'] = '0.75'
```

or by setting

```python
figure.facecolor : '0.75'
```

in your `matplotlibrc` file.

### Grid lines

The default style of grid lines was changed from black dashed lines to thicker solid light grey lines.

---

9.1. **New in matplotlib 2.0**
The previous default can be restored by using:

```python
classic
mpl.rcParams['grid.color'] = 'k'
mpl.rcParams['grid.linestyle'] = ':'
mpl.rcParams['grid.linewidth'] = 0.5
```

or by setting:

```ini
grid.color : k       # grid color
grid.linestyle : :   # dotted
grid.linewidth : 0.5 # in points
```

in your matplotlibrc file.

**Figure size, font size, and screen dpi**

The default dpi used for on-screen display was changed from 80 dpi to 100 dpi, the same as the default dpi for saving files. Due to this change, the on-screen display is now more what-you-see-is-what-you-get for saved files. To keep the figure the same size in terms of pixels, in order to maintain approximately the same size on the screen, the default figure size was reduced from 8x6 inches to 6.4x4.8 inches. As a consequence of this the default font sizes used for the title, tick labels, and axes labels were reduced to maintain their size relative to the overall size of the figure. By default the dpi of the saved image is now the dpi of the `Figure` instance being saved.

This will have consequences if you are trying to match text in a figure directly with external text.

The previous defaults can be restored by:

```python
v2.0
mpl.rcParams['figure.figsize'] = [8.0, 6.0]
mpl.rcParams['figure.dpi'] = 80
mpl.rcParams['savefig.dpi'] = 100
mpl.rcParams['font.size'] = 12
```
mpl.rcParams['legend.fontsize'] = 'large'
mpl.rcParams['figure.titlesize'] = 'medium'

or by setting:

```python
defigure.figsize : [8.0, 6.0]
defigure.dpi : 80
defsavefig.dpi : 100
defont.size : 12.0
deflegend.fontsize : 'large'
deffigure.titlesize : 'medium'
```

In your matplotlibrc file.

In addition, the forward kwarg to set_size_inches now defaults to True to improve the interactive experience. Backend canvases that adjust the size of their bound `matplotlib.figure.Figure` must pass forward=False to avoid circular behavior. This default is not configurable.

**Plotting functions**

**scatter**

The following changes were made to the default behavior of `scatter`

- The default size of the elements in a scatter plot is now based on the rcParam `lines.markersize` so it is consistent with `plot(X, Y, 'o')`. The old value was 20, and the new value is 36 (6^2).
- scatter markers no longer have a black edge.
- if the color of the markers is not specified it will follow the property cycle, pulling from the ‘patches’ cycle on the Axes.
The classic default behavior of `scatter` can only be recovered through `mpl.style.use('classic')`. The marker size can be recovered via

```python
mpl.rcParams['lines.markersize'] = np.sqrt(20)
```

however, this will also affect the default marker size of `plot`. To recover the classic behavior on a per-call basis pass the following kwargs:

```python
classic_kwargs = {'s': 20, 'edgecolors': 'k', 'c': 'b'}
```

**plot**

The following changes were made to the default behavior of `plot`

- the default linewidth increased from 1 to 1.5
- the dash patterns associated with '--', ':', and '-.' have changed
- the dash patterns now scale with line width

The previous defaults can be restored by setting:

```python
mpl.rcParams['lines.linewidth'] = 1.0
mpl.rcParams['lines.dashed_pattern'] = [6, 6]
mpl.rcParams['lines.dashdot_pattern'] = [3, 5, 1, 5]
mpl.rcParams['lines.dotted_pattern'] = [1, 3]
mpl.rcParams['lines.scale_dashes'] = False
```

or by setting:

```python
lines.linewidth : 1.0
lines.dashed_pattern : 6, 6
lines.dashdot_pattern : 3, 5, 1, 5
```
in your matplotlibrc file.

**errorbar**

By default, caps on the ends of errorbars are not present.

This also changes the return value of `errorbar()` as the list of ‘caplines’ will be empty by default.

The previous defaults can be restored by setting:

```python
mpl.rcParams['errorbar.capsize'] = 3
```

or by setting

```python
errorbar.capsize : 3
```

in your matplotlibrc file.

**boxplot**

Previously, boxplots were composed of a mish-mash of styles that were, for better for worse, inherited from Matlab. Most of the elements were blue, but the medians were red. The fliers (outliers) were black plus-symbols (+) and the whiskers were dashed lines, which created ambiguity if the (solid and black) caps were not drawn.

For the new defaults, everything is black except for the median and mean lines (if drawn), which are set to the first two elements of the current color cycle. Also, the default flier markers are now hollow circles, which maintain the ability of the plus-symbols to overlap without obscuring data too much.
The previous defaults can be restored by setting:

```python
mpl.rcParams['boxplot.flierprops.color'] = 'k'
mpl.rcParams['boxplot.flierprops.marker'] = '+'
mpl.rcParams['boxplot.flierprops.markerfacecolor'] = 'none'
mpl.rcParams['boxplot.flierprops.markeredgecolor'] = 'k'
mpl.rcParams['boxplot.boxprops.color'] = 'b'
mpl.rcParams['boxplot.whiskerprops.color'] = 'b'
mpl.rcParams['boxplot.whiskerprops.linestyle'] = '--'
mpl.rcParams['boxplot.medianprops.color'] = 'r'
mpl.rcParams['boxplot.meanprops.color'] = 'r'
mpl.rcParams['boxplot.meanprops.marker'] = '^'
mpl.rcParams['boxplot.meanprops.markerfacecolor'] = 'r'
mpl.rcParams['boxplot.meanprops.markeredgecolor'] = 'k'
mpl.rcParams['boxplot.meanprops.markersize'] = 6
mpl.rcParams['boxplot.meanprops.linestyle'] = '--'
mpl.rcParams['boxplot.meanprops.linewidth'] = 1.0
```

or by setting:

```python
boxplot.flierprops.color: 'k'
boxplot.flierprops.marker: '+'
boxplot.flierprops.markerfacecolor: 'none'
boxplot.flierprops.markeredgecolor: 'k'
boxplot.boxprops.color: 'b'
boxplot.whiskerprops.color: 'b'
```
boxplot.whiskerprops.linestyle: '--'
boxplot.medianprops.color: 'r'
boxplot.meanprops.color: 'r'
boxplot.meanprops.marker: '^'
boxplot.meanprops.markerfacecolor: 'r'
boxplot.meanprops.markeredgecolor: 'k'
boxplot.meanprops.markersize: 6
boxplot.meanprops.linestyle: '--'
boxplot.meanprops.linewidth: 1.0

in your matplotlibrc file.

**fill_between and fill_betweenx**

*fill_between* and *fill_betweenx* both follow the patch color cycle.

![Graph showing classic and v2.0 styles]

If the facecolor is set via the *facecolors* or *color* keyword argument, then the color is not cycled.

To restore the previous behavior, explicitly pass the keyword argument *facecolors='C0'* to the method call.

**Patch edges and color**

Most artists drawn with a patch (~matplotlib.axes.Axes.bar, ~matplotlib.axes.Axes.pie, etc) no longer have a black edge by default. The default face color is now 'C0' instead of 'b'.
The previous defaults can be restored by setting:

```
mpl.rcParams['patch.force_edgecolor'] = True
mpl.rcParams['patch.facecolor'] = 'b'
```

or by setting:

```
patch.facecolor : b
patch.force_edgecolor : True
```

in your `matplotlibrc` file.

**bar and barh**

The default value of the `align` kwarg for both `bar` and `barh` is changed from 'edge' to 'center'.
To restore the previous behavior explicitly pass the keyword argument align='edge' to the method call.

### Hatching

The color of the lines in the hatch is now determined by

- If an edge color is explicitly set, use that for the hatch color
- If the edge color is not explicitly set, use rcParam['hatch.color'] which is looked up at artist creation time.

The width of the lines in a hatch pattern is now configurable by the rcParams hatch.linewidth, which defaults to 1 point. The old behavior for the line width was different depending on backend:

- PDF: 0.1 pt
- SVG: 1.0 pt
- PS: 1 px
- Agg: 1 px
The old line width behavior can not be restored across all backends simultaneously, but can be restored for a single backend by setting:

```python
cmppl.rcParams['hatch.linewidth'] = 0.1  # previous pdf hatch linewidth
mpl.rcParams['hatch.linewidth'] = 1.0  # previous svg hatch linewidth
```

The behavior of the PS and Agg backends was DPI dependent, thus:

```python
cmppl.rcParams['figure.dpi'] = dpi
mpl.rcParams['savefig.dpi'] = dpi  # or leave as default 'figure'
mpl.rcParams['hatch.linewidth'] = 1.0 / dpi  # previous ps and Agg hatch linewidth
```

There is no direct API level control of the hatch color or linewidth.

Hatching patterns are now rendered at a consistent density, regardless of DPI. Formerly, high DPI figures would be more dense than the default, and low DPI figures would be less dense. This old behavior cannot be directly restored, but the density may be increased by repeating the hatch specifier.

**Fonts**

**Normal text**

The default font has changed from “Bitstream Vera Sans” to “DejaVu Sans”. DejaVu Sans has additional international and math characters, but otherwise has the same appearance as Bitstream Vera Sans. Latin, Greek, Cyrillic, Armenian, Georgian, Hebrew, and Arabic are all supported (but right-to-left rendering is still not handled by matplotlib). In addition, DejaVu contains a sub-set of emoji symbols.
Math text

The default math font when using the built-in math rendering engine (mathtext) has changed from “Computer Modern” (i.e. LaTeX-like) to “DejaVu Sans”. This change has no effect if the TeX backend is used (i.e. `text.usetex` is True).
To revert to the old behavior set the:

```python
cmpl.rcParams['mathtext.fontset'] = 'cm'
cmpl.rcParams['mathtext.rm'] = 'serif'
```

or set:

```python
mathtext.fontset: cm
cmathtext.rm : serif
```

in your `matplotlibrc` file.

This `rcParam` is consulted when the text is drawn, not when the artist is created. Thus all `mathtext` on a given canvas will use the same fontset.
Legends

- By default, the number of points displayed in a legend is now 1.
- The default legend location is 'best', so the legend will be automatically placed in a location to minimize overlap with data.
- The legend defaults now include rounded corners, a lighter boundary, and partially transparent boundary and background.

The previous defaults can be restored by setting:

```python
mpl.rcParams['legend.fancybox'] = False
mpl.rcParams['legend.loc'] = 'upper right'
mpl.rcParams['legend.numpoints'] = 2
mpl.rcParams['legend.fontsize'] = 'large'
mpl.rcParams['legend.framealpha'] = None
mpl.rcParams['legend.scatterpoints'] = 3
mpl.rcParams['legend.edgecolor'] = 'inherit'
```

or by setting:

```plaintext
legend.fancybox : False
legend.loc : upper right
legend.numpoints : 2  # the number of points in the legend line
legend.fontsize : large
legend.framealpha : None  # opacity of legend frame
legend.scatterpoints : 3  # number of scatter points
legend.edgecolor : inherit  # legend edge color ("inherit"
                # means it uses axes.edgecolor)
```

in your matplotlibrc file.

9.1. New in matplotlib 2.0
Image

Interpolation

The default interpolation method for `imshow` is now 'nearest' and by default it resamples the data (both up and down sampling) before color mapping.

To restore the previous behavior set:

```python
mpl.rcParams['image.interpolation'] = 'bilinear
mpl.rcParams['image.resample'] = False
```

or set:

```python
image.interpolation : bilinear  # see help(imshow) for options
image.resample : False
```

in your `matplotlibrc` file.

Colormapping pipeline

Previously, the input data was normalized, then color mapped, and then resampled to the resolution required for the screen. This meant that the final resampling was being done in color space. Because the color maps are not generally linear in RGB space, colors not in the color map may appear in the final image. This bug was addressed by an almost complete overhaul of the image handling code.

The input data is now normalized, then resampled to the correct resolution (in normalized dataspace), and then color mapped to RGB space. This ensures that only colors from the color map appear in the final image. (If your viewer subsequently resamples the image, the artifact may reappear.)

The previous behavior cannot be restored.
**Shading**

- The default shading mode for light source shading, in `matplotlib.colors.LightSource.shade`, is now `overlay`. Formerly, it was `hsv`.

**Plot layout**

**Auto limits**

The previous auto-scaling behavior was to find ‘nice’ round numbers as view limits that enclosed the data limits, but this could produce bad plots if the data happened to fall on a vertical or horizontal line near the chosen ‘round number’ limit. The new default sets the view limits to 5% wider than the data range.

The size of the padding in the x and y directions is controlled by the `axes.xmargin` and `axes.ymargin` rcParams respectively. Whether the view limits should be ‘round numbers’ is controlled by the `axes.autolimit_mode` rcParam. In the original 'round_number' mode, the view limits coincide with ticks.

The previous default can be restored by using:

```python
mpl.rcParams['axes.autolimit_mode'] = 'round_numbers'
mpl.rcParams['axes.xmargin'] = 0
mpl.rcParams['axes.ymargin'] = 0
```

or setting:

```python
axes.autolimit_mode: round_numbers
axes.xmargin: 0
axes.ymargin: 0
```

in your `matplotlibrc` file.
Z-order

- Ticks and grids are now plotted above solid elements such as filled contours, but below lines. To return to the previous behavior of plotting ticks and grids above lines, set `rcParams['axes.axisbelow'] = False`.

Ticks

Direction

To reduce the collision of tick marks with data, the default ticks now point outward by default. In addition, ticks are now drawn only on the bottom and left spines to prevent a porcupine appearance, and for a cleaner separation between subplots.
To restore the previous behavior set:

```python
mpl.rcParams['xtick.direction'] = 'in'
mpl.rcParams['ytick.direction'] = 'in'
mpl.rcParams['xtick.top'] = True
mpl.rcParams['ytick.right'] = True
```

or set:

```python
xtick.top: True
xtick.direction: in

ytick.right: True
ytick.direction: in
```

in your matplotlibrc file.

### Number of ticks

The default `Locator` used for the x and y axis is `AutoLocator` which tries to find, up to some maximum number, 'nicely' spaced ticks. The locator now includes an algorithm to estimate the maximum number of ticks that will leave room for the tick labels. By default it also ensures that there are at least two ticks visible.

![Graph showing classic and v2.0 tick placements](image)

There is no way, other than using `mpl.style.use('classic')`, to restore the previous behavior as the default. On an axis-by-axis basis you may either control the existing locator via:

```python
ax.xaxis.get_major_locator().set_params(nbins=9, steps=[1, 2, 5, 10])
```

or create a new `MaxNLocator`:

```python
import matplotlib.ticker as mticker
ax.set_major_locator(mticker.MaxNLocator(nbins=9, steps=[1, 2, 5, 10]))
```
The algorithm used by `MaxNLocator` has been improved, and this may change the choice of tick locations in some cases. This also affects `AutoLocator`, which uses `MaxNLocator` internally.

For a log-scaled axis the default locator is the `LogLocator`. Previously the maximum number of ticks was set to 15, and could not be changed. Now there is a `numticks` kwarg for setting the maximum to any integer value, to the string ‘auto’, or to its default value of None which is equivalent to ‘auto’. With the ‘auto’ setting the maximum number will be no larger than 9, and will be reduced depending on the length of the axis in units of the tick font size. As in the case of the AutoLocator, the heuristic algorithm reduces the incidence of overlapping tick labels but does not prevent it.

**Tick label formatting**

**LogFormatter labeling of minor ticks**

Minor ticks on a log axis are now labeled when the axis view limits span a range less than or equal to the interval between two major ticks. See `LogFormatter` for details. The minor tick labeling is turned off when using `mpl.style.use('classic')`, but cannot be controlled independently via `rcParams`.

**ScalarFormatter tick label formatting with offsets**

With the default of `rcParams['axes.formatter.useoffset'] = True`, an offset will be used when it will save 4 or more digits. This can be controlled with the new `rcParam`, `axes.formatter.offset_threshold`. To restore the previous behavior of using an offset to save 2 or more digits, use `rcParams['axes.formatter.offset_threshold'] = 2`. 
AutoDateFormatter format strings

The default date formats are now all based on ISO format, i.e., with the slowest-moving value first. The date
formatters are configurable through the `date.autoformatter.*` rcParams.

<table>
<thead>
<tr>
<th>Threshold</th>
<th>rcParam</th>
<th>classic</th>
<th>v2.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>365 days</td>
<td>'date.autoformatter.year'</td>
<td>'%Y'</td>
<td>'%Y'</td>
</tr>
<tr>
<td>30 days</td>
<td>'date.autoformatter.month'</td>
<td>'%b %Y'</td>
<td>'%Y-%m'</td>
</tr>
<tr>
<td>1 day</td>
<td>'date.autoformatter.day'</td>
<td>'%b %d %Y'</td>
<td>'%Y-%m-%d'</td>
</tr>
<tr>
<td>1 hour</td>
<td>'date.autoformatter.hour'</td>
<td>'%H:%M:%S'</td>
<td>'%H:%M'</td>
</tr>
<tr>
<td>1 minute</td>
<td>'date.autoformatter.minute'</td>
<td>'%H:%M:%S.%f'</td>
<td>'%H:%M:%S'</td>
</tr>
<tr>
<td>1 second</td>
<td>'date.autoformatter.second'</td>
<td>'%H:%M:%S.%f'</td>
<td>'%H:%M:%S'</td>
</tr>
<tr>
<td>1 microsecond</td>
<td>'date.autoformatter.microsecond'</td>
<td>'%H:%M:%S.%f'</td>
<td>'%H:%M:%S.%f'</td>
</tr>
</tbody>
</table>

Python’s %x and %X date formats may be of particular interest to format dates based on the current locale.

The previous default can be restored by:

```python
mpl.rcParams['date.autoformatter.year'] = '%Y'
mpl.rcParams['date.autoformatter.month'] = '%b %Y'
mpl.rcParams['date.autoformatter.day'] = '%b %d %Y'
mpl.rcParams['date.autoformatter.hour'] = '%H:%M:%S'
mpl.rcParams['date.autoformatter.minute'] = '%H:%M:%S.%f'
mpl.rcParams['date.autoformatter.second'] = '%H:%M:%S.%f'
mpl.rcParams['date.autoformatter.microsecond'] = '%H:%M:%S.%f'
```

or setting
include:

- `xtick.major.width`
- `ytick.major.width`
- `xtick.color`
- `ytick.color`
- `axes.linewidth`
- `axes.edgecolor`
- `grid.color`
- `grid.linewidth`
- `grid.linestyle`

### 9.1.2 Improved color conversion API and RGBA support

The `colors` gained a new color conversion API with full support for the alpha channel. The main public functions are `is_color_like()`, `matplotlib.colors.to_rgba()`, `matplotlib.colors.to_rgba_array()` and `to_hex()`. RGBA quadruplets are encoded in hex format as `#rrggbbaa`.

A side benefit is that the Qt options editor now allows setting the alpha channel of the artists as well.

### 9.1.3 New Configuration (rcParams)

New rcparams added
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>date.autoformatter.year</td>
<td>format string for 'year' scale dates</td>
</tr>
<tr>
<td>date.autoformatter.month</td>
<td>format string for 'month' scale dates</td>
</tr>
<tr>
<td>date.autoformatter.day</td>
<td>format string for 'day' scale dates</td>
</tr>
<tr>
<td>date.autoformatter.hour</td>
<td>format string for 'hour' scale times</td>
</tr>
<tr>
<td>date.autoformatter.minute</td>
<td>format string for 'minute' scale times</td>
</tr>
<tr>
<td>date.autoformatter.second</td>
<td>format string for 'second' scale times</td>
</tr>
<tr>
<td>date.autoformatter.microsecond</td>
<td>format string for 'microsecond' scale times</td>
</tr>
<tr>
<td>scatter.marker</td>
<td>default marker for scatter plot</td>
</tr>
<tr>
<td>svg.hashsalt</td>
<td>see note</td>
</tr>
<tr>
<td>xtick.top, xtick.minor.top, xtick.major.top,</td>
<td>Control where major and minor ticks are</td>
</tr>
<tr>
<td>xtick.bottom, xtick.minor.bottom,</td>
<td>drawn. The global values are added with the</td>
</tr>
<tr>
<td>xtick.major.bottom,ytick.left,</td>
<td>corresponding major/minor values.</td>
</tr>
<tr>
<td>ytick.minor.left,ytick.major.left</td>
<td></td>
</tr>
<tr>
<td>ytick.right,ytick.minor.right,</td>
<td></td>
</tr>
<tr>
<td>ytick.major.right</td>
<td></td>
</tr>
<tr>
<td>hist.bins</td>
<td>The default number of bins to use in hist.</td>
</tr>
<tr>
<td></td>
<td>This can be an int, a list of floats, or</td>
</tr>
<tr>
<td></td>
<td>'auto' if numpy &gt;= 1.11 is installed.</td>
</tr>
<tr>
<td>lines.scale_dashes</td>
<td>Whether the line dash patterns should scale</td>
</tr>
<tr>
<td></td>
<td>with linewidth.</td>
</tr>
<tr>
<td>axes.formatter.offset_threshold</td>
<td>Minimum number of digits saved in tick</td>
</tr>
<tr>
<td></td>
<td>labels that triggers using an offset.</td>
</tr>
</tbody>
</table>

**Added svg.hashsalt key to rcParams**

If `svg.hashsalt` is `None` (which it is by default), the svg backend uses `uuid4` to generate the hash salt. If it is not `None`, it must be a string that is used as the hash salt instead of `uuid4`. This allows for deterministic SVG output.

**Removed the svg.image_noscale rcParam**

As a result of the extensive changes to image handling, the `svg.image_noscale` rcParam has been removed. The same functionality may be achieved by setting `interpolation='none'` on individual images or globally using the `image.interpolation` rcParam.

### 9.1.4 Qualitative colormaps

ColorBrewer’s “qualitative” colormaps (“Accent”, “Dark2”, “Paired”, “Pastel1”, “Pastel2”, “Set1”, “Set2”, “Set3”) were intended for discrete categorical data, with no implication of value, and therefore have been converted to `ListedColormap` instead of `LinearSegmentedColormap`, so the colors will no longer be interpolated and they can be used for choropleths, labeled image features, etc.
9.1.5 Axis offset label now responds to labelcolor

Axis offset labels are now colored the same as axis tick markers when labelcolor is altered.

9.1.6 Improved offset text choice

The default offset-text choice was changed to only use significant digits that are common to all ticks (e.g. 1231..1239 -> 1230, instead of 1231), except when they straddle a relatively large multiple of a power of ten, in which case that multiple is chosen (e.g. 1999..2001->2000).

9.1.7 Style parameter blacklist

In order to prevent unexpected consequences from using a style, style files are no longer able to set parameters that affect things unrelated to style. These parameters include:

- interactive
- backend
- backend.qt4
- webagg.port
- webagg.port_retries
- webagg.open_in_browser
- backend_fallback
- toolbar
- timezone
- datapath
- figure.max_open_warning
- savefig.directory
- tk.window_focus
- docstring.hardcopy

9.1.8 Change in default font

The default font used by matplotlib in text has been changed to DejaVu Sans and DejaVu Serif for the sans-serif and serif families, respectively. The DejaVu font family is based on the previous matplotlib default –Bitstream Vera– but includes a much wider range of characters.

The default mathtext font has been changed from Computer Modern to the DejaVu family to maintain consistency with regular text. Two new options for the mathtext.fontset configuration parameter have been added: dejavusans (default) and dejavuserif. Both of these options use DejaVu glyphs whenever possible and fall back to STIX symbols when a glyph is not found in DejaVu. To return to the previous behavior, set the rcParam mathtext.fontset to cm.

9.1.9 Faster text rendering

Rendering text in the Agg backend is now less fuzzy and about 20% faster to draw.

9.1.10 Improvements for the Qt figure options editor

Various usability improvements were implemented for the Qt figure options editor, among which:

- Line style entries are now sorted without duplicates.
- The colormap and normalization limits can now be set for images.
- Line edits for floating values now display only as many digits as necessary to avoid precision loss. An important bug was also fixed regarding input validation using Qt5 and a locale where the decimal separator is ",".
• The axes selector now uses shorter, more user-friendly names for axes, and does not crash if there are no axes.

• Line and image entries using the default labels (_lineX", ",_imageX") are now sorted numerically even when there are more than 10 entries.

9.1.11 Improved image support

Prior to version 2.0, matplotlib resampled images by first applying the color map and then resizing the result. Since the resampling was performed on the colored image, this introduced colors in the output image that didn’t actually exist in the color map. Now, images are resampled first (and entirely in floating-point, if the input image is floating-point), and then the color map is applied.

In order to make this important change, the image handling code was almost entirely rewritten. As a side effect, image resampling uses less memory and fewer datatype conversions than before.

The experimental private feature where one could “skew” an image by setting the private member _image_skew_coordinate has been removed. Instead, images will obey the transform of the axes on which they are drawn.

Non-linear scales on image plots

imshow() now draws data at the requested points in data space after the application of non-linear scales.

The image on the left demonstrates the new, correct behavior. The old behavior can be recreated using pcolormesh() as demonstrated on the right.

This can be understood by analogy to plotting a histogram with linearly spaced bins with a logarithmic x-axis. Equal sized bins will be displayed as wider for small x and narrower for large x.

9.1. New in matplotlib 2.0
9.1.12 Support for HiDPI (Retina) displays in the NbAgg and WebAgg backends

The NbAgg and WebAgg backends will now use the full resolution of your high-pixel-density display.

9.1.13 Change in the default animation codec

The default animation codec has been changed from mpeg4 to h264, which is more efficient. It can be set via the animation.codec rcParam.

9.1.14 Deprecated support for mencoder in animation

The use of mencoder for writing video files with mpl is problematic; switching to ffmpeg is strongly advised. All support for mencoder will be removed in version 2.2.

9.1.15 Boxplot Zorder Keyword Argument

The zorder parameter now exists for boxplot(). This allows the zorder of a boxplot to be set in the plotting function call.

```python
boxplot(np.arange(10), zorder=10)
```

9.1.16 Filled + and x markers

New fillable plus and x markers have been added. See the markers module and marker reference examples.

9.1.17 rcount and ccount for plot_surface()

As of v2.0, mplot3d’s plot_surface() now accepts rcount and ccount arguments for controlling the sampling of the input data for plotting. These arguments specify the maximum number of evenly spaced samples to take from the input data. These arguments are also the new default sampling method for the function, and is considered a style change.

The old rstride and cstride arguments, which specified the size of the evenly spaced samples, become the default when ‘classic’ mode is invoked, and are still available for use. There are no plans for deprecating these arguments.

9.1.18 Streamplot Zorder Keyword Argument Changes

The zorder parameter for streamplot() now has default value of None instead of 2. If None is given as zorder, streamplot() has a default zorder of matplotlib.lines.Line2D.zorder.
9.1.19 Extension to `matplotlib.backend_bases.GraphicsContextBase`

To support standardizing hatch behavior across the backends we ship the `matplotlib.backend_bases.GraphicsContextBase.get_hatch_color` method as added to `matplotlib.backend_bases.GraphicsContextBase`. This is only used during the render process in the backends we ship so will not break any third-party backends.

If you maintain a third-party backend which extends `GraphicsContextBase` this method is now available to you and should be used to color hatch patterns.

9.2 Previous Whats New

9.2.1 New in matplotlib 0.98.4

Table of Contents

- New in matplotlib 0.98.4
  - Legend enhancements
  - Fancy annotations and arrows
  - Native OS X backend
  - psd amplitude scaling
  - Fill between
  - Lots more

It’s been four months since the last matplotlib release, and there are a lot of new features and bug-fixes.

Thanks to Charlie Moad for testing and preparing the source release, including binaries for OS X and Windows for python 2.4 and 2.5 (2.6 and 3.0 will not be available until numpy is available on those releases). Thanks to the many developers who contributed to this release, with contributions from Jae-Joon Lee, Michael Droettboom, Ryan May, Eric Firing, Manuel Metz, Jouni K. Seppänen, Jeff Whitaker, Darren Dale, David Kaplan, Michiel de Hoon and many others who submitted patches.

Legend enhancements

Jae-Joon has rewritten the legend class, and added support for multiple columns and rows, as well as fancy box drawing. See `legend()` and `matplotlib.legend.Legend`.
Fancy annotations and arrows

Jae-Joon has added lots of support to annotations for drawing fancy boxes and connectors in annotations. See `annotate()` and `BoxStyle`, `ArrowStyle`, and `ConnectionStyle`. 
Native OS X backend

Michiel de Hoon has provided a native Mac OSX backend that is almost completely implemented in C. The backend can therefore use Quartz directly and, depending on the application, can be orders of magnitude faster than the existing backends. In addition, no third-party libraries are needed other than Python and NumPy. The backend is interactive from the usual terminal application on Mac using regular Python. It hasn’t been tested with ipython yet, but in principle it should to work there as well. Set ‘backend : macosx’ in your matplotlibrc file, or run your script with:

```bash
> python myfile.py -dmacosx
```
psd amplitude scaling

Ryan May did a lot of work to rationalize the amplitude scaling of \texttt{psd()} and friends. See \texttt{pylab_examples example code: psd_demo2.py} and \texttt{pylab_examples example code: psd_demo3.py}. The changes should increase MATLAB compatibility and increase scaling options.

Fill between

Added a \texttt{fill_between()} function to make it easier to do shaded region plots in the presence of masked data. You can pass an \texttt{x} array and a \texttt{ylower} and \texttt{yupper} array to fill between, and an optional \texttt{where} argument which is a logical mask where you want to do the filling.

![Fill between example](image)

Lots more

Here are the 0.98.4 notes from the CHANGELOG:

- Added mdehoon’s native macosx backend from sf patch 2179017 - JDH
- Removed the prints \texttt{in} the \texttt{set_*style} commands. Return the \texttt{list} of pretty-printed strings instead - JDH
- Some of the changes Michael made to improve the output of the
property tables in the rest docs broke of made difficult to use some of the interactive doc helpers, e.g., setp and getp. Having all the rest markup in the ipython shell also confused the docstrings. I added a new rc param docstring.harcopy, to format the docstrings differently for hardcopy and other use. The ArtistInspector could use a little refactoring now since there is duplication of effort between the rest out put and the non-rest output - JDH

Updated spectral methods (psd, csd, etc.) to scale one-sided densities by a factor of 2 and, optionally, scale all densities by the sampling frequency. This gives better MATLAB compatibility. -RM

Fixed alignment of ticks in colorbars. -MGD

drop the deprecated "new" keyword of np.histogram() for numpy 1.2 or later. -JJL

Fixed a bug in svg backend that new_figure_manager() ignores keywords arguments such as figsize, etc. -JJL

Fixed a bug that the handlelength of the new legend class set too short when numpoints=1 -JJL

Added support for data with units (e.g., dates) to Axes.fill_between. -RM

Added fancybox keyword to legend. Also applied some changes for better look, including baseline adjustment of the multiline texts so that it is center aligned. -JJL

The transmuter classes in the patches.py are reorganized as subclasses of the Style classes. A few more box and arrow styles are added. -JJL

Fixed a bug in the new legend class that didn't allowed a tuple of coordinate values as loc. -JJL

Improve checks for external dependencies, using subprocess (instead of deprecated popen*) and distutils (for version checking) - DSD

Reimplementation of the legend which supports baseline alignment, multi-column, and expand mode. - JJL

Fixed histogram autoscaling bug when bins or range are given explicitly (fixes Debian bug 503148) - MM

Added rcParam axes.unicode_minus which allows plain hyphen for minus when False - JDH

Added scatterpoints support in Legend. patch by Erik Tollerud - JJL
Fix crash in log ticking. - MGD

Added static helper method BrokenHBarCollection.span_where and Axes/pyplot method fill_between. See examples/pylab/fill_between.py - JDH

Add x_isdata and y_isdata attributes to Artist instances, and use them to determine whether either or both coordinates are used when updating dataLim. This is used to fix autoscaling problems that had been triggered by axhline, axhspan, axvline, axvspan. - EF

Update the psd(), csd(), cohere(), and specgram() methods of Axes and the csd() cohere(), and specgram() functions in mlab to be in sync with the changes to psd(). In fact, under the hood, these all call the same core to do computations. - RM

Add 'pad_to' and 'sides' parameters to mlab.psd() to allow controlling of zero padding and returning of negative frequency components, respectively. These are added in a way that does not change the API. - RM

Fix handling of c kwarg by scatter; generalize is_string_like to accept numpy and numpy.ma string array scalars. - RM and EF

Fix a possible EINTR problem in dviread, which might help when saving pdf files from the qt backend. - JKS

Fix bug with zoom to rectangle and twin axes - MGD

Added Jae Joon's fancy arrow, box and annotation enhancements -- see examples/pylab_examples/annotation_demo2.py

Autoscaling is now supported with shared axes - EF

Fixed exception in dviread that happened with Minion - JKS

set_xlim, ylim now return a copy of the viewlim array to avoid modify inplace surprises

Added image thumbnail generating function
matplotlib.image.thumbnail. See examples/misc/image_thumbnail.py - JDH

Applied scatleg patch based on ideas and work by Erik Tollerud and Jae-Joon Lee. - MM

Fixed bug in pdf backend: if you pass a file object for output instead of a filename, e.g., in a wep app, we now flush the object at the end. - JKS

Add path simplification support to paths with gaps. - EF
Fix problem with AFM files that don't specify the font's full name or family name. - JKS

Added 'scilimits' kwarg to Axes.ticklabel_format() method, for easy access to the set_powerlimits method of the major ScalarFormatter. - EF

Experimental new kwarg borderpad to replace pad in legend, based on suggestion by Jae-Joon Lee. - EF

Allow spy to ignore zero values in sparse arrays, based on patch by Tony Yu. Also fixed plot to handle empty data arrays, and fixed handling of markers in figlegend. - EF

Introduce drawstyles for lines. Transparently split linestyles like 'steps--' into drawstyle 'steps' and linestyle '--'. Legends always use drawstyle 'default'. - MM

Fixed quiver and quiverkey bugs (failure to scale properly when resizing) and added additional methods for determining the arrow angles - EF

Fix polar interpolation to handle negative values of theta - MGD

Reorganized cbook and mlab methods related to numerical calculations that have little to do with the goals of those two modules into a separate module numerical_methods.py Also, added ability to select points and stop point selection with keyboard in ginput and manual contour labeling code. Finally, fixed contour labeling bug. - DMK

Fix backtick in Postscript output. - MGD

[2089958] Path simplification for vector output backends
Leverage the simplification code exposed through path_to_polygons to simplify certain well-behaved paths in the vector backends (PDF, PS and SVG). "path.simplify" must be set to True in matplotlibrc for this to work. - MGD

Add "filled" kwarg to Path.intersects_path and Path.intersects_bbox. - MGD

Changed full arrows slightly to avoid an xpdf rendering problem reported by Friedrich Hagedorn. - JKS

Fix conversion of quadratic to cubic Bezier curves in PDF and PS backends. Patch by Jae-Joon Lee. - JKS

Added 5-point star marker to plot command q- EF

Fix hatching in PS backend - MGD

Fix log with base 2 - MGD

9.2. Previous Whats New
Added support for bilinear interpolation in NonUniformImage; patch by Gregory Lielens. - EF

Added support for multiple histograms with data of different length - MM

Fix step plots with log scale - MGD

Fix masked arrays with markers in non-Agg backends - MGD

Fix clip_on kwarg so it actually works correctly - MGD

Fix locale problems in SVG backend - MGD

Fix masked arrays with markers in non-Agg backends - MGD

Fix clip_on kwarg so it actually works correctly - MGD

Fix locale problems in SVG backend - MGD

Fix masked arrays with markers in non-Agg backends - MGD

fix quiver so masked values are not plotted - JSW

improve interactive pan/zoom in qt4 backend on windows - DSD

Fix more bugs in NaN/inf handling. In particular, path simplification (which does not handle NaNs or infs) will be turned off automatically when infs or NaNs are present. Also masked arrays are now converted to arrays with NaNs for consistent handling of masks and NaNs - MGD and EF

Added support for arbitrary rasterization resolutions to the SVG backend. - MW

9.2.2 New in matplotlib 0.99

Table of Contents

- New in matplotlib 0.99
  - New documentation
  - mplot3d
  - axes grid toolkit
  - Axis spine placement

New documentation

Jae-Joon Lee has written two new guides Legend guide and Advanced Annotation. Michael Sarahan has written Image tutorial. John Hunter has written two new tutorials on working with paths and transformations: Path Tutorial and Transformations Tutorial.
mplot3d

Reinier Heeres has ported John Porter’s mplot3d over to the new matplotlib transformations framework, and it is now available as a toolkit mpl_toolkits.mplot3d (which now comes standard with all mpl installs). See mplot3d Examples and mplot3d tutorial.

axes grid toolkit

Jae-Joon Lee has added a new toolkit to ease displaying multiple images in matplotlib, as well as some support for curvilinear grids to support the world coordinate system. The toolkit is included standard with all new mpl installs. See axes_grid Examples and The Matplotlib AxesGrid Toolkit User’s Guide.
Axis spine placement

Andrew Straw has added the ability to place “axis spines” – the lines that denote the data limits – in various arbitrary locations. No longer are your axis lines constrained to be a simple rectangle around the figure – you can turn on or off left, bottom, right and top, as well as “detach” the spine to offset it away from the data. See `pylab_examples example code: spine_placement_demo.py` and `matplotlib.spines.Spine`. 
9.2.3 New in matplotlib 1.0

**Table of Contents**

- New in matplotlib 1.0
  - HTML5/Canvas backend
  - Sophisticated subplot grid layout
  - Easy pythonic subplots
  - Contour fixes and and triplot
  - multiple calls to show supported
  - mplot3d graphs can be embedded in arbitrary axes
  - tick_params
  - Lots of performance and feature enhancements
  - Much improved software carpentry
  - Bugfix marathon
HTML5/Canvas backend

Simon Ratcliffe and Ludwig Schwardt have released an HTML5/Canvas backend for matplotlib. The backend is almost feature complete, and they have done a lot of work comparing their html5 rendered images with our core renderer Agg. The backend features client/server interactive navigation of matplotlib figures in an html5 compliant browser.

Sophisticated subplot grid layout

Jae-Joon Lee has written gridspec, a new module for doing complex subplot layouts, featuring row and column spans and more. See Customizing Location of Subplot Using GridSpec for a tutorial overview.

Easy pythonic subplots

Fernando Perez got tired of all the boilerplate code needed to create a figure and multiple subplots when using the matplotlib API, and wrote a subplots() helper function. Basic usage allows you to create the figure and an array of subplots with numpy indexing (starts with 0). e.g.:

```python
fig, axarr = plt.subplots(2, 2)
axarr[0,0].plot([1,2,3])  # upper, left
```

See pylab_examples example code: subplots_demo.py for several code examples.
Contour fixes and and triplot

Ian Thomas has fixed a long-standing bug that has vexed our most talented developers for years. `contourf()` now handles interior masked regions, and the boundaries of line and filled contours coincide. Additionally, he has contributed a new module `tri` and helper function `triplot()` for creating and plotting unstructured triangular grids.
multiple calls to show supported

A long standing request is to support multiple calls to `show()`. This has been difficult because it is hard to get consistent behavior across operating systems, user interface toolkits and versions. Eric Firing has done a lot of work on rationalizing show across backends, with the desired behavior to make show raise all newly created figures and block execution until they are closed. Repeated calls to show should raise newly created figures since the last call. Eric has done a lot of testing on the user interface toolkits and versions and platforms he has access to, but it is not possible to test them all, so please report problems to the mailing list and bug tracker.

mplot3d graphs can be embedded in arbitrary axes

You can now place an mplot3d graph into an arbitrary axes location, supporting mixing of 2D and 3D graphs in the same figure, and/or multiple 3D graphs in a single figure, using the “projection” keyword argument to `add_axes` or `add_subplot`. Thanks Ben Root.
tick_params

Eric Firing wrote tick_params, a convenience method for changing the appearance of ticks and tick labels. See pyplot function `tick_params()` and associated Axes method `tick_params()`.

Lots of performance and feature enhancements

- Faster magnification of large images, and the ability to zoom in to a single pixel
- Local installs of documentation work better
- Improved “widgets” – mouse grabbing is supported
- More accurate snapping of lines to pixel boundaries
- More consistent handling of color, particularly the alpha channel, throughout the API

Much improved software carpentry

The matplotlib trunk is probably in as good a shape as it has ever been, thanks to improved software carpentry. We now have a buildbot which runs a suite of nose regression tests on every svn commit, auto-generating a set of images and comparing them against a set of known-goods, sending emails to developers on failures.
with a pixel-by-pixel image comparison. Releases and release bugfixes happen in branches, allowing active new feature development to happen in the trunk while keeping the release branches stable. Thanks to Andrew Straw, Michael Droettboom and other matplotlib developers for the heavy lifting.

Bugfix marathon

Eric Firing went on a bug fixing and closing marathon, closing over 100 bugs on the bug tracker with help from Jae-Joon Lee, Michael Droettboom, Christoph Gohlke and Michiel de Hoon.

9.2.4 New in matplotlib 1.1

Table of Contents

- New in matplotlib 1.1
  - Sankey Diagrams
  - Animation
  - Tight Layout
  - PyQt4, PySide, and IPython
  - Legend
  - mplot3d
  - Numerix support removed
  - Markers
  - Other improvements

Note: matplotlib 1.1 supports Python 2.4 to 2.7

Sankey Diagrams

Kevin Davies has extended Yannick Copin’s original Sankey example into a module (sankey) and provided new examples (api example code: sankey_demo Basics.py, api example code: sankey_demo links.py, api example code: sankey demo rankine.py).
Rankine Power Cycle: Example 8.6 from Moran and Shapiro

Animation

Ryan May has written a backend-independent framework for creating animated figures. The animation module is intended to replace the backend-specific examples formerly in the Matplotlib Examples listings. Examples using the new framework are in animation Examples; see the entrancing double pendulum which uses matplotlib.animation.Animation.save() to create the movie below.
This should be considered as a beta release of the framework; please try it and provide feedback.

**Tight Layout**

A frequent issue raised by users of matplotlib is the lack of a layout engine to nicely space out elements of the plots. While matplotlib still adheres to the philosophy of giving users complete control over the placement of plot elements, Jae-Joon Lee created the `tight_layout` module and introduced a new command `tight_layout()` to address the most common layout issues.

The usage of this functionality can be as simple as

```python
plt.tight_layout()
```

and it will adjust the spacing between subplots so that the axis labels do not overlap with neighboring subplots. A *Tight Layout guide* has been created to show how to use this new tool.
PyQT4, PySide, and IPython

Gerald Storer made the Qt4 backend compatible with PySide as well as PyQT4. At present, however, PySide does not support the PyOS_InputHook mechanism for handling gui events while waiting for text input, so it cannot be used with the new version 0.11 of IPython. Until this feature appears in PySide, IPython users should use the PyQT4 wrapper for QT4, which remains the matplotlib default.

An rcParam entry, “backend.qt4”, has been added to allow users to select PyQt4, PyQt4v2, or PySide. The latter two use the Version 2 Qt API. In most cases, users can ignore this rcParam variable; it is available to aid in testing, and to provide control for users who are embedding matplotlib in a PyQt4 or PySide app.

Legend

Jae-Joon Lee has improved plot legends. First, legends for complex plots such as stem() plots will now display correctly. Second, the ‘best’ placement of a legend has been improved in the presence of NANs.

See the Legend guide for more detailed explanation and examples.

mplot3d

In continuing the efforts to make 3D plotting in matplotlib just as easy as 2D plotting, Ben Root has made several improvements to the mplot3d module.
Matplotlib, Release 2.0.2

- `Axes3D` has been improved to bring the class towards feature-parity with regular Axes objects
- Documentation for `mplot3d` was significantly expanded
- Axis labels and orientation improved
- Most 3D plotting functions now support empty inputs
- Ticker offset display added:

![3D Plot Example]

- `contourf()` gains `zdir` and `offset` kwargs. You can now do this:
**Numerix support removed**

After more than two years of deprecation warnings, Numerix support has now been completely removed from matplotlib.

**Markers**

The list of available markers for `plot()` and `scatter()` has now been merged. While they were mostly similar, some markers existed for one function, but not the other. This merge did result in a conflict for the ‘d’ diamond marker. Now, ‘d’ will be interpreted to always mean “thin” diamond while ‘D’ will mean “regular” diamond.

Thanks to Michael Droettboom for this effort.

**Other improvements**

- Unit support for polar axes and `arrow()`
- `PolarAxes` gains getters and setters for “theta_direction”, and “theta_offset” to allow for theta to go in either the clock-wise or counter-clockwise direction and to specify where zero degrees should be placed. `set_theta_zero_location()` is an added convenience function.
Matplotlib, Release 2.0.2

- Fixed error in argument handling for tri-functions such as `tripcolor()`
- `axes.labelweight` parameter added to rcParams.
- For `imshow()`, `interpolation='nearest'` will now always perform an interpolation. A “none” option has been added to indicate no interpolation at all.
- An error in the Hammer projection has been fixed.
- `clabel` for `contour()` now accepts a callable. Thanks to Daniel Hyams for the original patch.
- Jae-Joon Lee added the HBox and VBox classes.
- Christoph Gohlke reduced memory usage in `imshow()`.
- `scatter()` now accepts empty inputs.
- The behavior for ‘symlog’ scale has been fixed, but this may result in some minor changes to existing plots. This work was refined by ssyr.
- Peter Butterworth added named figure support to `figure()`.
- Michiel de Hoon has modified the MacOSX backend to make its interactive behavior consistent with the other backends.
- Pim Schellart added a new colormap called “cubehelix”. Sameer Grover also added a colormap called “coolwarm”. See it and all other colormaps here.
- Many bug fixes and documentation improvements.

9.2.5 New in matplotlib 1.2

Table of Contents

- New in matplotlib 1.2
  - Python 3.x support
  - PGF/TikZ backend
  - Locator interface
  - Tri-Surface Plots
  - Control the lengths of colorbar extensions
  - Figures are picklable
  - Set default bounding box in matplotlib
  - New Boxplot Functionality
  - New RC parameter functionality
  - Streamplot
  - New hist functionality
– Updated shipped dependencies
– Face-centred colors in tripcolor plots
– Hatching patterns in filled contour plots, with legends
– Known issues in the matplotlib 1.2 release

Note: matplotlib 1.2 supports Python 2.6, 2.7, and 3.1

Python 3.x support

Matplotlib 1.2 is the first version to support Python 3.x, specifically Python 3.1 and 3.2. To make this happen in a reasonable way, we also had to drop support for Python versions earlier than 2.6.

This work was done by Michael Droettboom, the Cape Town Python Users’ Group, many others and supported financially in part by the SAGE project.

The following GUI backends work under Python 3.x: Gtk3Cairo, Qt4Agg, TkAgg and MacOSX. The other GUI backends do not yet have adequate bindings for Python 3.x, but continue to work on Python 2.6 and 2.7, particularly the Qt and QtAgg backends (which have been deprecated). The non-GUI backends, such as PDF, PS and SVG, work on both Python 2.x and 3.x.

Features that depend on the Python Imaging Library, such as JPEG handling, do not work, since the version of PIL for Python 3.x is not sufficiently mature.

PGF/TikZ backend

Peter Würtz wrote a backend that allows matplotlib to export figures as drawing commands for LaTeX. These can be processed by PdfLaTeX, XeLaTeX or LuaLaTeX using the PGF/TikZ package. Usage examples and documentation are found in Typesetting With XeLaTeX/LuaLaTeX.
Locator interface

Philip Elson exposed the intelligence behind the tick Locator classes with a simple interface. For instance, to get no more than 5 sensible steps which span the values 10 and 19.5:

```python
>>> import matplotlib.ticker as mticker
>>> locator = mticker.MaxNLocator(nbins=5)
>>> print(locator.tick_values(10, 19.5))
```

Tri-Surface Plots

Damon McDougall added a new plotting method for the mplot3d toolkit called `plot_trisurf()`.

Control the lengths of colorbar extensions

Andrew Dawson added a new keyword argument `extendfrac` to `colorbar()` to control the length of minimum and maximum colorbar extensions.
Figures are picklable

Philip Elson added an experimental feature to make figures picklable for quick and easy short-term storage of plots. Pickle files are not designed for long term storage, are unsupported when restoring a pickle saved in another matplotlib version and are insecure when restoring a pickle from an untrusted source. Having said this, they are useful for short term storage for later modification inside matplotlib.

Set default bounding box in matplotlibrc

Two new defaults are available in the matplotlibrc configuration file: savefig.bbox, which can be set to ‘standard’ or ‘tight’, and savefig.pad_inches, which controls the bounding box padding.

New Boxplot Functionality

Users can now incorporate their own methods for computing the median and its confidence intervals into the boxplot() method. For every column of data passed to boxplot, the user can specify an accompanying median and confidence interval.
New RC parameter functionality

Matthew Emmett added a function and a context manager to help manage RC parameters: `rc_file()` and `rc_context`. To load RC parameters from a file:

```python
>>> mpl.rc_file('mpl.rc')
```

To temporarily use RC parameters:

```python
>>> with mpl.rc_context(fname='mpl.rc', rc={'text.usetex': True}):
    >>> ...
```

Streamplot

Tom Flannaghan and Tony Yu have added a new `streamplot()` function to plot the streamlines of a vector field. This has been a long-requested feature and complements the existing `quiver()` function for plotting vector fields. In addition to simply plotting the streamlines of the vector field, `streamplot()` allows users to map the colors and/or line widths of the streamlines to a separate parameter, such as the speed or local intensity of the vector field.
New hist functionality

Nic Eggert added a new stacked kwarg to `hist()` that allows creation of stacked histograms using any of the histogram types. Previously, this functionality was only available by using the `barstacked` histogram type. Now, when `stacked=True` is passed to the function, any of the histogram types can be stacked. The `barstacked` histogram type retains its previous functionality for backwards compatibility.

Updated shipped dependencies

The following dependencies that ship with matplotlib and are optionally installed alongside it have been updated:

- `pytz` 2012d
- `dateutil` 1.5 on Python 2.x, and 2.1 on Python 3.x

Face-centred colors in tripcolor plots

Ian Thomas extended `tripcolor()` to allow one color value to be specified for each triangular face rather than for each point in a triangulation.
tripcolor of Delaunay triangulation, flat shading
tripcolor of Delaunay triangulation, gouraud shading
Hatching patterns in filled contour plots, with legends

Phil Elson added support for hatching to `contourf()`, together with the ability to use a legend to identify contoured ranges.
Chapter 9. What’s new in matplotlib
Known issues in the matplotlib 1.2 release

- When using the Qt4Agg backend with IPython 0.11 or later, the save dialog will not display. This should be fixed in a future version of IPython.

9.2.6 New in matplotlib 1.2.2

Table of Contents

- New in matplotlib 1.2.2
  - Improved collections
  - Multiple images on same axes are correctly transparent

Improved collections

The individual items of a collection may now have different alpha values and be rendered correctly. This also fixes a bug where collections were always filled in the PDF backend.
Multiple images on same axes are correctly transparent

When putting multiple images onto the same axes, the background color of the axes will now show through correctly.

9.2.7 New in matplotlib 1.3

Table of Contents

- New in matplotlib 1.3
  - New in 1.3.1
  - New plotting features
  - Updated Axes3D.contour methods
  - Drawing
  - Text
  - Configuration (rcParams)
  - Backends
  - Documentation and examples
  - Infrastructure

Note: matplotlib 1.3 supports Python 2.6, 2.7, 3.2, and 3.3

New in 1.3.1

1.3.1 is a bugfix release, primarily dealing with improved setup and handling of dependencies, and correcting and enhancing the documentation.

The following changes were made in 1.3.1 since 1.3.0.

Enhancements

- Added a context manager for creating multi-page pdfs (see matplotlib.backends.backend_pdf.PdfPages).
- The WebAgg backend should now have lower latency over heterogeneous Internet connections.
Bug fixes

- Histogram plots now contain the endline.
- Fixes to the Molleweide projection.
- Handling recent fonts from Microsoft and Macintosh-style fonts with non-ascii metadata is improved.
- Hatching of fill between plots now works correctly in the PDF backend.
- Tight bounding box support now works in the PGF backend.
- Transparent figures now display correctly in the Qt4Agg backend.
- Drawing lines from one subplot to another now works.
- Unit handling on masked arrays has been improved.

Setup and dependencies

- Now works with any version of pyparsing 1.5.6 or later, without displaying hundreds of warnings.
- Now works with 64-bit versions of Ghostscript on MS-Windows.
- When installing from source into an environment without Numpy, Numpy will first be downloaded and built and then used to build matplotlib.
- Externally installed backends are now always imported using a fully-qualified path to the module.
- Works with newer version of wxPython.
- Can now build with a PyCXX installed globally on the system from source.
- Better detection of Gtk3 dependencies.

Testing

- Tests should now work in non-English locales.
- PEP8 conformance tests now report on locations of issues.

New plotting features

xkcd-style sketch plotting

To give your plots a sense of authority that they may be missing, Michael Droettboom (inspired by the work of many others in PR #1329) has added an xkcd-style sketch plotting mode. To use it, simply call `matplotlib.pyplot.xkcd()` before creating your plot. For really fine control, it is also possible to modify each artist’s sketch parameters individually with `matplotlib.artist.Artist.set_sketch_params()`.

9.2. Previous Whats New
"STOVE OWNERSHIP" FROM XKCD BY RANDALL MONROE
Updated Axes3D.contour methods

Damon McDougall updated the `tricontour()` and `tricontourf()` methods to allow 3D contour plots on arbitrary unstructured user-specified triangulations.
New eventplot plot type

Todd Jennings added a `eventplot()` function to create multiple rows or columns of identical line segments.
As part of this feature, there is a new `EventCollection` class that allows for plotting and manipulating rows or columns of identical line segments.

**Triangular grid interpolation**

Geoffroy Billotey and Ian Thomas added classes to perform interpolation within triangular grids: (`LinearTriInterpolator` and `CubicTriInterpolator`) and a utility class to find the triangles in which points lie (`TrapezoidMapTriFinder`). A helper class to perform mesh refinement and smooth contouring was also added (`UniformTriRefiner`). Finally, a class implementing some basic tools for triangular mesh improvement was added (`TriAnalyzer`).
**Baselines for stackplot**

Till Stensitzki added non-zero baselines to `stackplot()`. They may be symmetric or weighted.
Rectangular colorbar extensions

Andrew Dawson added a new keyword argument `extendrect` to `colorbar()` to optionally make colorbar extensions rectangular instead of triangular.

More robust boxplots

Paul Hobson provided a fix to the `boxplot()` method that prevent whiskers from being drawn inside the box for oddly distributed data sets.

Calling subplot() without arguments

A call to `subplot()` without any arguments now acts the same as `subplot(111)` or `subplot(1,1,1)` – it creates one axes for the whole figure. This was already the behavior for both `axes()` and `subplots()`, and now this consistency is shared with `subplot()`.

Drawing
Independent alpha values for face and edge colors

Wes Campaigne modified how Patch objects are drawn such that (for backends supporting transparency) you can set different alpha values for faces and edges, by specifying their colors in RGBA format. Note that if you set the alpha attribute for the patch object (e.g. using `set_alpha()` or the `alpha` keyword argument), that value will override the alpha components set in both the face and edge colors.

Path effects on lines

Thanks to Jae-Joon Lee, path effects now also work on plot lines.

Easier creation of colormap and normalizer for levels with colors

Phil Elson added the `matplotlib.colors.from_levels_and_colors()` function to easily create a colormap and normalizer for representation of discrete colors for plot types such as `matplotlib.pyplot.pcolormesh()`, with a similar interface to that of `contourf()`.

Full control of the background color

Wes Campaigne and Phil Elson fixed the Agg backend such that PNGs are now saved with the correct background color when `fig.patch.get_alpha()` is not 1.

Improved bbox_inches="tight" functionality

Passing `bbox_inches="tight"` through to `plt.save()` now takes into account all artists on a figure - this was previously not the case and led to several corner cases which did not function as expected.
**Initialize a rotated rectangle**

Damon McDougall extended the `Rectangle` constructor to accept an `angle` kwarg, specifying the rotation of a rectangle in degrees.

**Text**

**Anchored text support**

The svg and pgf backends are now able to save text alignment information to their output formats. This allows to edit text elements in saved figures, using Inkscape for example, while preserving their intended position. For svg please note that you’ll have to disable the default text-to-path conversion (`mpl.rc('svg', fonttype='none')`).

**Better vertical text alignment and multi-line text**

The vertical alignment of text is now consistent across backends. You may see small differences in text placement, particularly with rotated text.

If you are using a custom backend, note that the `draw_text` renderer method is now passed the location of the baseline, not the location of the bottom of the text bounding box.

Multi-line text will now leave enough room for the height of very tall or very low text, such as superscripts and subscripts.

**Left and right side axes titles**

Andrew Dawson added the ability to add axes titles flush with the left and right sides of the top of the axes using a new keyword argument `loc` to `title()`.

**Improved manual contour plot label positioning**

Brian Mattern modified the manual contour plot label positioning code to interpolate along line segments and find the actual closest point on a contour to the requested position. Previously, the closest path vertex was used, which, in the case of straight contours was sometimes quite distant from the requested location. Much more precise label positioning is now possible.

**Configuration (rcParams)**

**Quickly find rcParams**

Phil Elson made it easier to search for rcParameters by passing a valid regular expression to `matplotlib.RcParams.find_all()`. `matplotlib.RcParams` now also has a pretty repr and str representation so that search results are printed prettily.
>>> import matplotlib
>>> print(matplotlib.rcParams.find_all('\size'))
RcParams({'font.size': 12,
         'xtick.major.size': 4,
         'xtick.minor.size': 2,
         'ytick.major.size': 4,
         'ytick.minor.size': 2})

\textbf{axes.xmargin and axes.ymargin added to rcParams}

rcParam values (axes.xmargin and axes.ymargin) were added to configure the default margins used. Previously they were hard-coded to default to 0, default value of both rcParam values is 0.

\textbf{Changes to font rcParams}

The font.* rcParams now affect only text objects created after the rcParam has been set, and will not retroactively affect already existing text objects. This brings their behavior in line with most other rcParams.

\textbf{savefig.jpeg_quality added to rcParams}

rcParam value savefig.jpeg_quality was added so that the user can configure the default quality used when a figure is written as a JPEG. The default quality is 95; previously, the default quality was 75. This change minimizes the artifacting inherent in JPEG images, particularly with images that have sharp changes in color as plots often do.

\textbf{Backends}

\textbf{WebAgg backend}

Michael Droettboom, Phil Elson and others have developed a new backend, WebAgg, to display figures in a web browser. It works with animations as well as being fully interactive.
Future versions of matplotlib will integrate this backend with the IPython notebook for a fully web browser based plotting frontend.

**Remember save directory**

Martin Spacek made the save figure dialog remember the last directory saved to. The default is configurable with the new `savefig.directory` rcParam in `matplotlibrc`.

**Documentation and examples**

**Numpydoc docstrings**

Nelle Varoquaux has started an ongoing project to convert matplotlib’s docstrings to numpydoc format. See [MEP10](#) for more information.
Example reorganization

Tony Yu has begun work reorganizing the examples into more meaningful categories. The new gallery page is the fruit of this ongoing work. See MEP12 for more information.

Examples now use subplots()

For the sake of brevity and clarity, most of the examples now use the newer subplots(), which creates a figure and one (or multiple) axes object(s) in one call. The old way involved a call to figure(), followed by one (or multiple) subplot() calls.

Infrastructure

Housecleaning

A number of features that were deprecated in 1.2 or earlier, or have not been in a working state for a long time have been removed. Highlights include removing the Qt version 3 backends, and the FltkAgg and Emf backends. See Changes in 1.3.x for a complete list.

New setup script

matplotlib 1.3 includes an entirely rewritten setup script. We now ship fewer dependencies with the tarballs and installers themselves. Notably, pytz, dateutil, pyparsing and six are no longer included with matplotlib. You can either install them manually first, or let pip install them as dependencies along with matplotlib. It is now possible to not include certain subcomponents, such as the unit test data, in the install. See setup.cfg.template for more information.

XDG base directory support

On Linux, matplotlib now uses the XDG base directory specification to find the matplotlibrc configuration file. matplotlibrc should now be kept in config/matplotlib, rather than matplotlib. If your configuration is found in the old location, it will still be used, but a warning will be displayed.

Catch opening too many figures using pyplot

Figures created through pyplot.figure are retained until they are explicitly closed. It is therefore common for new users of matplotlib to run out of memory when creating a large series of figures in a loop without closing them.

matplotlib will now display a RuntimeWarning when too many figures have been opened at once. By default, this is displayed for 20 or more figures, but the exact number may be controlled using the figure.max_open_warning rcParam.
9.2.8 New in matplotlib 1.4

Thomas A. Caswell served as the release manager for the 1.4 release.

Table of Contents

- New in matplotlib 1.4
  - New colormap
  - The nbagg backend
  - New plotting features
  - Date handling
  - Configuration (rcParams)
  - style package added
  - Backends
  - Text
  - Sphinx extensions
  - Legend and PathEffects documentation
  - Widgets
  - GAE integration

Note: matplotlib 1.4 supports Python 2.6, 2.7, 3.3, and 3.4

New colormap

In heatmaps, a green-to-red spectrum is often used to indicate intensity of activity, but this can be problematic for the red/green colorblind. A new, colorblind-friendly colormap is now available at matplotlib.cm. Wistia. This colormap maintains the red/green symbolism while achieving deuteranopic legibility through brightness variations. See here for more information.

The nbagg backend

Phil Elson added a new backend, named “nbagg”, which enables interactive figures in a live IPython notebook session. The backend makes use of the infrastructure developed for the webagg backend, which itself gives standalone server backed interactive figures in the browser, however nbagg does not require a dedicated matplotlib server as all communications are handled through the IPython Comm machinery.

As with other backends nbagg can be enabled inside the IPython notebook with:
import matplotlib
matplotlib.use('nbagg')

Once figures are created and then subsequently shown, they will placed in an interactive widget inside the notebook allowing panning and zooming in the same way as any other matplotlib backend. Because figures require a connection to the IPython notebook server for their interactivity, once the notebook is saved, each figure will be rendered as a static image - thus allowing non-interactive viewing of figures on services such as nbviewer.

New plotting features

Power-law normalization

Ben Gamari added a power-law normalization method, `PowerNorm`. This class maps a range of values to the interval [0,1] with power-law scaling with the exponent provided by the constructor’s `gamma` argument. Power law normalization can be useful for, e.g., emphasizing small populations in a histogram.

Fully customizable boxplots

Paul Hobson overhauled the `boxplot()` method such that it is now completely customizable in terms of the styles and positions of the individual artists. Under the hood, `boxplot()` relies on a new function (`boxplot_stats()`), which accepts any data structure currently compatible with `boxplot()`, and returns a list of dictionaries containing the positions for each element of the boxplots. Then a second method, `bxp()` is called to draw the boxplots based on the stats.

The `boxplot()` function can be used as before to generate boxplots from data in one step. But now the user has the flexibility to generate the statistics independently, or to modify the output of `boxplot_stats()` prior to plotting with `bxp()`.

Lastly, each artist (e.g., the box, outliers, cap, notches) can now be toggled on or off and their styles can be passed in through individual kwargs. See the examples: statistics example code: boxplot_demo.py and statistics example code: bxp_demo.py

Added a bool kwarg, `manage_xticks`, which if False disables the management of the ticks and limits on the x-axis by `bxp()`.

Support for datetime axes in 2d plots

Andrew Dawson added support for datetime axes to `contour()`, `contourf()`, `pcolormesh()` and `pcolor()`.

Support for additional spectrum types

Todd Jennings added support for new types of frequency spectrum plots: `magnitude_spectrum()`, `phase_spectrum()`, and `angle_spectrum()`, as well as corresponding functions in mlab.
He also added these spectrum types to `specgram()`, as well as adding support for linear scaling there (in addition to the existing dB scaling). Support for additional spectrum types was also added to `specgram()`.

He also increased the performance for all of these functions and plot types.

**Support for detrending and windowing 2D arrays in mlab**

Todd Jennings added support for 2D arrays in the `detrend_mean()`, `detrend_none()`, and `detrend()`, as well as adding `apply_window()` which support windowing 2D arrays.

**Support for strides in mlab**

Todd Jennings added some functions to mlab to make it easier to use numpy strides to create memory-efficient 2D arrays. This includes `stride_repeat()`, which repeats an array to create a 2D array, and `stride_windows()`, which uses a moving window to create a 2D array from a 1D array.

**Formatter for new-style formatting strings**

Added `FormatStrFormatterNewStyle` which does the same job as `FormatStrFormatter`, but accepts new-style formatting strings instead of printf-style formatting strings.

**Consistent grid sizes in streamplots**

`streamplot()` uses a base grid size of 30x30 for both `density=1` and `density=(1, 1)`. Previously a grid size of 30x30 was used for `density=1`, but a grid size of 25x25 was used for `density=(1, 1)`.

**Get a list of all tick labels (major and minor)**

Added the kwarg ‘which’ to `get_xticklabels()`, `get_yticklabels()` and `get_ticklabels()`. ‘which’ can be ‘major’, ‘minor’, or ‘both’ select which ticks to return, like `set_ticks_position()`. If ‘which’ is `None` then the old behaviour (controlled by the bool `minor`).

**Separate horizontal/vertical axes padding support in ImageGrid**

The kwarg ‘axes_pad’ to `mpl_toolkits.axes_grid1.ImageGrid` can now be a tuple if separate horizontal/vertical padding is needed. This is supposed to be very helpful when you have a labelled legend next to every subplot and you need to make some space for legend’s labels.
Support for skewed transformations

The Affine2D gained additional methods skew and skew_deg to create skewed transformations. Additionally, matplotlib internals were cleaned up to support using such transforms in Axes. This transform is important for some plot types, specifically the Skew-T used in meteorology.

Support for specifying properties of wedge and text in pie charts.

Added the kwargs ‘wedgeprops’ and ‘textprops’ to pie() to accept properties for wedge and text objects in a pie. For example, one can specify wedgeprops = {'linewidth':3} to specify the width of the borders of the wedges in the pie. For more properties that the user can specify, look at the docs for the wedge and text objects.
Fixed the direction of errorbar upper/lower limits

Larry Bradley fixed the `errorbar()` method such that the upper and lower limits (`lolims`, `uplims`, `xlolims`, `xuplims`) now point in the correct direction.

More consistent add-object API for Axes

Added the Axes method `add_image` to put image handling on a par with artists, collections, containers, lines, patches, and tables.

Violin Plots

Per Parker, Gregory Kelsie, Adam Ortiz, Kevin Chan, Geoffrey Lee, Deokjae Donald Seo, and Taesu Terry Lim added a basic implementation for violin plots. Violin plots can be used to represent the distribution of sample data. They are similar to box plots, but use a kernel density estimation function to present a smooth approximation of the data sample used. The added features are:

`violin()`: Renders a violin plot from a collection of statistics. `violin_stats()`: Produces a collection of statistics suitable for rendering a violin plot. `violinplot()`: Creates a violin plot from a set of sample data. This method makes use of `violin_stats()` to process the input data, and `violin_stats()` to do the actual rendering. Users are also free to modify or replace the output of `violin_stats()` in order to customize the violin plots to their liking.

This feature was implemented for a software engineering course at the University of Toronto, Scarborough, run in Winter 2014 by Anya Tafliovich.

More markevery options to show only a subset of markers

Rohan Walker extended the `markevery` property in `Line2D`. You can now specify a subset of markers to show with an int, slice object, numpy fancy indexing, or float. Using a float shows markers at approximately equal display-coordinate-distances along the line.

Added size related functions to specialized Collections

Added the `get_size` and `set_size` functions to control the size of elements of specialized collections (`AsteriskPolygonCollection BrokenBarHCollection CircleCollection PathCollection PolyCollection RegularPolyCollection StarPolygonCollection`).

Fixed the mouse coordinates giving the wrong theta value in Polar graph

Added code to `transform_non_affine()` to ensure that the calculated theta value was between the range of 0 and 2 * pi since the problem was that the value can become negative after applying the direction and rotation to the theta calculation.
Simple quiver plot for mplot3d toolkit

A team of students in an Engineering Large Software Systems course, taught by Prof. Anya Tafliovich at the University of Toronto, implemented a simple version of a quiver plot in 3D space for the mplot3d toolkit as one of their term project. This feature is documented in `quiver()`. The team members are: Ryan Steve D'Souza, Victor B, xbtsw, Yang Wang, David, Caradec Bisesar and Vlad Vassilovski.

polar-plot r-tick locations

Added the ability to control the angular position of the r-tick labels on a polar plot via `set_rlabel_position()`.

Date handling

n-d array support for date conversion

Andrew Dawson added support for n-d array handling to `matplotlib.dates.num2date()`, `matplotlib.dates.date2num()` and `matplotlib.dates.datestr2num()`. Support is also added to the unit conversion interfaces `matplotlib.dates.DateConverter` and `matplotlib.units.Registry`. 
Configuration (rcParams)

**savefig.transparent added**

Controls whether figures are saved with a transparent background by default. Previously savefig always defaulted to a non-transparent background.

**axes.titleweight**

Added rcParam to control the weight of the title

**axes.formatter.useoffset added**

Controls the default value of useOffset in ScalarFormatter. If True and the data range is much smaller than the data average, then an offset will be determined such that the tick labels are meaningful. If False then the full number will be formatted in all conditions.

**nbagg.transparent added**

Controls whether nbagg figures have a transparent background. nbagg.transparent is True by default.

**XDG compliance**

Matplotlib now looks for configuration files (both rcparams and style) in XDG compliant locations.

**style package added**

You can now easily switch between different styles using the new style package:

```python
>>> from matplotlib import style
>>> style.use('dark_background')
```

Subsequent plots will use updated colors, sizes, etc. To list all available styles, use:

```python
>>> print style.available
```

You can add your own custom <style name>.mplstyle files to ~/.matplotlib/stylelib or call use with a URL pointing to a file with matplotlibrc settings.

*Note that this is an experimental feature*, and the interface may change as users test out this new feature.
Backends

Qt5 backend

Martin Fitzpatrick and Tom Badran implemented a Qt5 backend. The differences in namespace locations between Qt4 and Qt5 was dealt with by shimming Qt4 to look like Qt5, thus the Qt5 implementation is the primary implementation. Backwards compatibility for Qt4 is maintained by wrapping the Qt5 implementation.

The Qt5Agg backend currently does not work with IPython’s %matplotlib magic.

The 1.4.0 release has a known bug where the toolbar is broken. This can be fixed by:

```
cd path/to/installed/matplotlib
wget https://github.com/matplotlib/matplotlib/pull/3322.diff
# unix2dos 3322.diff (if on windows to fix line endings)
patch -p2 < 3322.diff
```

Qt4 backend

Rudolf Höfler changed the appearance of the subplottool. All sliders are vertically arranged now, buttons for tight layout and reset were added. Furthermore, the subplottool is now implemented as a modal dialog. It was previously a QMainWindow, leaving the SPT open if one closed the plot window.

In the figure options dialog one can now choose to (re-)generate a simple automatic legend. Any explicitly set legend entries will be lost, but changes to the curves’ label, linestyle, et cetera will now be updated in the legend.

Interactive performance of the Qt4 backend has been dramatically improved under windows.

The mapping of key-signals from Qt to values matplotlib understands was greatly improved (For both Qt4 and Qt5).

Cairo backends

The Cairo backends are now able to use the cairocffi bindings which are more actively maintained than the pycairo bindings.

Gtk3Agg backend

The Gtk3Agg backend now works on Python 3.x, if the cairocffi bindings are installed.

PDF backend

Added context manager for saving to multi-page PDFs.
Text

Text URLs supported by SVG backend

The svg backend will now render Text objects’ url as a link in output SVGs. This allows one to make clickable text in saved figures using the url kwarg of the Text class.

Anchored sizebar font

Added the fontproperties kwarg to AnchoredSizeBar to control the font properties.

Sphinx extensions

The :context: directive in the plot_directive Sphinx extension can now accept an optional reset setting, which will cause the context to be reset. This allows more than one distinct context to be present in documentation. To enable this option, use :context: reset instead of :context: any time you want to reset the context.

Legend and PathEffects documentation

The Legend guide and Path effects guide have both been updated to better reflect the full potential of each of these powerful features.

Widgets

Span Selector

Added an option span_stays to the SpanSelector which makes the selector rectangle stay on the axes after you release the mouse.

GAE integration

Matplotlib will now run on google app engine.

9.2.9 New in matplotlib 1.5

Table of Contents

- New in matplotlib 1.5
  - Interactive OO usage
Interactive OO usage

All Artists now keep track of if their internal state has been changed but not reflected in the display (‘stale’) by a call to draw. It is thus possible to pragmatically determine if a given Figure needs to be re-drawn in an interactive session.

To facilitate interactive usage a draw_all method has been added to pyplot which will redraw all of the figures which are ‘stale’.

To make this convenient for interactive use matplotlib now registers a function either with IPython’s ‘post_execute’ event or with the displayhook in the standard python REPL to automatically call plt.draw_all just before control is returned to the REPL. This ensures that the draw command is deferred and only called once.

The upshot of this is that for interactive backends (including %matplotlib notebook) in interactive mode (with plt.ion())

```python
In [1]: import matplotlib.pyplot as plt
In [2]: fig, ax = plt.subplots()
In [3]: ln, = ax.plot([0, 1, 4, 9, 16])
```
will automatically update the plot to be green. Any subsequent modifications to the Artist objects will do likewise.

This is the first step of a larger consolidation and simplification of the pyplot internals.

**Working with labeled data like pandas DataFrames**

Plot methods which take arrays as inputs can now also work with labeled data and unpack such data. This means that the following two examples produce the same plot:

Example

```python
df = pandas.DataFrame({'var1':[1,2,3,4,5,6], 'var2':[1,2,3,4,5,6]})
plt.plot(df['var1'], df['var2'])
```

Example

```python
plt.plot('var1', 'var2', data=df)
```

This works for most plotting methods, which expect arrays/sequences as inputs. *data* can be anything which supports *__getitem__* (dict, pandas.DataFrame, h5py, ...) to access array-like values with string keys.

In addition to this, some other changes were made, which makes working with labeled data (ex pandas.Series) easier:

- For plotting methods with *label* keyword argument, one of the data inputs is designated as the label source. If the user does not supply a *label* that value object will be introspected for a label, currently by looking for a *name* attribute. If the value object does not have a *name* attribute but was specified by as a key into the *data* kwarg, then the key is used. In the above examples, this results in an implicit *label="var2"* for both cases.
- *plot()* now uses the index of a Series instead of *np.arange(len(y))* if no x argument is supplied.

**Added axes.prop_cycle key to rcParams**

This is a more generic form of the now-deprecated *axes.color_cycle* param. Now, we can cycle more than just colors, but also line styles, hatches, and just about any other artist property. Cycler notation is used for defining property cycles. Adding cyclers together will be like you are *zip()*-ing together two or more property cycles together:

```python
axes.prop_cycle: cycler('color', 'rgb') + cycler('lw', [1, 2, 3])
```

You can even multiply cyclers, which is like using *iterools.product()* on two or more property cycles. Remember to use parentheses if writing a multi-line *prop_cycle* parameter.
New Colormaps

All four of the colormaps proposed as the new default are available as 'viridis' (the new default in 2.0), 'magma', 'plasma', and 'inferno'
**Styles**

Several new styles have been added, including many styles from the Seaborn project. Additionally, in order to prep for the upcoming 2.0 style-change release, a ‘classic’ and ‘default’ style has been added. For this release, the ‘default’ and ‘classic’ styles are identical. By using them now in your scripts, you can help ensure a smooth transition during future upgrades of matplotlib, so that you can upgrade to the snazzy new defaults when you are ready!

```python
import matplotlib.style
matplotlib.style.use('classic')
```

The ‘default’ style will give you matplotlib’s latest plotting styles:

```python
matplotlib.style.use('default')
```

**Backends**

**New backend selection**

The environment variable `MPLBACKEND` can now be used to set the matplotlib backend.
wx backend has been updated

The wx backend can now be used with both wxPython classic and Phoenix.

wxPython classic has to be at least version 2.8.12 and works on Python 2.x. As of May 2015 no official release of wxPython Phoenix is available but a current snapshot will work on Python 2.7+ and 3.4+.

If you have multiple versions of wxPython installed, then the user code is responsible setting the wxPython version. How to do this is explained in the comment at the beginning of the example examples/user_interfaces/embedding_in_wx2.py.

Configuration (rcParams)

Some parameters have been added, others have been improved.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{x,y}axis.labelpad</td>
<td>mplot3d now respects these parameters</td>
</tr>
<tr>
<td>axes.labelpad</td>
<td>Default space between the axis and the label</td>
</tr>
<tr>
<td>errorbar.capsize</td>
<td>Default length of end caps on error bars</td>
</tr>
<tr>
<td>{x,y}tick.minor.visible</td>
<td>Default visibility of minor x/y ticks</td>
</tr>
<tr>
<td>legend.framealpha</td>
<td>Default transparency of the legend frame box</td>
</tr>
<tr>
<td>legend.facecolor</td>
<td>Default facecolor of legend frame box (or 'inherit' from axes.facecolor)</td>
</tr>
<tr>
<td>legend.edgecolor</td>
<td>Default edgecolor of legend frame box (or 'inherit' from axes.edgecolor)</td>
</tr>
<tr>
<td>figure.titlesize</td>
<td>Default font size for figure suptitles</td>
</tr>
<tr>
<td>figure.titleweight</td>
<td>Default font weight for figure suptitles</td>
</tr>
<tr>
<td>image.composite_image</td>
<td>Whether a vector graphics backend should composite several images into a single image or not when saving. Useful when needing to edit the files further Inkscape or other programs.</td>
</tr>
<tr>
<td>markers.fillstyle</td>
<td>Default fillstyle of markers. Possible values are 'full' (the default), 'left', 'right', 'bottom', 'top' and 'none'</td>
</tr>
<tr>
<td>toolbar</td>
<td>Added 'toolmanager' as a valid value, enabling the experimental ToolManager feature.</td>
</tr>
</tbody>
</table>

Widgets
**Active state of Selectors**

All selectors now implement `set_active` and `get_active` methods (also called when accessing the `active` property) to properly update and query whether they are active.

**Moved ignore, set_active, and get_active methods to base class Widget**

Pushes up duplicate methods in child class to parent class to avoid duplication of code.

**Adds enable/disable feature to MultiCursor**

A MultiCursor object can be disabled (and enabled) after it has been created without destroying the object. Example:

```
multi_cursor.active = False
```

**Improved RectangleSelector and new EllipseSelector Widget**

Adds an `interactive` keyword which enables visible handles for manipulating the shape after it has been drawn.

Adds keyboard modifiers for:

- Moving the existing shape (default key = ‘space’)
- Making the shape square (default ‘shift’)
- Make the initial point the center of the shape (default ‘control’)
- Square and center can be combined

**Allow Artists to Display Pixel Data in Cursor**

Adds `get_pixel_data` and `format_pixel_data` methods to artists which can be used to add zdata to the cursor display in the status bar. Also adds an implementation for Images.

**New plotting features**

**Auto-wrapping Text**

Added the keyword argument “wrap” to Text, which automatically breaks long lines of text when being drawn. Works for any rotated text, different modes of alignment, and for text that are either labels or titles. This breaks at the Figure, not Axes edge.
Contour plot corner masking

Ian Thomas rewrote the C++ code that calculates contours to add support for corner masking. This is controlled by a new keyword argument `corner_mask` in the functions `contour()` and `contourf()`. The previous behaviour, which is now obtained using `corner_mask=False`, was for a single masked point to completely mask out all four quads touching that point. The new behaviour, obtained using `corner_mask=True`, only masks the corners of those quads touching the point; any triangular corners comprising three unmasked points are contoured as usual. If the `corner_mask` keyword argument is not specified, the default value is taken from rcParams.
Mostly unified linestyles for Line2D, Patch and Collection

The handling of linestyles for Lines, Patches and Collections has been unified. Now they all support defining linestyles with short symbols, like "--", as well as with full names, like "dashed". Also the definition using a dash pattern ((0., [3., 3.])) is supported for all methods using Line2D, Patch or Collection.

Legend marker order

Added ability to place the label before the marker in a legend box with markerfirst keyword

Support for legend for PolyCollection and stackplot

Added a legend_handler for PolyCollection as well as a labels argument to stackplot().

Support for alternate pivots in mplot3d quiver plot

Added a pivot kwarg to quiver() that controls the pivot point around which the quiver line rotates. This also determines the placement of the arrow head along the quiver line.
Logit Scale

Added support for the ‘logit’ axis scale, a nonlinear transformation

\[ x \rightarrow \log \left( \frac{x}{1 - x} \right) \]  

(9.1)

for data between 0 and 1 excluded.

Add step kwargs to fill_between

Added step kwarg to Axes.fill_between to allow to fill between lines drawn using the ‘step’ draw style. The values of step match those of the where kwarg of Axes.step. The asymmetry of the kwarg names is not ideal, but Axes.fill_between already has a where kwarg.

This is particularly useful for plotting pre-binned histograms.
Square Plot

Implemented square plots feature as a new parameter in the axis function. When argument ‘square’ is specified, equal scaling is set, and the limits are set such that $x_{\text{max}} - x_{\text{min}} = y_{\text{max}} - y_{\text{min}}$. 
Updated figimage to take optional resize parameter

Added the ability to plot simple 2D-Array using `plt.figimage(X, resize=True)`. This is useful for plotting simple 2D-Array without the Axes or whitespacing around the image.

Updated Figure.savefig() can now use figure’s dpi

Added support to save the figure with the same dpi as the figure on the screen using `dpi='figure'`.

Example:

```python
f = plt.figure(dpi=25)  # dpi set to 25
S = plt.scatter([1,2,3],[4,5,6])
f.savefig('output.png', dpi='figure')  # output savefig dpi set to 25 (same as figure)
```

Updated Table to control edge visibility

Added the ability to toggle the visibility of lines in Tables. Functionality added to the `pyplot.table()` factory function under the keyword argument “edges”. Values can be the strings “open”, “closed”, “horizon-
“vertical” or combinations of the letters “L”, “R”, “T”, “B” which represent left, right, top, and bottom respectively.

Example:

```python
table(..., edges="open")  # No line visible
table(..., edges="closed")  # All lines visible
table(..., edges="horizontal")  # Only top and bottom lines visible
table(..., edges="LT")  # Only left and top lines visible.
```

### Zero r/cstride support in plot_wireframe

Adam Hughes added support to mplot3d’s plot_wireframe to draw only row or column line plots.

![3D plot with wireframe support](image)

### Plot bar and barh with labels

Added kwarg "tick_label" to bar and barh to support plotting bar graphs with a text label for each bar.

![Bar chart with labels](image)
Added center and frame kwargs to pie

These control where the center of the pie graph are and if the Axes frame is shown.

Fixed 3D filled contour plot polygon rendering

Certain cases of 3D filled contour plots that produce polygons with multiple holes produced improper rendering due to a loss of path information between PolyCollection and Poly3DCollection. A function set_verts_and_codes() was added to allow path information to be retained for proper rendering.

Dense colorbars are rasterized

Vector file formats (pdf, ps, svg) are efficient for many types of plot element, but for some they can yield excessive file size and even rendering artifacts, depending on the renderer used for screen display. This is a problem for colorbars that show a large number of shades, as is most commonly the case. Now, if a colorbar is showing 50 or more colors, it will be rasterized in vector backends.
**DateFormatter strftime**

strftime method will format a datetime.datetime object with the format string passed to the formatter's constructor. This method accepts datetimes with years before 1900, unlike datetime.datetime.strftime().

**Artist-level {get,set}_usetex for text**

Add {get,set}_usetex methods to Text objects which allow artist-level control of LaTeX rendering vs the internal mathtex rendering.

**ax.remove() works as expected**

As with artists added to an Axes, Axes objects can be removed from their figure via remove().

**API Consistency fix within Locators set_params() function**

set_params() function, which sets parameters within a Locator type instance, is now available to all Locator types. The implementation also prevents unsafe usage by strictly defining the parameters that a user can set.

To use, call set_params() on a Locator instance with desired arguments:

```python
loc = matplotlib.ticker.LogLocator()
# Set given attributes for loc.
loc.set_params(numticks=8, numdec=8, subs=[2.0], base=8)
# The below will error, as there is no such parameter for LogLocator
# named foo
#loc.set_params(foo='bar')
```

**Date Locators**

Date Locators (derived from DateLocator) now implement the tick_values() method. This is expected of all Locators derived from Locator.

The Date Locators can now be used easily without creating axes

```python
from datetime import datetime
from matplotlib.dates import YearLocator
t0 = datetime(2002, 10, 9, 12, 10)
tf = datetime(2005, 10, 9, 12, 15)
loc = YearLocator()
values = loc.tick_values(t0, tf)
```
OffsetBoxes now support clipping

Artists draw onto objects of type OffsetBox through DrawingArea and TextArea. The TextArea calculates the required space for the text and so the text is always within the bounds, for this nothing has changed.

However, DrawingArea acts as a parent for zero or more Artists that draw on it and may do so beyond the bounds. Now child Artists can be clipped to the bounds of the DrawingArea.

OffsetBoxes now considered by tight_layout

When tight_layout() or Figure.tight_layout() or GridSpec.tight_layout() is called, OffsetBoxes that are anchored outside the axes will not get chopped out. The OffsetBoxes will also not get overlapped by other axes in case of multiple subplots.

Per-page pdf notes in multi-page pdfs (PdfPages)

Add a new method attach_note() to the PdfPages class, allowing the attachment of simple text notes to pages in a multi-page pdf of figures. The new note is visible in the list of pdf annotations in a viewer that has this facility (Adobe Reader, OSX Preview, Skim, etc.). Per default the note itself is kept off-page to prevent it to appear in print-outs.

PdfPages.attach_note needs to be called before savefig() in order to be added to the correct figure.

Updated fignum_exists to take figure name

Added the ability to check the existence of a figure using its name instead of just the figure number. Example:

```
figure('figure')
fignum_exists('figure') #true
```

ToolManager

Federico Ariza wrote the new ToolManager that comes as replacement for NavigationToolbar2

ToolManager offers a new way of looking at the user interactions with the figures. Before we had the NavigationToolbar2 with its own tools like zoom/pan/home/save/... and also we had the shortcuts like yscale/grid/quit/.... Toolmanager relocate all those actions as Tools (located in backend_tools), and defines a way to access/trigger/reconfigure them.

The Toolbars are replaced for ToolContainers that are just GUI interfaces to trigger the tools. But don’t worry the default backends include a ToolContainer called toolbar

Note: At the moment, we release this primarily for feedback purposes and should be treated as experimental until further notice as API changes will occur. For the moment the ToolManager works only with the GTK3 and Tk backends. Make sure you use one of those. Port for the rest of the backends is coming soon.
To activate the ToolManager include the following at the top of your file

```python
>>> matplotlib.rcParams['toolbar'] = 'toolmanager'
```

**Interact with the ToolContainer**

The most important feature is the ability to easily reconfigure the ToolContainer (aka toolbar). For example, if we want to remove the “forward” button we would just do.

```python
>>> fig.canvas.manager.toolmanager.remove_tool('forward')
```

Now if you want to programmatically trigger the “home” button

```python
>>> fig.canvas.manager.toolmanager.trigger_tool('home')
```

**New Tools for ToolManager**

It is possible to add new tools to the ToolManager

A very simple tool that prints “You’re awesome” would be:

```python
from matplotlib.backend_tools import ToolBase
class AwesomeTool(ToolBase):
    def trigger(self, *args, **kwargs):
        print("You’re awesome")
```

To add this tool to ToolManager

```python
>>> fig.canvas.manager.toolmanager.add_tool('Awesome', AwesomeTool)
```

If we want to add a shortcut ("d") for the tool

```python
>>> fig.canvas.manager.toolmanager.update_keymap('Awesome', 'd')
```

To add it to the toolbar inside the group ‘foo’

```python
>>> fig.canvas.manager.toolbar.add_tool('Awesome', 'foo')
```

There is a second class of tools, “Toggleable Tools”, this are almost the same as our basic tools, just that belong to a group, and are mutually exclusive inside that group. For tools derived from ToolToggleBase there are two basic methods enable and disable that are called automatically whenever it is toggled.

A full example is located in user_interfaces example code: toolmanager.py

**cbook.is_sequence_of_strings recognizes string objects**

This is primarily how pandas stores a sequence of strings
import pandas as pd
import matplotlib.cbook as cbook

a = np.array(['a', 'b', 'c'])
print(cbook.is_sequence_of_strings(a))  # True

a = np.array(['a', 'b', 'c'], dtype=object)
print(cbook.is_sequence_of_strings(a))  # True

s = pd.Series(['a', 'b', 'c'])
print(cbook.is_sequence_of_strings(s))  # True

Previously, the last two prints returned false.

### New close-figs argument for plot directive

Matplotlib has a sphinx extension `plot_directive` that creates plots for inclusion in sphinx documents. Matplotlib 1.5 adds a new option to the plot directive - `close-figs` - that closes any previous figure windows before creating the plots. This can help avoid some surprising duplicates of plots when using `plot_directive`.

### Support for URL string arguments to imread

The `imread()` function now accepts URL strings that point to remote PNG files. This circumvents the generation of a HTTPResponse object directly.

### Display hook for animations in the IPython notebook

`Animation` instances gained a `_repr_html_` method to support inline display of animations in the notebook. The method used to display is controlled by the `animation.html` rc parameter, which currently supports values of `none` and `html5`. `none` is the default, performing no display. `html5` converts the animation to an h264 encoded video, which is embedded directly in the notebook.

Users not wishing to use the `_repr_html_` display hook can also manually call the `to_html5_video` method to get the HTML and display using IPython’s HTML display class:

```python
from IPython.display import HTML
HTML(anim.to_html5_video())
```

### Prefixed pkg-config for building

Handling of `pkg-config` has been fixed in so far as it is now possible to set it using the environment variable `PKG_CONFIG`. This is important if your toolchain is prefixed. This is done in a simplistic way as setting `CC` or `CXX` before building. An example follows.

```bash
export PKG_CONFIG=x86_64-pc-linux-gnu-pkg-config
```
GitHub stats for 2015/10/29 - 2017/05/09 (tag: v1.5.0)

These lists are automatically generated, and may be incomplete or contain duplicates.

We closed 1130 issues and merged 1003 pull requests.

The following 292 authors contributed 4014 commits.

- 4over7
- Aashil Patel
- AbdealiJK
- Acanthostega
- Adam
- Adam Williamson
- Adrian Price-Whelan
- Adrien Chardon
- Adrien F. Vincent
- Alan Bernstein
- Alberto
- alcinos
- Aleksey Bilogur
- Alex Rothberg
- Alexander Buchkovsky
- Alexis Bienvenüe
- Ali Uneri
- Alvaro Sanchez
- alvarosg
- AndersonDaniel
• Andreas Hilboll
• Andreas Mayer
• aneda
• Anthony Scopatz
• Anton Akhmerov
• Antony Lee
• Arvind
• bduick
• Ben Root
• Benedikt Daurer
• Benedikt J. Daurer
• Benjamin Berg
• Benjamin Congdon
• BHT
• Björn Dahlgren
• Bruno Zohreh
• BTWS
• buefox
• Cameron Davidson-Pilon
• Cameron Fackler
• Chen Karako
• Chris Holdgraf
• Christian Stade-Schuldt
• Christoph Deil
• Christoph Gohlke
• Christopher Holdgraf
• Cimarron Mittelsteadt
• CJ Carey
• Colin
• Conner R. Phillips
• DaCoEx
• Dan Hickstein
• Daniel C. Marcu
• Daniel Laidig
danielballan
• Danny Hermes
• DaveL17
• David A
• David Kent
• David Stansby
deenes
• Devashish Deshpande
• Diego Mora Cespedes
• Dietrich Brunn
dlmccaffrey
• Dmitry Shachnev
• Dora Fraeman
• DoriekeMG
• Drew J. Sonne
• Dylan Evans
  5. (a) Patrick Bos
• Egor Panfilov
• Elliott Sales de Andrade
• Elvis Stansvik
• endolith
• Eric Dill
• Eric Firing
• Eric Larson
• Erin Pintozzi
• Eugene Yurtsev
• Fabian-Robert Stöter
• FedeMiorelli
• Federico Ariza
• Felix Yan
• fibersnet
• Florencia Noriega
• Florian Le Bourdais
• Francoise Provencher
• Frank Yu
• Gauravjeet
• Gaute Hope
• gcallah
• Geoffrey Spear
• gepcel
• goldstarwebs
• greg-roper
• Grillard
• Guillermo Breto
• Gustavo Goretkin
• Hakan Kucukdereli
• hannah
• Hans Moritz Günther
• Hassan Kibirige
• Hastings Greer
• Heath Henley
• Heiko Oberdiek
• Henning Pohl
• Herbert Kruitbosch
• herilalaina
• Herilalaina Rakotoarison
• Ian Thomas
• Ilia Kurenkov
• Ilya Flyamer
• ImSoErgodic
• Isa Hassen
• Isaac Schwabacher
• Isaac Slavitt
• 10. Goutin
• Jaap Versteegh
• Jacob McDonald
• jacob-on-github
• Jae-Joon Lee
• James A. Bednar
• Jan Schlüter
• Jan Schulz
• Jarrod Millman
• Jason King
• Jason Zheng
• Jeffrey Hokanson @ Loki
• Jens Hedegaard Nielsen
• jli
• Johannes Wienke
• John Vandenberg
• JojoBoulix
• jonchar
• Joseph Fox-Rabinovitz
• Joseph Jon Booker
• Jouni K. Seppänen
• Juan Nunez-Iglesias
• juan.gonzalez
• Julia Sprenger
• Julian Mehne
• Julian V. Modesto
• Julien L
• Julien Lhermitte
• Julien Schueller
• Jun Tan
• Kacper Kowalik (Xarthisius)
• kalagau
• Kanchana Ranasinghe
• Kanwar245
• Keerysanth Sribaskaran
• Kenneth Ma
• Kevin Keating
• khyox
• Kjartan Myrdal
• Klara Gerlei
• klaus
• klonuo
• Kristen M. Thyng
• kshramt
• Kyle Bridgemohansingh
• Kyler Brown
• Laptop11_ASPP2016
• lboogaard
• Leo Singer
• Levi Kilcher
• lspvic
• Luis Pedro Coelho
• lzkelley
• Maarten Baert
• Magnus Nord
• mamrehn
• Manuel Jung
• Massimo Santini
• Matt Hancock
• Matthew Brett
• Matthew Newville
• Matthias Bussonnier
• Matthias Lüthi
• Maximilian Albert
• Maximilian Maahn
• Mher Kazandjian
• Michael Droettboom
• Michiel de Hoon
• Mike Henninger
• Mike Jarvis
• MinRK
• mitch
• mlub
• mobando
• muahah
• myyc
• Naoya Kanai
• Nathan Goldbaum
• Nathan Musoke
• nbrunett
• Nelle Varoquaux
• nepix32
• Nicolas P. Rougier
• Nicolas Tessore
• Nikita Kniazev
• Nils Werner
• Ninad Bhat
• OceanWolf
• Orso Meneghini
• Pankaj Pandey
• patniharshit
• Paul Ganssle
• Paul Hobson
• Paul Ivanov
• Paul Kirow
• Paul Romano
• Pavol Juhas
• Pete Huang
• Pete Peterson
• Peter Iannucci
• Peter Mortensen
• Peter Würtz
• Petr Danecek
• Phil Elson
• Phil Ruffwind
• Pierre de Buyl
• productivememberofsociety666
• Przemysław Dąbek
• Qingpeng “Q.P.” Zhang
• RAKOTOARISON Herilalaina
• Ramiro Gómez
• Randy Olson
• rebot
• Rishikesh
• rishikksh20
• Robin Dunn
• Robin Wilson
• Ronald Hartley-Davies
• Rui Lopes
• Ryan May
• RyanPan
• Salganos
• Salil Vanvari
• Samson
• Samuel St-Jean
• Sander
• scls19fr
- Scott Howard
- scott-vsi
- Sebastian Raschka
- Sebastián Vanrell
- Seraphim Alvanides
- serv-inc
- shaunwbell
- Simon Gibbons
- sindunuragarp
- Stefan Pfenninger
- Stephan Erb
- Sterling Smith
- Steven Silvester
- Steven Tilley
- Tadeo Corradi
- Terrence J. Katzenbaer
- Terrence Katzenbaer
- The Gitter Badger
- Thomas A Caswell
- Thomas Hisch
- Thomas Robitaille
- Thomas Spura
- Thorsten Liebig
- Tian Xia
- Till Stensitzki
- tmdavison
- Tobias Froehlich
- tomoemon
- Trish Gillett-Kawamoto
- Truong Pham
- Tuan
- Tuan333
GitHub issues and pull requests:

Pull Requests (1003):

- PR #8594: Missing return in _num_to_string()
- PR #8584: Add pandas to python 3.6 build
- PR #8583: Fix pandas datetime test on pandas 0.20
- PR #8566: adding keyword plotting
- PR #8567: Minor pytest parametrization
- PR #8554: added basic_units download link to units examples
- PR #8545: Add tutorials
- PR #8176: Custom error message for draw_path. issues : #8131 (bad error message from pyplot.plot)
- PR #8185: Implement blocking Qt event loop.
- PR #8346: Use some more pytest plugins: warnings & rerunfailures
- PR #8536: Update doc build.
- PR #8544: updating developer docs
- PR #8548: fixing scatter doc
- PR #8546: nested pie example
• PR #8539: Fix rectangular patches to be properly transformed on polar axes
• PR #8525: Sphinx Gallery API pages + deprecating old examples folder
• PR #8538: Update doc/thirdpartypackages/index.rst
• PR #8535: Remove use of (deprecated) is_string_like in mplot3d.
• PR #8526: Clarify docs for rcdefaults and friends.
• PR #8513: Fix autoscaling with twinx and vspans: consider axis with one pair of finite limits ONLY
• PR #8523: Update conda patch for AppVeyor build.
• PR #8522: adding backreferences_dir param
• PR #8491: Remove codecov coverage targets.
• PR #8518: Discourage omitting backend in matplotlibrc
• PR #8486: changed inherited Axes calls to super
• PR #8511: Update front page so there’s only one gallery
• PR #8510: MNT: update GH issue template [ci skip]
• PR #8478: Fixed Error: local variable ‘xdata’ referenced before assignment” in legend_handler.py
• PR #8502: Update PR template to encourage PRs off not master.
• PR #8495: Fix incorrect text line spacing.
• PR #8472: migrate examples to sphinx-gallery
• PR #8488: Build docs with oldest numpy on 2.7.
• PR #8377: Clean up unit examples
• PR #8011: Deprecate is_string_like, is_sequence_of_strings
• PR #7990: Added a workaround for Jupyter notebooks
• PR #8324: Update svg_tooltip.py
• PR #8380: Make image_comparison more pytest-y
• PR #8485: FIX markevery only accepts builtin integers, not numpy integers
• PR #8489: Fix markup in plt.subplots docstring.
• PR #8490: Clarify that Path.contains_x implicitly closes the Path.
• PR #8492: Remove useless, confusing check in hist().
• PR #7931: The font with the same weight name as the user specified weight name . . .
• PR #8256: [DOC] Clean up bar_demo2
• PR #8455: Added axes inversion to cla()
• PR #8474: Check for non-finite axis limits placed on converted_limit
• PR #8482: Modified PR Template
• PR #7572: Overhaul external process calls
• PR #8394: Unify WM_CLASS across backends
• PR #8447: Let imshow handle float128 data.
• PR #8476: Pull Request template
• PR #8450: Don’t bother disconnecting signals in TimerQt.__del__.
• PR #8468: Clarify that Image.set_data doesn’t update normalization.
• PR #8403: New Feature - PolygonSelector Widget
• PR #8157: add which kwarg to autofmt_xdate
• PR #8022: Fixed Issue #7460: Raised an error if argument to xlim is invalid
• PR #8336: Merged streamline examples
• PR #8399: Fix % formatting and Transform equality.
• PR #8319: FancyArrowPatch docstring overhaul
• PR #8452: Revert #5392
• PR #8344: Add simple ax.arrow test
• PR #8462: Add container module to API docs
• PR #8456: Migration to sphinx-gallery
• PR #8454: Finish deprecating idle_event; style cleanups to backend_bases
• PR #8326: Orthographic projection for mplot3d
• PR #8453: Manually collect lines on checkbox example
• PR #8446: Download the depsy.org badge when building the html documentation
• PR #8435: Improve hist2d docstring by inlining properties.
• PR #8376: Remove exceltools and gtktools from docs
• PR #8322: Use scalars below a certain exponent in labes of log-scales axis
• PR #8374: DOC update build dependencies
• PR #8339: Give wide code blocks a scrollbar on website
• PR #8253: Handle floating point round-off error when converting to pixels for h264 animations
• PR #8156: DOC: Add missing cmaps to perception doc (fix for #8073)
• PR #8373: [DOC] Updated the documentation
• PR #8391: DOC: Update MEP 28
• PR #8340: Refactor code duplication in matplotlib.markers
• PR #8396: DOC: Show off FuncFormatter as a decorator
• PR #8383: Merge v2.0.x into master
• PR #8372: MNT: cleanup whitespace around @_preprocess decorator
• PR #6310: Make checkbuttons with all plotted lines with correct visibility automatically
• PR #7786: Don’t reshape offsets into the correct shape.
• PR #8254: Adding headers for examples/units for MEP12/sphinx-gallery compliance
• PR #8369: Use cbook._reshape_2D in hist.
• PR #8371: DOC: Clean up the pie docstring PR
• PR #8343: Make ArrowStyle docstrings numpydoc compatible
• PR #8368: Cleanup appveyor.yml.
• PR #8334: Fix Appveyor build.
• PR #8367: symlog + minor ticks = exception
• PR #8258: DOC: Clean up equal-aspect example
• PR #8116: Simplify _reshape_2D.
• PR #8240: DOC refactored installation instruction
• PR #8363: DOC: update link to mpl-probscale
• PR #8362: Add adjustText to the list of third party packages
• PR #7691: ENH: Optional 3d bar shading
• PR #8264: Fix leaky ps
• PR #8338: Renamed all ‘mtrans’ into more common ‘mtransforms’
• PR #8331: Don’t index into __builtins__ (not supported by PyPy).
• PR #8311: DOC api’s transition to sphinx-gallery is now complete
• PR #8287: FIX: add __setstate__ function
• PR #8281: Fix testing with tests.py on Py3.6.
• PR #8149: Fix check for DISPLAY on PyQt5.
• PR #7844: Fix containment test with nonlinear transforms.
• PR #8306: DOC added titles to the rest of the 3D plots
• PR #8328: Use neutral pronoun in docs.
• PR #8295: Removes OldScalarFormatter #7956
• PR #8310: DOC shapes and collections is fully SG compatible
• PR #8304: Remove executable bit from examples and headers.
• PR #8229: MEP12 ganged example
• PR #8301: STY: fix whitespace in the tests
• PR #8248: Inkscape shell mode.
• PR #8298: Fix sphinx required version
• PR #8276: MAINT moved some maintenance and helper python scripts to tools/
• PR #8275: DOC moved changelog to the documentation
• PR #8262: TST: fail on missing baseline file
• PR #8244: BUG Ignore invisible axes in computing tight_layout
• PR #8018: Cleanup visual_tests and disable browser opening
• PR #8268: DOC moved spines examples sphinx-gallery
• PR #8239: changes in travis’s build environment
• PR #8274: Removed obsolete license.py file
• PR #8165: FIX: Remove type checking for strings in `_validate_linestyle`
• PR #8261: Set __name__ for list validators in rcsetup.
• PR #8217: Add option to rotate labels in a pie chart (#2304)
• PR #8227: Contouring 1x1 array (issue #8197)
• PR #8269: Use sys.executable -msphinx instead of sphinx-build.
• PR #8252: Memoize parse_fontconfig_pattern; speeds up test suite by ~1min.
• PR #8047: Correct theta values when drawing a non-circular arc
• PR #8245: DOC: sphinx-gallery histograms
• PR #8241: Remove image with non-free color calibration profile
• PR #7878: Update vlines example with axes wide lines.
• PR #8237: Fix pep8 violation
• PR #8224: Implement Path.intersects_bbox in C++ to speed up legend positioning.
• PR #8228: MEP12 text alignment example
• PR #8179: TST: Enable cache directories on AppVeyor.
• PR #8211: Mep12 text labels and annotations
• PR #8234: fix gitter badge
• PR #8233: changes to MEP12/sphinx-gallery compliant
• PR #8196: Issue #8141: Dash validator allowing None values in addition to floats
• PR #8154: merge fill_demo and fill_demo_features
• PR #8213: TST: skip fc-list related tests if not installed
• PR #8172: [MRG+1] [DOC] Turn ginput dostring into a numpydocstring
• PR #8173: [MRG+1] Simplify and clean multicolor_line example
• PR #8221: Early check for dot binary (from graphviz) when building the doc (fixes #8207)
• PR #8215: Mep12 showcase
• PR #8212: Mep12 ticks and spines
• PR #8219: [DOC] Plural of axis is axes
• PR #7744: Added axis limit check for non-finite values
• PR #5691: Update documentation of stem to mention StemContainer
• PR #8175: Add autoclass entry for Artist API doc.
• PR #8158: Fix layout of spectrum_demo.py
• PR #8190: add jitter link in README
• PR #8007: Clean up BoundaryNorm docstring
• PR #8178: Addresses #8177, Readme badges
• PR #8166: MAINT: mappingview check for Python 3.4
• PR #8171: DOC: Fix small typos in ‘eventplot’ docstring
• PR #8167: Fixes typos in Artist tutorial
• PR #8161: Add a code block in ‘installing’ docs.
• PR #8150: Deprecate Axes.axesPatch, Figure.figurePatch.
• PR #8148: Remove support for -dbackend argv.
• PR #8137: Regenerate the gitwash docs.
• PR #6977: Handle dvi font names as ASCII bytestrings
• PR #8066: Clean up and move text rotation example
• PR #8134: Update Slider docs and type check slidermin and slidermax.
• PR #8139: DOC: Fixed x, y, docstring in errorbar
• PR #8133: Disable network tests on AppVeyor.
• PR #8065: Clean up and move accented text example
• PR #8119: Drop None from Container.get_children().
• PR #8115: Add branch coverage; exclude _version.py from coverage.
• PR #7995: Set sticky_edges correctly for negative height bar().
• PR #8118: Deprecate matplotlib.tests.assert_str_equal.
• PR #7394: Cleanup transforms.py.
• PR #8036: Tweak coverage
• PR #8110: Mrg2.0.x
• PR #8103: Use XDG config path on FreeBSD
• PR #8026: Pytest documentation + build tweaks
• PR #8101: Named Matplotlib module in windows instructions
• PR #8099: Update examples/README.txt.
• PR #8094: Remove example of matrix of size (12, 12) and (64, 64)
• PR #8040: ENH: Stricter validation of line style rcParams (and extended accepted types for grid.
  linestyle)
• PR #8097: use plt.gca instead of plt.axes for already existing implicit axes
• PR #8096: Improve error message for image_comparison decorator.
• PR #8085: Fix PYTHONHASHSEED setup on OS X.
• PR #8086: DOC: add SOURCELINK_SUFFIX for compatibility with Sphinx 1.5
• PR #8063: Update MovieWriter dpi default
• PR #8084: Add link to scipython book
• PR #7871: Use backports.functools_lru_cache instead of functools32
• PR #8070: Switch to suppress option to True in setup.cfg.template.
• PR #4997: The url of downloading historical prices of Yahoo Finance has changed
• PR #8043: Fix pyplot.axis(ax) when ax is in other figure.
• PR #8055: Undeprecate is_scalar_or_string.
• PR #8060: Added tight_layout() to example.
• PR #7968: Fix cohere-demo
• PR #8033: Update inline comment in set_and_get.py
• PR #7985: Catch specgram warnings during tests
• PR #7965: ENH: Fixed PercentFormatter usage with latex
• PR #8014: do not ignore “closed” parameter in Poly3DCollection
• PR #7933: Cleanup: use is not instead of not ... is, etc.
• PR #7981: Clarify backports documentation
• PR #8020: Allow choosing logit scale in qt figure options.
• PR #8003: Coverage config
• PR #7974: Switch testing to pytest completely
• PR #8001: Switch to pytest-pep8.
• PR #7993: MAINT: Updated tick and category test formatting
• PR #8002: Remove pytest_pycollect_makeitem.
• PR #7925: Fix a number of Deprecated/Invalid escape sequences
• PR #7999: More ebook deprecations.
• PR #7973: Convert test decorators to pytest fixtures
• PR #7996: Simplify implementation of is_numlike & is_string_like.
• PR #7998: Display relative image paths when tests fail.
• PR #7997: Default cmap is not jet anymore...
• PR #7809: Fix for marker verts bug
• PR #7987: Add vega deprecations to tests on master
• PR #7625: Legend autopoisioning with “spiraling” lines.
• PR #7983: Remove assert_true calls from new PRs.
• PR #7980: Coding Guide Edits
• PR #7959: Allow zero sized ticks
• PR #7767: Don’t check iterable() before len().
• PR #7913: Clean up quiver docstring + add simple quiver example
• PR #7023: Add clf kwarg to plt.figure()
• PR #7857: Fix/hide some deprecations
• PR #7961: Compute glyph widths similarly in Type 42 as in Type 3
• PR #7972: MAINT cleaning up of gallery examples.
• PR #7952: MEP12 of showcase’s examples + other folders.
• PR #7904: twinx / twiny inherit autoscale behavior for shared axis
• PR #7935: Finish removing nose
• PR #7859: Fix typo in Axes3D.set_autoscalez_on.
• PR #7947: MAINT testing.nose -> testing._nose to make it explicitly private
• PR #7960: Don’t try to build for py34 on appveyor
• PR #7949: Remove sharex_foreign example, now useless.
• PR #7843: MAINT: add ability to specify recursionlimit
• PR #7941: Cleanup: remove unused variable/assignment/expression and debug comments
• PR #7944: Improve hexbin performance
• PR #7938: Fix typo in toolkits docs
• PR #7752: bugfix for wx backend: release mouse on loss of focus and before trying to recapture
• PR #7914: Fix unpickling of CallbackRegistry on Py2.
• PR #7929: Remove a dead code (font_manager.ttfdict_fnames)
• PR #7932: Convert remaining tests to pytest
• PR #7926: Stop codecov from posting messages
- PR #7892: Configure AppVeyor to fail fast
- PR #7934: Run animation smoketest in a temporary directory.
- PR #7872: Convert font/text tests to pytest
- PR #7915: Convert sphinxext tests to pytest.
- PR #7898: MAINT moved g-i-l* modules to pytest
- PR #7897: MAINT moved all remaining “f” modules to pytest
- PR #7863: Convert backend tests to use pytest
- PR #7907: BUG: Add a newline separator in fc-list call
- PR #7920: Convert preprocess tests to pytest
- PR #7887: Convert mpl toolkits tests to pytest + minor cleanup
- PR #7918: Convert test_s* files to pytest and flake8 them
- PR #7916: Convert test_[lab]* files to pytest.
- PR #7923: Fix leak of filedescriptor if fontsize cannot be set.
- PR #7900: DOC MEP12: pie/polar and color examples + style sheets fix
- PR #7818: Tripcolor.py: Remove documentation rendering error
- PR #7896: Reject floatlike strings in mcolors.to_rgba.
- PR #7830: MAINT moved _backports to cbook module
- PR #7883: Convert mlab tests to pytest
- PR #7885: MAINT moved all “d” modules to pytest.
- PR #7889: Convert remaining test_t* files to pytest.
- PR #7748: MAINT: Deterministic SVG and PDF tests
- PR #7884: MAINT moved “c” modules to pytest
- PR #7890: DOC Convert style sheet examples to MEP12
- PR #7888: Transform test updates (pytest + cleanup)
- PR #7882: MAINT pytest now exit on first failure on travis
- PR #7327: DOC MEP12 - converted lines, bars and markers to SG/MEP12 compatible
- PR #7811: Allow figure.legend to be called without arguments
- PR #7854: !B [#7852] fix for _rrule maximum recursion depth exceeded on multiprocessing usage
- PR #7861: Make radio and check buttons visible
- PR #7868: MNT: reference the proper variable in bootstrapper
- PR #7817: better input validation on fill_between
- PR #7864: Minor simplification of inset_locator_demo.
• PR #7865: FIX Preserve title case when saving through GUI (issue #7824)
• PR #7850: Allow AnchoredOffset to take a string-like location code
• PR #7845: Fixed bug with default parameters NFFT and noverlap in specgram()
• PR #7800: DOC: explain non-linear scales and imshow (closes #7661)
• PR #7639: Enh color names
• PR #7829: MAINT tests should not use relative imports
• PR #7828: MAINT added early checks for dependencies for doc building
• PR #7424: Numpy Doc Format
• PR #7821: DOC: Changes to screenshots plots.
• PR #7644: Allow scalar height for plt.bar
• PR #7838: Merge v2.x
• PR #7823: MAINT matplotlib -> Matplotlib
• PR #7833: Deprecate unused verification code.
• PR #7827: Cast stackplot input to float when required.
• PR #7834: Remove deprecated get_example_data.
• PR #7826: Remove invalid dimension checking in axes_rgb.
• PR #7831: Function wrapper for examples/api/two_scales.py
• PR #7801: Add short-circuit return to matplotlib.artist.setp if input is length 0
• PR #7740: Beautified frontpage plots and two pylab examples
• PR #7730: Fixed GraphicsContextBase linestyle getter
• PR #7747: Update qhull to 2015.2
• PR #7645: Clean up stock sample data.
• PR #7753: Clarify the uses of whiskers float parameter.
• PR #7765: TST: Clean up figure tests
• PR #7729: For make raw_input compatible with python3
• PR #7783: Raise exception if negative height or width is passed to axes()
• PR #7727: DOC: Fix invalid nose argument in testing.rst
• PR #7731: Check origin when saving image to PNG
• PR #7782: Fix some more integer type inconsistencies in Freetype code
• PR #7781: Fix integer types for font metrics in PyGlyph class
• PR #7791: Use reliable int type for mesh size in draw_quad_mesh (#7788)
• PR #7796: Only byte-swap 16-bit PNGs on little-endian (#7792)
• PR #7794: Ignore images that doc build produces
• PR #7790: Adjust markdown and text in ISSUE_TEMPLATE.md
• PR #7773: Fix more invalid escapes sequences.
• PR #7769: Remove redundant pep8 entry in .travis.yml.
• PR #7760: DOC: Correct subplot() doc
• PR #7768: Convert unicode index to long, not int, in get_char_index
• PR #7770: BUG: improve integer step selection in MaxNLocator
• PR #7766: Invalid escape sequences are deprecated in Py3.6.
• PR #7758: fix axes barh default align option document
• PR #7749: DOC: Sync keyboard shortcuts for fullscreen toggle
• PR #7757: By default, don’t include tests in binary distributions.
• PR #7762: DOC: Fix finance depr docs to point to mpl_finance
• PR #7737: Ensure that pyenv command is in a literal block
• PR #7732: Add rcsetup_api.rst, fix typo for rcsetup.cycler
• PR #7726: FIX: Clean up in the new quiverkey test; make new figs in scale tests
• PR #7620: Add warning context
• PR #7719: Add angle kwarg to quiverkey
• PR #7701: DOC: Add bug report reqs and template to contributing guide
• PR #7723: Use mplDeprecation class for all deprecations.
• PR #7676: Makes eventplot legend work
• PR #7714: TST: switch from 3.6-dev to 3.6
• PR #7713: Declare Python 3.6 support via classifier in setup.py
• PR #7693: Change majority of redirected links in docs
• PR #7705: Fixes tzname return type
• PR #7703: BF: Convert namespace path to list
• PR #7702: DOC: Add link to bokeh/colorect in colormaps.rst
• PR #7700: DOC: Add gitter to home page
• PR #7692: Corrected default values of xextent in specgram(). Fixes Bug #7666.
• PR #7698: Update INSTALL for Python 3.6
• PR #7694: Fix a few broken links in docs
• PR #7349: Add support for png_text metadata, allow to customize metadata for other backends.
• PR #7700: Decode error messages from image converters.
• PR #7677: Make new default style examples consistent
• PR #7674: Serialize comparison of multiple baseline images.
• PR #7665: FIX: Fix super call for Python 2.7
• PR #7668: Save SVG test directly to file instead of its name.
• PR #7549: Cleanup: sorted, dict iteration, array.{ndim,size}, ...
• PR #7667: FIX: Fix missing package
• PR #7651: BUG,ENH: make deprecated decorator work (and more flexibly)
• PR #7658: Avoid comparing numpy array to strings in two places
• PR #7657: Fix warning when setting markeredgecolor to a numpy array
• PR #7659: DOC: Original documentation was misleading
• PR #6780: Call _transform_vmin_vmax during SymLogNorm.__init__
• PR #7646: Improve deprecation documentation. Closes #7643
• PR #7604: Warn if different axis projection requested
• PR #7568: Deprecate unused functions in ebook.
• PR #6428: Give a better error message on missing PostScript fonts
• PR #7585: Fix a bug in TextBox where shortcut keys were not being reenabled
• PR #7628: picker may not be callable.
• PR #7464: ENH: _StringFuncParser to get numerical functions callables from strings
• PR #7622: Mrg animation merge
• PR #7618: DOC: fixed typo in mlab.py
• PR #7596: Delay fc-list warning by 5s.
• PR #7607: TST: regenerate patheffect2
• PR #7608: Don’t call np.min on generator.
• PR #7570: Correctly skip colors for nan points given to scatter
• PR #7605: Make bars stick to explicitly-specified edges.
• PR #6597: Reproducible PS/PDF output (master)
• PR #7546: Deprecate update_datalim_numerix&update_from_data.
• PR #7574: Docs edits
• PR #7538: Don’t work out packages to install if user requests information from setup.p
• PR #7577: Spelling fix: corosponding -> corresponding
• PR #7536: Rectangle patch angle attribute and patch __str__ improvements
• PR #7547: Additional cleanups
• PR #7544: Cleanups
• PR #7548: Clarify to_rgba docstring.
• PR #7476: Sticky margins
• PR #7552: Correctly extend a lognormed colorbar
• PR #7499: Improve the the marker table in markers_api documentation
• PR #7468: TST: Enable pytest-xdist
• PR #7530: MAINT: TkAgg default backend depends on tkinter
• PR #7531: double tolerance for test_png.py/pngsuite on Windows
• PR #7533: FIX chinese character are hard to deal with in latex
• PR #7525: Avoid division by zero if headlength=0 for quiver
• PR #7522: Check at least one argument is provided for plt.table
• PR #7520: Fix table.py bug
• PR #7397: Numpydoc for backends
• PR #7513: Doc: Typo in gridspec example subtitle
• PR #7494: Remove some numpy 1.6 workarounds
• PR #7500: Set hexbin default linecolor to ‘face’
• PR #7498: Fix double running of explicitly chosen tests.
• PR #7475: Remove deprecated “shading” option to pcolor.
• PR #7436: DOC: Fixed Unicode error in gallery template cache
• PR #7496: Commit to fix a broken link
• PR #6062: Add maximum streamline length property.
• PR #7470: Clarify cross correlation documentation #1835
• PR #7481: Minor cleanup of hist().
• PR #7474: FIX/API: regenerate test figure due to hatch changes
• PR #7469: TST: Added codecov
• PR #7467: TST: Fixed part of a test that got displaced in all the changes somehow
• PR #7447: Showcase example: (kind of mandatory) Mandelbrot set
• PR #7463: Added additional coverage excludes
• PR #7449: Clarify documentation of pyplot.draw
• PR #7454: Avoid temporaries when preparing step plots.
• PR #7455: Update two_scales.py example.
• PR #7456: Add pytest’s .cache to .gitignore.
• PR #7453: TST: Fixed test_log_margins test
• PR #7144: Cleanup scales
• PR #7442: Added spacer to Tk toolbar
• PR #7444: Enhance annotation_demoX examples
• PR #7439: MEP12 API examples
• PR #7416: MAINT deprecated ‘spectral’ in favor of ‘nipy_spectral’
• PR #7435: restore test that was inadvertently removed by 5901b38
• PR #7363: Add appropriate error on color size mismatch in scatter
• PR #7433: FIX: search for tkinter first in builtins
• PR #7362: Added -j shortcut for --processes=
• PR #7408: Handle nan-masked values Axes.vlines and hlines
• PR #7409: FIX: MPL should not use new tool manager unless explictely asked for. Closes #7404
• PR #7389: DOC Convert axes docstrings to numpydoc: #7205
• PR #7417: Merge from v2.x
• PR #7398: Moved python files from doc/pyplots to examples folder
• PR #7291: MEP 29: Markup text
• PR #6560: Fillbetween
• PR #7399: Clarify wspace/hspace in documentation/comments
• PR #7400: fix ReST tag
• PR #7381: Updating the readme
• PR #7384: change hardcopy.docstring to docstring.hardcopy
• PR #7386: ENH examples are now reproducible
• PR #7395: Drop code that supports numpy pre-1.6.
• PR #7385: remove unused random import
• PR #7236: ENH Improving the contribution guidelines
• PR #7370: Add example use of axes.spines.SIDE prop in matplotlibrc
• PR #7367: Warn on invalid log axis limits, per issue #7299
• PR #7360: Updated violin plot example as per suggestions in issue #7251
• PR #7357: Added notes on how to use matplotlib in pyenv
• PR #7329: DOC MEP12 - converted animation to SG/MEP12 compatible
• PR #7337: FIX symlog scale now shows negative labels.
• PR #7354: fix small error in poly_editor example
• PR #7310: TST: Make proj3d tests into real tests
• PR #7331: MEP12 improvments for statistics plots
• PR #7340: DOC: Normalize symlink target
• PR #7328: TST: Fixed rcparams test_Issue_1713 test
• PR #7303: Traceback to help fixing double-calls to mpl.use.
• PR #7346: DOC: Fix annotation position (issue #7345)
• PR #5392: BUG: arrowhead drawing code
• PR #7318: Convert a few test files to Pytest
• PR #7323: Fix #6448: set xmin/ymin even without non-zero bins in ‘step’ hist
• PR #7326: Enable coverage sending on pytest build
• PR #7321: Remove bundled virtualenv module
• PR #7290: Remove deprecated stuff schedule for removal.
• PR #7324: DOC: Boxplot color demo update
• PR #6476: Add a common example to compare style sheets
• PR #7309: MEP28: fix rst syntax for code blocks
• PR #7250: Adds docstrings to demo_curvelinear_grid.py and demo_curvelinear_grid...
• PR #4128: Code removal for post 1.5/2.1
• PR #7308: Fix travis nightly build
• PR #7282: Draft version of MEP28: Simplification of boxplots
• PR #7304: DOC: Remove duplicate documentation from last merge.
• PR #7249: add docstring to example: axisartist/demo_floating_axes.py
• PR #7296: MAINT removing docstring dedent_interpd when possible
• PR #7298: Changed Examples for Pep8 Compliance
• PR #7295: MAINT finance module is deprecated
• PR #7214: FIX: Only render single patch for scatter
• PR #7297: MAINT docstring appending doesn’t mess with rendering anymore.
• PR #6907: Filled + and x markers
• PR #7288: Style typos fixes
• PR #7277: MEP12 - added sphinx-gallery docstrings
• PR #7286: DOC: Fix for #7283 by adding a trailing underscore to misrendered URLs
• PR #7285: added some fixes to the documentation of the functions
• PR #6690: Tidying up and tweaking mplot3d examples [MEP12]
• PR #7273: Fix image watermark example where image was hidden by axes (#7265)
• PR #7276: FIX: don’t compute flier positions if not showing
• PR #7267: DOC: changed documentation for axvspan to numpydoc format
• PR #7268: DOC Numpydoc documentation for def fill()
• PR #7272: Don’t use __builtins__ (an impl. detail) in pylab.
• PR #7241: Categorical support for NumPy string arrays.
• PR #7232: DOC improved subplots’ docstring
• PR #7256: CI: skip failing test on appveyor
• PR #7255: CI: pin to qt4
• PR #7229: DOC: instructions on installing matplotlib for dev
• PR #7252: ENH: improve PySide2 loading
• PR #7245: TST: Always produce image comparison test result images
• PR #6677: Remove a copy in pcolormesh.
• PR #6814: Customize violin plot demo, see #6723
• PR #7067: DOC: OO interface in api and other examples
• PR #6790: BUG: fix C90 warning -> error in new tkagg code
• PR #7242: Add mplcursors to third-party packages.
• PR #7222: Catch IO errors when building font cache
• PR #7220: Fix innocent typo in comments
• PR #7192: DOC: switch pylab example mri_with_eeg.py to OO interface + cosmetick fixes
• PR #6583: Fix default parameters of FancyArrow
• PR #7195: remove check under linux for ~/.matplotlib
• PR #6753: Don’t warn when legend() finds no labels.
• PR #7178: Boxplot zorder kwarg
• PR #6327: Fix captions for plot directive in latex target
• PR #7188: Remove hard-coded streamplot zorder kwarg
• PR #7170: DOC updated hexbin documentation to numpydoc format.
• PR #7031: DOC Replaced documentation with numpydoc for semilogx
• PR #7029: [WIP] DOC Updated documentation of arrow function to numpy docs format.
• PR #7167: Less stringent normalization test for float128.
• PR #7169: Remove unused variable.
• PR #7066: DOC: switch to O-O interface in basic examples
• PR #7084: [DOC] Tick locators & formatters examples
• PR #7152: Showcase example: Bézier curves & SVG
• PR #7019: Check for fontproperties in figure.suptitle.
• PR #7145: Add style to api doc; fix capitalization.
• PR #7097: image_comparison decorator refactor
• PR #7096: DOC refer to plot in the scatter plot doc
• PR #7140: FIX added matplotlib.testing.nose.plugins to setupext.py
• PR #5112: OffsetImage: use dpi_cor in get_extent
• PR #7136: DOC: minor fix in development.workflow.rst
• PR #7137: DOC: improve engineering formatter example
• PR #7131: Fix branch name in “Deleting a branch on GitHub_” section
• PR #6521: Issue #6429 fix
• PR #7111: [DOC] Fix example following comments in issue #6865
• PR #7118: PR # 7038 rebased (DOC specgram() documentation now in numpy style)
• PR #7117: PR #7030 rebased
• PR #6618: Small improvements to legend’s docstring.
• PR #7102: Adding the artist data on mouse move event message
• PR #7110: [DOC] Apply comments from issue #7017
• PR #7087: [DOC] Example of user-defined linestyle (TikZ linestyle)
• PR #7108: Typos in ticker.py
• PR #7035: DOC Update semilogy docstring to numpy doc format
• PR #7033: DOC Updated plot_date to NumPy/SciPy style
• PR #7032: DOC: Updating docstring to numpy doc format for errorbar
• PR #7094: TST: Restore broken test_use14corefonts
• PR #6995: Turn off minor grids when interactively turning off major grids.
• PR #7072: [DOC] New figure for the gallery (showcase section)
• PR #7077: label_outer() should remove inner minor ticks too.
• PR #7037: DOC change axhspan to numpydoc format
• PR #7047: DOC - SpanSelector widget documentation
• PR #7049: Documentated dependencies to the doc and remove unnecessary dependencies.
• PR #7063: Tweek tol for test_hist_steplog to fix tests on appveyor
• PR #7055: FIX: testings.nose was not installed
• PR #7058: Minor animation fixes
• PR #7057: FIX: Removed financial demos that stalled because of yahoo requests
• PR #7052: Uncaught exns are fatal for PyQt5, so catch them.
• PR #7048: FIX: remove unused variable
• PR #7042: FIX: ticks filtered by Axis, not in Tick.draw
• PR #7026: Merge 2.x to master
• PR #6988: Text box widget, take over of PR5375
• PR #6957: DOC: clearing out some instances of using pylab in the docs
• PR #7012: Don’t blacklist test_usetex using pytest
• PR #7011: TST: Fixed skip_if_command_unavailable decorator problem
• PR #6918: enable previously leftout test_usetex
• PR #7006: FIX: sphinx 1.4.0 details
• PR #6900: Enh: break website screenshot banner into 4 pieces and introduce a responsive layout
• PR #6997: FIX: slow plots of pandas objects (Second try)
• PR #6792: PGF Backend: Support interpolation=’none’
• PR #6983: Catch invalid interactive switch to log scale.
• PR #6491: Don’t warn in Collections.contains if picker is not numlike.
• PR #6978: Add link to O’Reilly video course covering matplotlib
• PR #6930: BUG: PcolorImage handles non-contiguous arrays, provides data readout
• PR #6889: support for updating axis ticks for categorical data
• PR #6974: Fixed wrong expression
• PR #6730: Add Py.test testing framework support
• PR #6904: Use edgecolor rather than linewidth to control edge display.
• PR #6919: Rework MaxNLocator, eliminating infinite loop; closes #6849
• PR #6955: Add parameter checks to DayLocator initiator
• PR #5161: Proposed change to default log scale tick formatting
• PR #6875: Add keymap (default: G) to toggle minor grid.
• PR #6920: Prepare for cross-framework test suite
• PR #6944: Restore cbook.report_memory, which was deleted in d063dee.
• PR #6961: remove extra “a”
• PR #6947: Changed error message. Issue #6933
• PR #6923: Make sure nose is only imported when needed
- PR #6851: Do not restrict coverage to matplotlib module only
- PR #6938: Image interpolation selector in Qt figure options.
- PR #6787: Python3.5 dictview support
- PR #6407: adding default toggled state for toggle tools
- PR #6898: Fix read mode when loading cached AFM fonts
- PR #6892: Don’t force anncoords to fig coords upon dragging.
- PR #6895: Prevent forced alpha in figureoptions.
- PR #6877: Fix Path deepcopy signature
- PR #6822: Use travis native cache
- PR #6821: Break reference cycle Line2D <-> Line2D._lineFunc.
- PR #6879: Delete font cache in one of the configurations
- PR #6832: Fix for ylabel title in example tex_unicode_demo.py
- PR #6848: test_tinypages: pytest compatible module level setup
- PR #6881: add doi to bibtex entry for Hunter (2007)
- PR #6842: Clarify Axes.hexbin extent docstring
- PR #6861: Update ggplot URLs
- PR #6878: DOC: use venv instead of virtualenv on python 3
- PR #6837: Fix Normalize(<signed integer array>).
- PR #6874: Update bachelors_degree_by_gender example.
- PR #6867: Mark make_all_2d_testfuncs as not a test
- PR #6854: Fix for PyQt5.7 support.
- PR #6862: Change default doc image format to png and pdf
- PR #6819: Add mpl_toolkits to coveragerc.
- PR #6840: Fixed broken test_pickle.test_complete test
- PR #6841: DOC: Switch to OO code style & ensure fixed y-range in psd_demo3
- PR #6843: DOC: Fix psd_demo_complex similarly to psd_demo3
- PR #6829: Tick label rotation via set_tick_params
- PR #6799: Allow creating annotation arrows w/ default props.
- PR #6262: Properly handle UTC conversion in date2num.
- PR #6777: Raise lock timeout as actual exception
- PR #6817: DOC: Fix a few typos and formulations
- PR #6826: Clarify doc for “norm” kwarg to imshow.
PR #6807: Deprecate \{get, set\}_cursorprops.
PR #6811: Add xkcd font as one of the options
PR #6815: Rename tests in test_mlab.py
PR #6808: Don’t forget to disconnect callbacks for dragging.
PR #6803: better freetype version checking
PR #6778: Added contribute information to readme
PR #6786: 2.0 Examples fixes. See #6762
PR #6774: Appveyor: use newer conda packages and only run all tests on one platform
PR #6779: Fix tutorial pyplot scales (issue #6775)
PR #6768: Takeover #6535
PR #6763: Invalidate test cache on gs/inkscape version
PR #6765: Get more rcParams for 3d
PR #6764: Support returning polylines from to_polygons
PR #6760: DOC: clean up of demo_annotation_box.py
PR #6735: Added missing side tick rcParams
PR #6761: Fixed warnings catching and counting with warnings.catch_warnings
PR #5349: Add a Gitter chat badge to README.rst
PR #6755: PEP: fix minor formatting issues
PR #6699: Warn if MPLBACKEND is invalid.
PR #6754: Fixed error handling in ImageComparisonTest.setup_class
PR #6734: register IPython’s eventloop integration in plt.install_repl_displayhook
PR #6745: DOC: typo in broken_axis pylab example
PR #6747: Also output the actual error on svg backend tests using subprocess
PR #6744: Add workaround for failures due to newer miktex
PR #6741: Missing cleanup decorator in test_subplots.test_exceptions
PR #6736: doc: fix unescaped backslash
PR #6733: Mergev2.x to master
PR #6729: Fix crash if byte-compiled level 2
PR #6575: setup.py: Recommend installation command for pkgs
PR #6645: Fix containment and subslice optim. for steps.
PR #6619: Hide “inner” \{x,y\}labels in label_outer too.
PR #6639: Simplify get_legend_handler method
- PR #6694: Improve Line2D and MarkerStyle instantiation
- PR #6692: Remove explicit children invalidation in update_position method
- PR #6703: DOC: explain behavior of notches beyond quartiles
- PR #6707: Call gc.collect after each test only if the user asks for it
- PR #6711: Added support for mgs to Ghostscript dependency checker
- PR #6700: Don’t convert vmin, vmax to floats.
- PR #6714: fixed font_manager.is_opentype_cff_font()
- PR #6701: Colours like ‘XeYYYY’ don’t get recognised properly if X, Y’s are numbers
- PR #6512: Add computer modern font family
- PR #6383: Qt editor alpha
- PR #6381: Fix canonical name for “None” linestyle.
- PR #6689: Str Categorical Axis Support
- PR #6686: Merged _bool from axis into cbook._string_to_bool
- PR #6683: New entry in .mailmap
- PR #6520: Appveyor overhaul
- PR #6697: Fixed path caching bug in Path.unit_regular_star
- PR #6688: DOC: fix radial increase of size & OO style in polar_scatter_demo
- PR #6681: Fix #6680 (minor typo in IdentityTransform docstring)
- PR #6676: Fixed AppVeyor building script
- PR #6672: Fix example of streamplot start_points option
- PR #6601: BF: protect against locale in sphinxext text
- PR #6662: adding from_list to custom cmap tutorial
- PR #6666: Guard against too-large figures
- PR #6659: Fix image alpha
- PR #6642: Fix rectangle selector release bug
- PR #6652: Minor doc updates.
- PR #6653: DOC: Incorrect rendering of dashes
- PR #6648: adding a new color and editing an existing color in fivethirtyeight.mplstyle
- PR #6548: Fix typo.
- PR #6628: fix the swab bug to compile on solaris system
- PR #6622: colors: ensure masked array data is an ndarray
- PR #6625: DOC: Found a typo.
• PR #6614: Fix docstring for PickEvent.
• PR #6554: Update mpl_toolkits.gtktools.
• PR #6564: Cleanup for drawstyles.
• PR #6577: Fix mlab.rec_join.
• PR #6596: Added a new example to create error boxes using a PatchCollection
• PR #2370: Implement draw_markers in the cairo backend.
• PR #6599: Drop conditional import of figureoptions.
• PR #6573: Some general cleanups
• PR #6568: Add OSX to travis tests
• PR #6600: Typo: markeredgewith -> markeredgewidth
• PR #6526: ttconv: Also replace carriage return with spaces.
• PR #6530: Update make.py
• PR #6405: ToolManager/Tools adding methods to set figure after initialization
• PR #6553: Drop prettyplotlib from the list of toolkits.
• PR #6557: Merge 2.x to master
• PR #5626: New toolbar icons
• PR #6555: Fix docstrings for warn_deprecated.
• PR #6544: Fix typo in margins handling.
• PR #6014: Patch for issue #6009
• PR #6517: Fix conversion of string grays with alpha.
• PR #6522: DOC: made sure boxplot demos share y-axes
• PR #6529: TST Remove plt.show() from test_axes.test_dash_offset
• PR #6519: Fix FigureCanvasAgg.print_raw(...)
• PR #6481: Default boxplot style rebase
• PR #6504: Patch issue 6035 rebase
• PR #5593: ENH: errorbar color cycle clean up
• PR #6497: Line2D._path obeys drawstyle.
• PR #6487: Added docstring to scatter_with_legend.py [MEP12]
• PR #6485: Barchart demo example clean up [MEP 12]
• PR #6472: Install all dependencies from pypi
• PR #6482: Skip test broken with numpy 1.11
• PR #6475: Do not turn on interactive mode on in example script
- PR #6442: loading TCL / Tk symbols dynamically
- PR #6467: ENH: add unified seaborn style sheet
- PR #6465: updated boxplot figure
- PR #6462: CI: Use Miniconda already installed on AppVeyor.
- PR #6456: FIX: unbreak master after 2.x merge
- PR #6445: Offset text colored by labelcolor param
- PR #6417: Showraise gtk gtk3
- PR #6423: TST: splitlines in rec2txt test
- PR #6427: Output pdf dicts in deterministic order
- PR #6431: Merge from v2.x
- PR #6433: Make the frameworkpython script compatible with Python 3
- PR #6358: Stackplot weighted_wiggle zero-area fix
- PR #6382: New color conversion machinery.
- PR #6372: DOC: add whats_new for qt configuration editor.
- PR #6415: removing unused DialogLineprops from gtk3
- PR #6390: Use xkcd: prefix to avoid color name clashes.
- PR #6397: key events handler return value to True to stop propagation
- PR #6402: more explicit message for missing image
- PR #5785: Better choice of offset-text.
- PR #6302: FigureCanvasQT key auto repeat
- PR #6334: ENH: webagg: Handle ioloop shutdown correctly
- PR #5267: AutoMinorLocator and and logarithmic axis
- PR #6386: Minor improvements concerning #6353 and #6357
- PR #6388: Remove wrongly commited test.txt
- PR #6379: Install basemap from git trying to fix build issue with docs
- PR #6369: Update demo_floating_axes.py with comments
- PR #6377: Remove unused variable in GeoAxes class
- PR #6373: Remove misspelled and unused variable in GeoAxes class
- PR #6376: Update index.rst - add Windrose as third party tool
- PR #6371: Set size of static figure to match widget on hidp displays
- PR #6370: Restore webagg backend following the merge of widget nbagg backend
- PR #6366: Sort default labels numerically in Qt editor.
- PR #6367: Remove stray nonascii char from nbagg
- PR #5754: IPython Widget
- PR #6146: ticker.LinearLocator view_limits algorithm improvement closes #6142
- PR #6287: ENH: add axisbelow option ‘line’, make it the default
- PR #6339: Fix #6335: Queue boxes to update
- PR #6347: Allow setting image clims in Qt options editor.
- PR #6354: Update events handling documentation to work with Python 3.
- PR #6356: Merge 2.x to master
- PR #6304: Updating animation file writer to allow keyword arguments when using with construct
- PR #6328: Add default scatter marker option to rcParams
- PR #6342: Remove shebang lines from all examples. [MEP12]
- PR #6337: Add a ‘useMathText’ param to method ‘ticklabel_format’
- PR #6346: Avoid duplicate cmap in image options.
- PR #6253: MAINT: Updates to formatters in matplotlib.ticker
- PR #6291: Color cycle handling
- PR #6340: BLD: make minimum cycler version 0.10.0
- PR #6322: Typo fixes and wording modifications (minor)
- PR #6319: Add PyUpSet as extension
- PR #6314: Only render markers on a line when markersize > 0
- PR #6303: DOC Clean up on about half the Mplot3d examples
- PR #6311: Seaborn sheets
- PR #6300: Remake of #6286
- PR #6297: removed duplicate word in Choosing Colormaps documentation
- PR #6200: Tick vertical alignment
- PR #6203: Fix #5998: Support fallback font correctly
- PR #6198: Make hatch linewidth an rcParam
- PR #6275: Fix cycler validation
- PR #6283: Use figure.stale instead of internal member in macosx
- PR #6247: DOC: Clarify fillbetween_x example.
- PR #6251: ENH: Added a PercentFormatter class to matplotlib.ticker
- PR #6267: MNT: trap inappropriate use of color kwarg in scatter; closes #6266
- PR #6249: Adjust test tolerance to pass for me on OSX
• PR #6263: TST: skip broken test
• PR #6260: Bug fix and general touch ups for hist3d_demo example (#1702)
• PR #6239: Clean warnings in examples
• PR #6170: getter for ticks for colorbar
• PR #6246: Merge v2.x into master
• PR #6238: Fix sphinx 1.4.0 issues
• PR #6241: Force Qt validator to use C locale.
• PR #6234: Limit Sphinx to 1.3.6 for the time being
• PR #6178: Use Agg for rendering in the Mac OSX backend
• PR #6232: MNT: use stdlib tools in allow_rasterization
• PR #6211: A method added to Colormap classes to reverse the colormap
• PR #6205: Use io.BytesIO instead of io.StringIO in examples
• PR #6229: Add a locator to AutoDateFormatters example code
• PR #6222: ENH: Added file keyword to setp to redirect output
• PR #6217: BUG: Made setp accept arbitrary iterables
• PR #6154: Some small cleanups based on Quantified code
• PR #4446: Label outer offset text
• PR #6218: DOC: fix typo
• PR #6202: Fix #6136: Don’t hardcode default scatter size
• PR #6195: Documentation bug #6180
• PR #6194: Documentation bug fix: #5517
• PR #6011: Fix issue #6003
• PR #6179: Issue #6105: Adds targetfig parameter to the subplot2grid function
• PR #6185: Fix to csv2rec bug for review
• PR #6192: More precise choice of axes limits.
• PR #6176: DOC: Updated docs for rc_context
• PR #5617: Legend tuple handler improve
• PR #6188: Merge 2x into master
• PR #6158: Fix: pandas series of strings
• PR #6156: Bug: Fixed regression of drawstyle=None
• PR #5343: Boxplot stats w/ equal quartiles
• PR #6132: Don’t check if in range if the caller passed norm
• PR #6091: Fix for issue 5575 along with testing
• PR #6123: docstring added
• PR #6145: BUG: Allowing unknown drawstyles
• PR #6148: Fix: Pandas indexing Error in collections
• PR #6140: clarified color argument in scatter
• PR #6137: Fixed outdated link to thirdpartypackages, and simplified the page
• PR #6095: Bring back the module level ‘backend’
• PR #6124: Fix about dialog on Qt 5
• PR #6110: Fixes matplotlib/matplotlib#1235
• PR #6122: MNT: improve image array argument checking in to_rgba. Closes #2499.
• PR #6047: bug fix related #5479
• PR #6119: added comment on “usetex=False” to ainde debugging when latex not ava... 
• PR #6073: fixed bug 6028
• PR #6116: CI: try explicitly including msvc_runtime
• PR #6100: Update INSTALL
• PR #6099: Fix #6069. Handle image masks correctly
• PR #6079: Fixed Issue 4346
• PR #6102: Update installing_faq.rst
• PR #6101: Update INSTALL
• PR #6074: Fixes an error in the documentation, linestyle is dash_dot and should be dashdot
• PR #6068: Text class: changed __str__ method and added __repr__ method
• PR #6018: Added get_status() function to the CheckButtons widget
• PR #6013: Mnt cleanup pylab setup
• PR #5984: Suggestion for Rasterization to docs pgf-backend
• PR #5911: Fix #5895: Properly clip MOVETO commands
• PR #6039: DOC: added missing import to navigation_toolbar.rst
• PR #6036: BUG: fix ListedColormap._resample, hence plt.get_cmap; closes #6025
• PR #6029: TST: Always use / in URLs for visual results.
• PR #6022: Make @cleanup really support generative tests.
• PR #6024: Add Issue template with some guidelines
• PR #5718: Rewrite of image infrastructure
• PR #3973: WIP: BUG: Convert qualitative colormaps to ListedColormap
• PR #6005: FIX: do not short-cut all white-space strings
• PR #5727: Refresh pgf baseline images.
• PR #5975: ENH: add kwarg normalization function to cbook
• PR #5931: use `locale.getpreferredencoding()` to prevent OS X locale issues
• PR #5972: add support for PySide2, #5971
• PR #5625: DOC: add FAQ about np.datetime64
• PR #5131: fix #4854: set default numpoints of legend entries to 1
• PR #5926: Fix #5917. New dash patterns. Scale dashes by lw
• PR #5976: Lock calls to latex in texmanager
• PR #5628: Reset the available animation movie writer on rcParam change
• PR #5951: tkagg: raise each new window; partially addresses #596
• PR #5958: TST: add a test for tilde in tempfile for the PS backend
• PR #5957: Win: add mgs as a name for ghostscript executable
• PR #5928: fix for latex call on PS backend (Issue #5895)
• PR #5954: Fix issues with getting tempdir when unknown uid
• PR #5922: Fixes for Windows test failures on appveyor
• PR #5953: Fix typos in Axes.boxplot and Axes.bxp docstrings
• PR #5947: Fix #5944: Fix PNG writing from notebook backend
• PR #5936: Merge 2x to master
• PR #5629: WIP: more windows build and CI changes
• PR #5914: Make barbs draw correctly (Fixes #5803)
• PR #5906: Merge v2x to master
• PR #5809: Support generative tests in @cleanup.
• PR #5910: Fix reading/writing from urllib.request objects
• PR #5882: mathtext: Fix comma behaviour at start of string
• PR #5880: mathtext: Fix bugs in conversion of apostrophes to primes
• PR #5872: Fix issue with Sphinx 1.3.4
• PR #5894: Boxplot concept figure update
• PR #5870: Docs / examples fixes.
• PR #5892: Fix gridspec.Gridspec: check ratios for consistency with rows and columns
• PR #5901: Fixes incorrect ipython sourcecode
• PR #5893: Show significant digits by default in QLineEdit.
• PR #5881: Allow build children to run
• PR #5886: Revert “Build the docs with python 3.4 which should fix the Traitlets…”
• PR #5877: DOC: added blurb about external mpl-proscale package
• PR #5879: Build the docs with python 3.4 which should fix the Traitlets/IPython…
• PR #5871: Fix sized delimiters for regular-sized mathtext (#5863)
• PR #5852: FIX: create _dashSeq and _dashOffset before use
• PR #5832: Rewordings for normalizations docs.
• PR #5849: Update setupext.py to solve issue #5846
• PR #5853: Typo: fix some typos in patches.FancyArrowPatch
• PR #5842: Allow image comparison outside tests module
• PR #5845: V2.x merge to master
• PR #5813: mathtext: no space after comma in brackets
• PR #5828: FIX: overzealous clean up of imports
• PR #5826: Strip spaces in properties doc after newline.
• PR #5815: Properly minimize the rasterized layers
• PR #5752: Reorganise mpl_toolkits documentation
• PR #5788: Fix ImportError: No module named ‘StringIO’ on Python 3
• PR #5797: Build docs on python3.5 with linkcheck running on python 2.7
• PR #5778: Fix #5777. Don’t warn when applying default style
• PR #4857: Toolbars keep history if axes change (navtoolbar2 + toolmanager)
• PR #5790: Fix ImportError: No module named ‘Tkinter’ on Python 3
• PR #5789: Index.html template. Only insert snippet if found
• PR #5783: MNT: remove reference to deleted example
• PR #5780: Choose offset text from ticks, not axes limits.
• PR #5776: Add .noseids to .gitignore.
• PR #5466: Fixed issue with rasterized not working for errorbar
• PR #5773: Fix eb rasterize
• PR #5440: Fix #4855: Blacklist rcParams that aren’t style
• PR #5764: BUG: make clabel obey fontsize kwarg
• PR #5771: Remove no longer used Scikit image code
• PR #5766: Deterministic LaTeX text in SVG images
• PR #5762: Don’t fallback to old ipython_console_highlighting
- PR #5728: Use custom RNG for sketch path
- PR #5454: ENH: Create an abstract base class for movie writers.
- PR #5600: Fix #5572: Allow passing empty range to broken_barh
- PR #4874: Document mpl_toolkits.axes_grid1.anchored_artists
- PR #5746: Clarify that easy_install may be used to install all dependencies
- PR #5739: Silence labeled data warning in tests
- PR #5732: RF: fix annoying parens bug
- PR #5735: Correct regex in filterwarnings
- PR #5640: Warning message prior to fc-list command
- PR #5686: Remove banner about updating styles in 2.0
- PR #5676: Fix #5646: bump the font manager version
- PR #5719: Fix #5693: Implemented is_sorted in C
- PR #5721: Remove unused broken doc example axes_zoom_effect
- PR #5664: Low-hanging performance improvements
- PR #5709: Addresses issue #5704. Makes usage of parameters clearer
- PR #5716: Fix #5715.
- PR #5690: Fix #5687: Don’t pass unicode to QApplication()
- PR #5707: Fix string format substitution key missing error
- PR #5706: Fix SyntaxError on Python 3
- PR #5700: BUG: handle colorbar ticks with boundaries and NoNorm; closes #5673
- PR #5702: Add missing substitution value
- PR #5701: str.formatter invalid
- PR #5697: TST: add missing decorator
- PR #5683: Include outward ticks in bounding box
- PR #5688: Improved documentation for FuncFormatter formatter class
- PR #5469: Image options
- PR #5677: Fix #5573: Use SVG in docs
- PR #4864: Add documentation for mpl_toolkits.axes_grid1.inset_locator
- PR #5434: Remove setup.py tests and adapt docs to use tests.py
- PR #5586: Fix errorbar extension arrows
- PR #5653: Update banner logo on main website
- PR #5667: Nicer axes names in selector for figure options.
• PR #5672: Fix #5670. No double endpoints in Path.to_polygon
• PR #5553: qt: raise each new window
• PR #5594: FIX: formatting in LogFormatterExponent
• PR #5588: Adjust number of ticks based on length of axis
• PR #5671: Deterministic svg
• PR #5659: Change savefig.dpi and figure.dpi defaults
• PR #5662: Bugfix for test_triage tool on Python 2
• PR #5661: Fix #5660. No FileNotFoundError on Py2
• PR #4921: Add a quit_all key to the default keymap
• PR #5651: Shorter svg files
• PR #5656: Fix #5495. Combine two tests to prevent race cond
• PR #5383: Handle HiDPI displays in WebAgg/NbAgg backends
• PR #5307: Lower test tolerance
• PR #5631: WX/WXagg backend add code that zooms properly on a Mac with a Retina display
• PR #5644: Fix typo in pyplot_scales.py
• PR #5639: Test if a frame is not already being deleted before trying to Destroy.
• PR #5583: Use data limits plus a little padding by default
• PR #4702: sphinxext/plot_directive does not accept a caption
• PR #5612: mathtext: Use DejaVu display symbols when available
• PR #5374: MNT: Mailmap fixes and simplification
• PR #5516: OSX virtualenv fixing by creating a simple alias
• PR #5546: Fix #5524: Use large, but finite, values for contour extensions
• PR #5621: Tst up coverage
• PR #5620: FIX: quiver key pivot location
• PR #5607: Clarify error when plot() args have bad shapes.
• PR #5604: WIP: testing on windows and conda packages/ wheels for master
• PR #5611: Update colormap user page
• PR #5587: No explicit mathdefault in log formatter
• PR #5591: fixed ordering of lightness plots and changed from getting lightness ...
• PR #5605: Fix DeprecationWarning in stackplot.py
• PR #5603: Draw markers around center of pixels
• PR #5596: No edges on filled things by default
- PR #5249: Keep references to modules required in pgf LatexManager destructor
- PR #5589: return extension metadata
- PR #5566: DOC: Fix typo in Axes.bxp.__doc__
- PR #5570: use base64.encodestring on python2.7
- PR #5578: Fix #5576: Handle CPLUS_INCLUDE_PATH
- PR #5555: Use shorter float repr in figure options dialog.
- PR #5552: Dep contourset vminmax
- PR #5433: ENH: pass dash_offset through to gc for Line2D
- PR #5342: Sort and uniquify style entries in figure options.
- PR #5484: fix small typo in documentation about CheckButtons.
- PR #5547: Fix #5545: Fix collection scale in data space
- PR #5500: Fix #5475: Support tolerance when picking patches
- PR #5501: Use facecolor instead of axisbg/axisbgcolor
- PR #5544: Revert “Fix #5524. Use finfo.max instead of np.inf”
- PR #5146: Move impl. of plt.subplots to Figure.add_subplots.
- PR #5534: Fix #5524. Use finfo.max instead of np.inf
- PR #5521: Add test triage tool
- PR #5537: Fix for broken matplotlib.test function
- PR #5539: Fix docstring of violin{,plot} for return value.
- PR #5515: Fix some theoretical problems with png reading
- PR #5526: Add boxplot params to rctemplate
- PR #5533: Fixes #5522, bug in custom scale example
- PR #5514: adding str to force string in format
- PR #5512: V2.0.x
- PR #5465: Better test for isarray in figaspect(). Closes #5464.
- PR #5503: Fix #4487: Take hist bins from rcParam
- PR #5485: Contour levels must be increasing
- PR #4678: TST: Enable coveralls/codecov code coverage
- PR #5437: Make “classic” style have effect
- PR #5458: Removed normalization of arrows in 3D quiver
- PR #5480: make sure an autoreleasepool is in place
- PR #5451: [Bug] masking of NaN Z values in pcolormesh
- PR #5453: Force frame rate of FFMpegFileWriter input
- PR #5452: Fix axes.set_prop_cycle to handle any generic iterable sequence.
- PR #5448: Fix #5444: do not access subsuper nucleus _metrics if not available
- PR #5439: Use DejaVu Sans as default fallback font
- PR #5204: Minor cleanup work on navigation, text, and customization files.
- PR #5432: Don’t draw text when it’s completely clipped away
- PR #5426: MNT: examples: Set the aspect ratio to “equal” in the double pendulum animation.
- PR #5214: Use DejaVu fonts as default for text and mathtext
- PR #5306: Use a specific version of Freetype for testing
- PR #5410: Remove uses of font.get_charmap
- PR #5407: DOC: correct indentation
- PR #4863: [mpl_toolkits] Allow “figure” kwarg for host functions in parasite_axes
- PR #5166: [BUG] Don’t allow 1d-arrays in plot_surface.
- PR #5360: Add a new memleak script that does everything
- PR #5361: Fix #347: Faster text rendering in Agg
- PR #5373: Remove various Python 2.6 related workarounds
- PR #5398: Updating 2.0 schedule
- PR #5389: Faster image generation in WebAgg/NbAgg backends
- PR #4970: Fixed ZoomPanBase to work with log plots
- PR #5387: Fix #3314 assert mods.pop(0) fails
- PR #5385: Faster event delegation in WebAgg/NbAgg backends
- PR #5384: BUG: Make webagg work without IPython installed
- PR #5358: Fix #5337. Turn off --no-capture (-s) on nose
- PR #5379: DOC: Fix typo, broken link in references
- PR #5371: DOC: Add what’s new entry for TransformedPatchPath.
- PR #5299: Faster character mapping
- PR #5356: Replace numpy funcs for scalars.
- PR #5359: Fix memory leaks found by memleak_hawaii3.py
- PR #5357: Fixed typo
- PR #4920: ENH: Add TransformedPatchPath for clipping.

Issues (1130):
- #8599: Pie Chart from CSV File
• #8586: update errorbar in Matplotlib 2.0.0
• #8463: wrong hatch color in legend
• #8558: Rendering really large image
• #8312: Matplotlib attempts to import PyQt4 when PyQt5 is not available
• #3528: PS backend is not tested
• #4389: Windows installer does not run -> restarts itself continually
• #8592: Qt4 backend (PySide) seemingly tries to use Qt5
• #8579: Python 2.7 travis build failing
• #8349: [feature request] Accepting slice arguments in ax.set_xlim() (Python 3)
• #4379: for the root example page, please provide more description
• #8571: plt.subplots return is inconsistent
• #8570: release 2.0.1 has qt4-incompatible code in backend_qt5.py
• #8569: Superimposed markers when scatterpoints=1
• #8565: Unexpected mixing of qt4backend and qt5backend
• #8563: mpl2.0.1 seems to have broken Qt4Agg
• #8562: ‘QPixmap’ object has no attribute ‘setDevicePixelRatio’
• #8560: Calling close() on a figure doesn’t seem to close it
• #8174: Add pillow and graphviz to doc build dependencies
• #8501: Remove false deprecation warning
• #7412: Documentation guidelines improvements 2
• #8541: Generate a tutorials sphinx gallery
• #8223: need to backport docathon PRs
• #7206: All examples should be MEP12/sphinx-gallery compliant
• #6457: The first subplot is missized after savefig to a png file.
• #8521: Rectangle patch not transformed correctly in polar plot
• #8542: Inconsistent image size with savefig(....,type=’png’)
• #8445: Cannot display np.array with dtype = np.float128
• #8508: %matplotlib notebook show nothing, both 2d and 3d plot
• #7289: ipython loses auto-show/auto-redraw behavior after call to plt.rcdefaults()
• #6284: ax.twinx().plot() will reset the x_limits if only an axvspan was used on ax
- #8527: Confusing docstring in matplotlib.pyplot.bar() in version 2.0.0
- #8316: Matplotlib User Interface Example Breaks With More Than Three Patches
- #7835: Deprecate is_string_like
- #8524: Errorbar limits arrow heads point in wrong direction on inverted axis
- #8520: Documentation builds are failing due to sphinx-gallery changes
- #8514: Can’t plot with empty markers
- #8516: .set_data() errors if number of points change
- #8517: Log-scale ignores Formatter
- #8506: text object clipping
- #6921: “Error: local variable ‘xdata’ referenced before assignment” in legend_handler.py
- #8505: plot window hangs and/or is generally unresponsive
- #8500: plot disappears when moved to second monitor
- #7523: Multi-line text instances differing in linespacing not rendered correctly
- #7725: is_string_like returns True for numpy object arrays
- #8057: markevery only accepts builtin integers, not numpy integers
- #8078: plt.subplots crashes when handed fig_kw argument
- #8038: Question on Path.contains_point/s
- #7688: Edit axis with multiple figures causes freeze with Qt5 on Windows
- #7754: Forgotten restore bounds?
- #5510: autoscale context manager
- #6649: UnboundLocalError in hist(x, bins, histtype='step', normed=1) on double entries in bins
- #6805: Axes.hist with no data in range raises UnboundLocalError
- #7512: ImportError: No module named Tkinter
- #6704: Spacing in math text is broken (AIX specific)
- #8050: Version 2.0.0.1 breaks OpenType font character spacing on PDF save
- #7924: Python 3.6 deprecated escape sequences.
- #8030: Unable to install matplotlib package using whl
- #8079: Inconsistency between ” and ‘ in the code
- #8128: figure.Figure.autofmt_xdate applied to major xtick labels only
- #8168: From matplotlib.font_manager, ImportError: cannot import name get_font
- #8220: “search” doesn’t find cmocean
- #8296: Remove idle_event from examples/event_handling/idle_and_timeout.py
• #8242: Investigate alternative svg renderers for the test suite
• #8424: does matplotlib install in virtual environment work?
• #8460: matplotlib API docs missing container module
• #8467: initialise imshow with zero array has unexpected side effects
• #7460: Raise error if argument to xlim is invalid, e.g., nan
• #8465: zorder values as a sequence are not respected by LineCollection
• #8457: Allow to change base of LogNorm?
• #8406: FancyArrow, error in the polygon coordinates for shape ‘full’
• #8431: Hatches filling an empty color contour not saved in pdf
• #7989: 300% CPU on Raspberry Pi
• #537: Orthogonal projection for mplot3d
• #8443: No version when building from github archive
• #8444: Matplotlib hangs with runtime error
• #8441: ImportError: No module named _backend_gdk
• #8302: Invalid certificate at https://matplotlib.org.
• #8432: asd
• #8153: Long lines in literal blocks run off the edge of the page
• #8160: Using plt.ion with startup commands
• #8428: Documentation: matplotlibrc example file confuses x and yticks.
• #8425: MaxNLocator prune isn’t working with decimals
• #8421: Consistent deprecation of the hold kwarg of hlines and vlines
• #8427: Matplotlib cannot find default matplotlib fonts on Scientific linux
• #7903: Regression: imshow on geo axes produces an empty plot in matplotlib 2.0.0
• #8419: Incorrect display of minor ticks in Log Scale plot with large font size
• #8418: Matplotlib defaults to and bundles vera fonts that do not contain Greek letters
• #8073: Please add Vega in perception documentation
• #8272: Convert docstring of Axes.pie() into numpydoc
• #8416: Python 3.4 image comparison test failure for mathtext_cm_52-expected.png
• #8402: plotting with TkAgg backend becomes unresponsive after a certain qt5agg backend import command
• #8412: Data points not rendered for figures saved in vector formats
• #8397: pyplot.subplots did not return the type of object specified in the documentation
• #8409: loading pickled figure gives: AttributeError: ‘CallbackRegistry’ object has no attribute ‘call-backs’
• #8111: Issue with sub-setting minor-ticks at intermediate number of decades
• #8401: weird bug when using custom formatter; tick labels overlapping
• #8398: restore rebase documentation
• #7227: Path effects for text outline is missing corners
• #3517: Issue with non-ascii paths in font look up
• #8208: make.py should not use os.system("sphinx-build ...")
• #8386: setting log major ticks leaves interfering minor tick labels
• #8370: Simpler code for common plot setting (e.g. legend, lable, title, figsize)?
• #8385: Make checkbox rectangles in CheckButtons widget have an edge outline
• #7785: Passing a transposed array to patch.set_offsets()
• #8378: ColorbarBase() is dpi dependent
• #8342: Make ArrowStyle docstrings numpydoc compatible
• #7683: Please add <shade = True> parameter to bar3d
• #8337: axes set_position ignored if using tight_layout
• #8260: test_backend_ps.py leaves temporary files in /tmp
• #8355: Artifact at border with tricontourf and float32 positions
• #8330: PyPy compatibility - __builtin__ is not iterable
• #8017: Font family is not set on log-scaled tick labels
• #8317: Fonts not changed in LaTeX mode (since version 2.0.0)
• #7293: Isolated points missing in pdf
• #8348: style.use not working for some cases
• #7655: Event picking does not seem to work on polar bar plots
• #3540: Pick events broken in log axes
• #8124: Actually deprecate Axes.axesPatch, Figure.figurePatch
• #8290: six environment polluted when misuse of matplotlib import
• #8230: cache local freetype source
• #8332: Importing Issue: “No module named externals”
• #8327: Inconsistency between documentation and actual effect of the bar plot function
• #8197: Matplotlib 2.0.0 crashes on plotting contour of array with two dimensions of size 1 in Python 3.4
• #8054: is_scalar_or_string deprecated too early
• #8284: Markers have compression artifacts
• #8294: x marker is too big
• #8288: che
• #7951: matplotlib 2.0.0 not using matplotlibrc file in IPython
• #8012: imshow interpolation uses masked values
• #8117: Trying to plot only error bars (with no line connecting data points) results in empty plot
• #7953: Cannot import _tkagg on windows 64 bits on 2.7
• #8225: tight_layout does not work with set_visible(False)
• #8145: Warning treated as error while generating docs
• #2304: Add an argument rotate_labels to pie chart
• #8046: Arc patch with starting and ending angle
• #8263: error saving pcolormesh with shading=gouraud to eps
• #8034: necked_tensile_specimen.png contains non-free color calibration profile
• #8231: gitter badge at top of README.rst is broken
• #8141: breaking change in dash verification in 2.0.0
• #8207: Add early check for “dot” binary (graphviz) when building docs
• #8194: Sample data not found
• #8186: Put back gitter plain text link on README.rst
• #8198: ticklabel font changes when using logscale
• #7616: make ‘dpi’ optional for animation.MovieWriter.setup
• #8192: %matplotlib inline doesn’t work well with ipywidgets 6.0 interact
• #8189: pip install matploblib broken
• #8177: Add badges to README.rst
• #8180: icu version problem with matplotlib 2
• #8169: Autowrapping text doesn’t work in “What’s New in Matplotlib 1.5” example
• #8164: Remove yield tests (now deprecated in pytest)
• #8159: Doc: Restructure environment variables in Installing from source docs
• #8123: Regenerate gitwash docs for improved syntax highlighting of non-python code sections
• #8147: Use Agg backend by default if DISPLAY is not set
• #8146: How to display D-stroke symbol in matplotlib
• #7905: ytick locations different for 1.5.3 and 2.0.0
• #8121: PNG output with PGF backend fails with pdftocairo exit code 99
• #8136: If division by zero, python cannot convert float infinity to integer in get_tick_space()
• #8067: Coordinates of text not properly set in pgf files
• #8112: Deprecate assert_str_equal
• #8122: keyword labelrotation is not recognized
• #5151: Horizontal lines in colorbars in PDF
• #4984: errorbar incorrectly accepts and plots Nx2 shaped yerr
• #6593: Feature request: filled error bar
• #7992: Hatching doesn’t respect alpha
• #7991: Dotted grid lines in matplotlib v2.0.0 appear dashed
• #8015: Document new testing procedure
• #8106: Error in demo scripts: to_rgba: Invalid rgba arg “2” in style_sheets_reference.py
• #8105: calling plt.hist() with np.nan value raises confusing error massage
• #5320: Some searches in documentation result in no results
• #6042: return _text property as __repr__ for Text class
• #7171: axes.boxplot does not have zorder kwarg
• #8098: matplotlibrc doesn’t work
• #8091: Bug with plt.hist when min==max and values are >= 2**53
• #8087: zorder is not respected by all parts of errorbar #1622 problem in matplotlib 2.0.0
• #7877: add link to scipython book
• #8053: save animation to mp4 fail
• #8076: Matplotlib hatch color broken
• #8077: ImportError: No module named ‘matplotlibrcsetup’
• #7686: ‘Extend’ keyword produces colorbar of zero width
• #7886: empty imshow crashes python
• #7901: Allow multiple hatch colors in same figure
• #6708: KnownFailure becomes an error with --processes=1 flag
• #7899: Behaviour of c=None is inconsistent in 2.0
• #8058: Updating matplotlib to 2.0.0 causing problems when plotting: error/warning message “failed to get the current screen resources”
• #8051: includes not-really-free BaKoMa fonts
• #8049: Matplotlib 3D projection axis cannot be created because of missing rc-Param[u’_internal.classic_mode’]
• #6285: plt.subplots() does not remove existing subplots when called on existing figure
- #8023: Numpy array not accepted for log scaled minor ticks
- #7967: Catch or stop specgram warnings during tests
- #7813: Qt5Agg backend not rendering nicely on Windows HiDPI display
- #6891: Greek letter display error [1.5.1]
- #6393: Pair of floats breaks plotting renderer (weirdest bug I’ve ever seen)
- #8027: Log Scale Ticklabels Refuse to Die (Data Dependent)
- #5859: savefig() as PNG get a different result than the image shown in ipython notebook (SVG probably)
- #5210: Unexpected replacement of right) with exclamation point in MathTextParser output
- #6902: Include test files in coverage report
- #7756: Cannot install the matplotlib 1.5.3 or 2.0.0rc2 on Mac OS Sierra, Python 3.6.0
- #5325: Migrate to py.test
- #8016: Unexpected HOME behavior after axis change
- #8009: Need Python 3.6 wheels for matplotlib 1.5.1 on PyPI
- #8005: Hex colours with alpha
- #7807: ValueError when specifying marker vertices
- #7937: Large TrueType font does not render correctly when saving in .pdf
- #7615: Subscript/superscript sequence is too long with $left. X right|$.
- #5287: Warnings in doc build
- #6860: The original xlim changed by twinx
- #6064: specgram(x) should warn if x.size < 256
- #7964: BUG: AxesImage.set_data([]) causes FloatingPointException
- #7963: savefig PDF removing the negative sign from yticklabels
- #7955: Failed install with “OSError: [Errno 12] Cannot allocate memory”
- #7958: Matplotlib 2.0 cannot find PyQt5 (Windows)
- #7943: ValueError when plotting Pandas Dataframe Summary
- #7940: Saving a square image constrained to the extent of axis without any border doesn’t work
- #7636: WX mouseevent assertion errors
- #7902: Saving an unpickled figure.
- #6048: AttributeError (no attribute ‘callbacks’) when setting attributes on pickle-loaded PatchCollection
- #7930: KeyError while Matplotlib is displaying >= 2 figures causes X server crash
- #7874: Wheels not picking correct default backend
• #7906: font_manager.get_fontconfig_fonts() reports a single item of all fonts concatenated
• #7922: FT2Font do not close open file, leads to delayed ResourceWarning
• #7803: tripcolor documentation rendering error.
• #7891: pyplot.bar does not cycle through colors (>2.0.0b1)
• #7912: matplotlib doesn’t work with numpy+mkl
• #7921: Text rendered with MathText looks really ugly (version 2.0 vs. 1.5)
• #7919: ~/.matplotlib/matplotlibrc not being read
• #7909: Unit length quiver arrow not drawn with a unit length
• #7852: _rrule maximum recursion depth exceeded on multiprocessing usage
• #7501: Segmentation fault in QuadMeshGenerator on ppc64
• #7879: UTF-8 error on import of matplotlib 1.5.1
• #7601: color_cycle deprecation warning could provide more context to user
• #7875: qt shims: QFontMetrics missing on PyQt5
• #7860: widget radio buttons and check buttons no longer display outlines
• #7870: DOC: FAQ out-of-date
• #7510: better input validation on fill_between
• #7867: pyplot import error when current directory contains a file named sip.py
• #7248: Adding names to the color in the new (Vega) default color cycle
• #6207: axes_grid1.zoomed_inset_axes does not accept location as string
• #7775: Locator tick_values for images
• #7810: Dimensions sanity check in axes_rgb swaps x and y of shape when checking, prevents use with non-square images.
• #7704: screenshots in the front page of devdocs are ugly
• #7746: imshow should silence warnings on invalid values, at least when masked
• #7661: document that imshow now respects scale
• #6820: nonsensical error message for invalid input to plt.bar
• #7814: Legend for lines created using LineCollection show different handle line scale.
• #7816: re-enable or delete xmllint tests
• #7802: plt.stackplot not working for integer input with non-default ‘baseline’ parameters
• #6149: travis dedup
• #7822: Weird stroke join with patheffects
• #7784: Simple IndexError in artist.setp if empty list is passed
• #3354: Unnecessary argument in GraphicsContextBase get_linestyle
• #7820: Figure.savefig() not respecting bbox_inches='tight' when dpi specified
• #7715: Can’t import pyplot in matplotlib 2.0 rc2
• #7745: Wheel distributables include unnecessary files
• #7812: On MacOS Sierra with IPython, inconsistent results with %gui, %matplotlib magic commands and –gui, and –matplotlib command-line options for ipython and qtconsole; complete failure of qtconsole inline figures
• #7808: Basemap uses deprecated methods
• #7487: Funny things happen when a rectangle with negative width/height is passed to axes()
• #7649: –nose-verbose isn’t a correct option for nose
• #7656: imsave ignores origin option
• #7792: test_png.test_pngsuite.test fails on ppc64 (big-endian)
• #7788: Colorbars contain no colors when created on ppc64 (big-endian)
• #4285: plt.yscale(“log”) gives FloatingPointError: underflow encountered in multiply
• #7724: Can’t import Matplotlib.widgets.TextBox
• #7798: test_psd_csd_equal fails for 12 (but not all) of the test_mlab.spectral_testcases on ppc64 (big-endian)
• #7778: Different font size between math mode and regular text
• #7777: mpl_toolkits.basemap: ValueError: level must be >= 0
• #4353: different behaviour of zoom while using ginput with MouseEvent vs KeyEvent
• #4380: horizontalalignment ‘left’ and ‘right’ do not handle spacing consistently
• #7393: subplot(): incorrect description of deletion of overlapping axes in the docs
• #7759: matplotlib dynamic plotting
• #2025: TkAgg build seems to favor Framework Tcl/Tk on OS-X
• #3991: SIGINT is ignored by MacOSX backend
• #2722: limited number of grid lines in matplotlib?
• #3983: Issue when trying to plot points with transform that requires more/fewer coordinates than it returns
• #7734: inconsistent doc regarding keymap.fullscreen default value
• #7761: Deprecation warning for finance is very unclear
• #7223: matplotlib.rcsetup docs
• #3917: OS X Cursor Not working on command line
• #4038: Hist Plot Normalization should allow a ‘Per Bin’ Normalization
- #3486: Update Selection Widgets
- #7457: Improvements to pylab_examples/stock_demo.py
- #7755: Can’t open figure
- #7299: Raise an error or a warning when ylim’s min == 0 and yscale == “log”
- #4977: Improve resolution of canvas on HiDPI with PyQt5 backend
- #7495: Missing facecolor2d attribute
- #3727: plot_date() does not work with x values of type pandas.Timestamp (pandas version 0.15.0)?
- #3368: Variable number of ticks with LogLocator for a fixed number of tick labels displayed
- #1835: docstrings of cross-correlation functions (acorr and xcorr) need clarification
- #6972: quiverkey problem when angles=array
- #6617: Problem of fonts with LaTeX rendering due to fonts-lyx package
- #7717: make all deprecation warnings be mplDeprecation instances
- #7662: eventplot legend fails (linewidth)
- #7673: Baseline image reuse breaks parallel testing
- #7666: Default scaling of x-axis in specgram() is incorrect (i.e. the default value for the extent parameter)
- #7709: Running into problems in seaborn after upgrading matplotlib
- #7684: 3-D scatter plot disappears when overlaid over a 3-D surface plot.
- #7630: Unicode issue in matplotlib.dates
- #7678: add link to bokeh/colorcet
- #2078: linespacing of multiline texts.
- #6727: scipy 2016 sprint ideas
- #3212: Why are numpoints and scatterpoints two different keywords?
- #7697: Update INSTALL file to include py3.6
- #4428: Hyphen as a subscript doesn’t appear at certain font sizes
- #2886: The wrong Game symbol is used
- #7603: scatter color vs c
- #7660: 2.0rc2: title too close to frame?
- #7672: standardize classic/v2.x order is docs
- #7680: OverflowError: Python int too large to convert to C long during plotting simple numbers on debian testing
- #7664: BUG: super requires at least 1 argument
- #7669: rc on conda-forge
• #5363: Warnings from test_contour.test_corner_mask
• #7663: BUG: Can’t import matplotlib._backports
• #7647: Decorator for deprecation ignores arguments other than ‘message’
• #5806: FutureWarning with Numpy 1.10
• #6480: Setting markeredgecolor raises a warning
• #7653: legend doesn’t show all markers
• #7643: Matplotlib 2.0 deprecations
• #7642: imshow seems to “shift” grid.
• #7633: All attempts to plot fail with “OverflowError: Python int too large to convert to C long”
• #7637: Stacked 2D plots with interconnections in Matplotlib
• #7353: auto legend position changes upon saving the figure
• #7626: Saturation mask for imshow()
• #7623: potential bug with plt.arrow and plt.annotate when setting linestyle via tuples
• #7005: rcParams[‘font.size’] is consulted at render time
• #7587: BUG: shared log axes lose _minpos and revert to default
• #7493: Plotting zero values with logarithmic axes triggers OverflowError, Matplotlib hangs permanently
• #7595: math domain error using symlog norm
• #7588: 2.0.0rc1 cannot import name ‘_macosx’
• #2051: Consider making default verticalalignment baseline
• #4867: Add additional minor labels in log axis with a span less than two decades
• #7489: Too small axis arrow when savefig to png
• #7611: UnicodeDecodeError when using matplotlib save SVG file and open it again
• #7592: font cache: a possibility to disable building it
• #5836: Repeated warning about fc-list
• #7609: The best channel to ask questions related to using matplotlib
• #7141: Feature request: auto locate minor ticks on log scaled color bar
• #3489: matplotlib scatter shifts color codes when NaN is present
• #4414: Specifying histtype=’stepfilled’ and normed=True when using plt.hist causes ymax to be set incorrectly
• #7597: python complain about “This application failed to start because it could not find or load the Qt platform plugin ‘xcb’ ” after an update of matplotlib
• #7578: Validate steps input to MaxNLocator
• #7590: Subtick labels are not disabled in classic style
• #6317: PDF file generation is not deterministic - results in different outputs on the same input
• #6543: Why does fill_betweenx not have interpolate?
• #7437: Broken path to example with strpdate2num
• #7593: Issue: Applying Axis Limits
• #7591: Number of subplots in mpl.axes.Subplot object
• #7056: setup.py –name and friends broken
• #7044: location of convert in rcparams on windows
• #6813: avoid hiding edge pixels of images
• #7579: OS X libpng incompatibility
• #7576: v2.0.0rc1 conda-forge dependency issue
• #7558: Colorbar becomes 0 to 1 after colorbar ax.yaxis.set_major_formatter
• #7526: Cannot Disable TkAgg Backend
• #6565: Questionable margin-cancellation logic
• #7175: new margin system doesn’t handle negative values in bars
• #5201: issue with colorbar using LogNorm and extend='min’
• #6580: Ensure install requirements in documentation are up to date before release
• #5654: Update static images in docs to reflect new style
• #7553: frange returns a last value greater than limit
• #5961: track bdist_wheel release and remove the workaround when 0.27 is released
• #7554: TeX formula rendering broken
• #6885: Check if ~/.matplotlib/ is a symlink to ~/.config/matplotlib/
• #7202: Colorbar with SymmetricalLogLocator : issue when handling only negative values
• #7542: Plotting masked array lose data points
• #6678: dead links in docs
• #7534: nbagg doesn’t change figure’s facecolor
• #7535: Set return of type Axes in Numpydoc docstring return type hint for Figure.add_subplot and Figure.add_axes to help jedi introspection
• #7443: pdf doc build is sort of broken
• #7521: Figure.show() fails with Qt5Agg on Windows (plt.show() works)
• #7423: Latex cache error when building docs
• #7519: plt.table() without any kwargs throws exception
• #3070: remove hold logic from library
• #1910: Pylab contourf plot using Mollweide projection create artefacts
• #5350: Minor Bug on table.py
• #7518: Incorrect transData in a simple plot
• #6985: Animation of contourf becomes extremely slow
• #7508: Legend not displayed
• #7484: Remove numpy 1.6 specific work-arounds
• #6746: Matplotlib.pyplot 2.0.0b2 fails to import with Conda Python 3.5 on OS X
• #7505: Default color cycler for plots should have more than 8 colors
• #7185: Hexbin default edgecolors kwarg is misnamed
• #7478: ‘alpha’ kwarg overrides facecolor=’none’ when plotting circle
• #7375: Patch edgecolor of a legend item does not follow look of figure
• #6873: examples/api/skewt.py is not displaying the right part of the grid by default
• #6773: Shifted image extents in 2.0.0b3
• #7350: Colors drawn outside axis for hist2d
• #7485: Is there a way to subclass the zoom() function from the NavigationToolbar backends and modify its mouse button definition?
• #7396: Bump numpy minimal version to 1.7.0?
• #7466: missing trigger for autoscale
• #7477: v2.0.0b4 fails to build with python-3.5: Requires pygtk
• #7113: Problems with anatomy figure on v2.x
• #6722: Text: rotation inconsistency
• #7244: Codecov instead of coveralls?
• #5076: RuntimeError: LaTeX was not able to process the following string: ‘z=$\mathregular{\{ \}^{\{\_\\}}}$’ in matplotlib
• #7450: Using Matplotlib in Abaqus
• #7314: Better error message in scatter plot when len(x) != len(c)
• #7432: Failure to re-render after Line2D.set_color
• #6695: support markdown or similar
• #6228: Rasterizing patch changes filling of hatches in pdf backend
• #3023: contourf hatching and saving to pdf
• #4108: Hatch pattern changes with dpi
• #6968: autoscale differences between 1.5.1 and 2.0.0b3
- #7452: test_log_margins test failure
- #7143: spurious warning with nans in log-scale plot
- #7448: Relative lengths in 3d quiver plots
- #7426: prop_cycler validation over-zealous
- #6899: savefig has sideeffects
- #7440: Confusing examples in annotation_demo2
- #7441: Loading a matplotlib figure pickle within a tkinter GUI
- #6643: Incorrect margins in log scale
- #7356: plt.hist does not obey the hist.bins rcparams
- #6845: SVG backend: anomaly in gallery scatter legend
- #6527: Documentation issues
- #7315: Spectral vs spectral Deprecation warning
- #7428: from matplotlib.backends import _tkagg raises AttributeError: ‘module’ object has no attribute ‘__file__’
- #7431: %matplotlib notebook offsetting sns.palplot
- #7361: add multi-process flag as -j to test.py
- #7406: NaN causes plt.vlines to not scale y limits
- #7104: set offset threshold to 4
- #7404: obnoxious double warning at each script startup
- #7373: Regression in imshow
- #7166: Hatching in legends is broken
- #6939: wspace is not “The amount of width reserved for blank space between subplots” as documented
- #4026: control hatch linewidth and fill border linewidth separately
- #7390: MAINT move the examples from doc/pyplots to examples and make them reproducible
- #7198: style blacklist includes hardcopy.docstring but it should be docstring.hardcopy
- #7391: How to apply ax.margins to current axes limits?
- #7234: Improving documentation: Tests failing on a osx setup
- #7379: Mp4’s generated by movie writer do not appear work in browser
- #6870: Figure is unpicklable after savefig
- #6181: When using Agg driver, pickling fails with TypeError after writing figure to PDF
- #6926: SVG backend closes BytesIO on print if wereusetex=True and cleanup decorator used
- #3899: Pickle not working in interactive ipython session
• #7251: Improve violin plot demo
• #7146: symlog scale no longer shows labels on the negative side
• #3420: simple plotting of numpy 2d-arrays
• #7287: Make matplotlib.use() report where the backend was set first, in case of conflict
• #7305: RuntimeError In FT2Font with NISC18030.ttf
• #7351: Interactive mode seems to be broken on MacOSX
• #7313: Axes3D.plot_surface with meshgrid args stopped working
• #7281: rcparam encoding test is broken
• #7345: Annotation minor issue in the example linestyles.py
• #7210: variable frame size support in animation is a misfeature
• #5222: legend–plot handle association
• #7312: get_facecolors() reports incorrect colors
• #7332: plot range
• #1719: Can’t pickle bar plots: Failed to pickle attribute “gridline”
• #6348: When I run a file that uses matplotlib animation, I keep getting this error. Using OS X, Python 2.7, installed Python from python.org then used homebrew. Matplotlib install from pip.
• #5386: Error loading fonts on OSX 10.11
• #6448: hist step UnboundLocalError
• #6958: Document the verts kwarg to scatter
• #7204: Integrate sphinx-gallery to our user documentation
• #7325: Anaconda broken after trying to install matplotlib 2.0 beta (ubuntu)
• #7218: v1.5.3: marker=None no longer works in plot()
• #7271: BUG: symmetric kwarg in locator is not honored by contourf
• #7095: _preprocess_data interferes in the docstrings Notes section
• #7283: DOC: Misrendered URLs in the development_workflow section of devdocs.
• #7109: backport #7108 to v2.x
• #7265: Image watermark hidden by Axes in example
• #7263: axes.bxp fails without fliers
• #7274: Latex greek letters in axis labels
• #7186: matplotlib 1.5.3 raise TypeError: ‘module’ object is not subscriptable on pylab.py
• #6865: custom_projection_example.py is completely out of date
• #7224: FancyArrowPatch linestyle always solid
• #7215: BUG: bar deals with bytes and string x data in different manners, both that are unexpected
• #7270: Pylab import
• #7230: subplots docstring: no example of NxN grid
• #7269: documentation: texinfo markup error in matplotlib 1.4.3 and matplotlib 1.5.3
• #7264: matplotlib dependency cycle matplotlib <- ipython <- matplotlib - how to resolve?
• #7261: Legend not displayed in Plot-Matplotlib lib
• #7260: Unknown exception in resize
• #7259: autoscaling of yaxis fails
• #7257: How can plot a figure with matplotlib like this?
• #3959: setting up matplotlib for development
• #7240: New tests without baseline images never produce a result
• #7156: Inverted imshow using Cairo backend
• #6723: How to customize violinplots?
• #5423: fill_between wrong edge line color
• #5999: Math accents are not correctly aligned
• #1039: Cairo backend marker/line style
• #7174: default value of lines.dash_capstyle
• #7246: Inconsistent behaviour of subplots for one and more-than-one axes
• #7228: axes tick_params label color not respected when showing scientific notation for axes scale
• #7225: get_geometry() wrong if subplots are nested (e.g., subplots with colorbars)
• #7221: Why does pyplot display wrong grayscale image?
• #7191: BUG: Animation bugs fixed in master should be backported to 2.x
• #7017: Doc typos in “Our favorite recipes”
• #3343: Issues with imshow and RGBA values
• #7157: should fill_between Cycle?
• #7159: test_colors.test_Normailize fails in 2.0.0b4 on Fedora rawhide/aarch64 (ARMv8)
• #7201: RGBA values produce different result for imshow and for markers
• #3232: Navigation API Needed
• #7001: Default log ticker can make too many ticks
• #806: Provide an option for the Animation class to retain the previously rendered frames
• #6135: matplotlib.animate writes png frames in cwd instead of temp files
• #7189: graph not showing when I set format to line
• #7080: Difference in symbol sizes using Mathtext with stixsans
• #7162: _axes.py linestyle_map unused
• #7163: pyplot.subplots() is slow
• #7161: matplotlib.ticker.FormatStrFormatter clashes with ax2.set_yticklabels when dual y-axis is used
• #6549: Log scale tick labels are overlapping
• #7154: bar graph with nan values leads to “No current point in closepath” in evince
• #7149: unable to save .eps plot
• #7090: fix building pdf docs
• #6996: FontProperties size and weight ignored by figure.suptitle
• #7139: float128s everywhere for dates?
• #7083: DOC: Clarify the relationship between plot and scatter
• #7125: Import Error on matplotlib.pyplot: PyQt4
• #7124: Updated matplotlib 1.5.3 broken in default Anaconda channel
• #6429: Segfault when calling show() after using Popen (test code inside)
• #7114: BUG: ax.tick_params change in tick length does not adjust tick labels
• #7120: Polar plot cos(2x)
• #7081: enh: additional colorblind-friendly colormaps
• #7103: Problem with discrete ListedColormaps when more than 4 colors are present
• #7115: Using matplotlib without Tkinter
• #7106: Wrong reader in mathext.py
• #7078: imshow() does not interpret aspect/extent when interpolation=’none’ in svg output
• #6616: Keyboard shortcuts for toggling minor ticks grid and opening figureoptions window
• #7105: Can’t pickle <type ‘instancemethod’>
• #7086: DOC (released) style is badly broken on the user doc.
• #7065: backport #7049
• #7091: v2.0.0b4 breaks viscm
• #7043: BUG: LogLocator.set_params is broken
• #7070: autoscale does not work for axes added by fig.add_axes()
• #3645: Proposal: Add rc setting to control dash spacing
• #7009: No good way to disable SpanSelector
• #7040: It is getting increasingly difficult to build the matplotlib documentation
• #6964: Docstring for ArtistAnimation is incorrect
• #6965: ArtistAnimation cannot animate Figure-only artists
• #7062: remove the contour on a Basemap object
• #7061: remove the contour on Basemap
• #7054: Whether the new version 2.0 will support high-definition screen?
• #7053: When will release 2.0 official version?
• #6797: Undefined Symbol Error On Ubuntu
• #6523: matplotlib-2.0.0b1 test errors on Windows
• #4753: rubber band in qt5agg slow
• #6959: extra box on histogram plot with a single value
• #6816: Segmentation fault on Qt5Agg when using the wrong linestyle
• #4212: Hist showing wrong first bin
• #4602: bar / hist : gap between first bar and other bars with lw=0.0
• #6641: Edge ticks sometimes disappear
• #7041: Python 3.5.2 crashes when launching matplotlib 1.5.1
• #7028: Latex Greek fonts not working in legend
• #6998: dash pattern scaling with linewidth should get it’s own rcParam
• #7021: How to prevent matplotlib from importing qt4 libraries when only
• #7020: Using tick_right() removes any styling applied to tick labels.
• #7018: Website Down
• #6785: Callbacks of draggable artists should check that they have not been removed
• #6783: Draggable annotations specified in offset coordinates switch to figure coordinates after dragging
• #7015: pcolor() not using “data” keyword argument
• #7014: matplotlib works well in ipython note book but can’t display in a terminal running
• #6999: cycler 0.10 is required due to change_key() usage
• #6794: Incorrect text clipping in presence of multiple subplots
• #7004: Zooming with a large range in y-values while using the linestyle “–” is very slow
• #6828: Spikes in small wedges of a pie chart
• #6940: large memory leak in new contour routine
• #6894: bar(..., linewidth=None) doesn’t display bar edges with mpl2.0b3
• #6989: bar3d no longer allows default colors
• #6980: problem accessing canvas on MacOS 10.11.6 with matplotlib 2.0.0b3
• #6804: Histogram of xarray.DataArray can be extremely slow
• #6859: Update URL for links to ggplot
• #6852: Switching to log scale when there is no positive data crashes the Qt5 backend, causes inconsistent internal state in others
• #6740: PGF Backend: Support interpolation='none'?
• #6665: regression: builtin latex rendering doesn’t find the right mathematical fonts
• #6984: plt.annotate(): segmentation fault when coordinates are too high
• #6979: plot won’t show with plt.show(block=False)
• #6981: link to ggplot is broken...
• #6975: [Feature request] Simple ticks generator for given range
• #6905: pcolorfast results in invalid cursor data
• #6970: quiver problems when angles is an array of values rather than ‘uv’ or ‘xy’
• #6966: No Windows wheel available on PyPI for new version of matplotlib (1.5.2)
• #6721: Font cache building of matplotlib blocks requests made to HTTPd
• #6844: scatter edgecolor is broken in Matplotlib 2.0.0b3
• #6849: BUG: endless loop with MaxNLocator integer kwarg and short axis
• #6935: matplotlib.dates.DayLocator cannot handle invalid input
• #6951: Ring over A in AA is too high in Matplotlib 1.5.1
• #6960: axvline is sometimes not shown
• #6473: Matplotlib manylinux wheel - ready to ship?
• #5013: Add Hershey Fonts a la IDL
• #6953: ax.vlines adds unwanted padding, changes ticks
• #6946: No Coveralls reports on GitHub
• #6933: Misleading error message for matplotlib.pyplot.errorbar()
• #6945: Matplotlib 2.0.0b3 wheel can’t load libpng in OS X 10.6
• #3865: Improvement suggestions for matplotlib.Animation.save(‘video.mp4’)
• #6932: Investigate issue with pyparsing 2.1.6
• #6941: Interfering with yahoo_finance
• #6913: Cant get currency from yahoo finance with matplotlib
• #6901: Add API function for removing legend label from graph
• #6510: 2.0 beta: Boxplot patches zorder differs from lines
• #6911: freetype build won’t become local
• #6866: examples/misc/longshort.py is outdated
• #6912: Matplotlib fail to compile matplotlib.png
• #1711: Autoscale to automatically include a tiny margin with Axes.errorbar()
• #6903: RuntimeError('Invalid DISPLAY variable') - With docker and django
• #6888: Can not maintain zoom level when left key is pressed
• #6855: imsave-generated PNG files missing edges for certain resolutions
• #6479: Hexbin with log scale takes extent range as logarithm of the data along the log axis
• #6795: suggestion: set_xticklabels and set_yticklabels default to current labels
• #6825: I broke imshow(<signed integer array>):-(
• #6858: PyQt5 pyplot error
• #6853: PyQt5 (v5.7) backend - TypeError upon calling figure()
• #6835: Which image formats to build in docs.
• #6856: Incorrect plotting for versions > 1.3.1 and GTK.
• #6838: Figures not showing in interactive mode with macosx backend
• #6846: GTK Warning
• #6839: Test test_pickle.test_complete is broken
• #6691: rcParam missing tick side parameters
• #6833: plot contour with levels from discrete data
• #6636: DOC: gallery supplies 2 pngs, neither of which is default
• #3896: dates.date2num bug with daylight switching hour
• #6685: 2.0 dev legend breaks on scatterplot
• #3655: ensure removal of font cache on version upgrade
• #6818: Failure to build docs: unknown property
• #6798: clean and regenerate travis cache
• #6782: 2.x: Contour level count is not respected
• #6796: plot/lines not working for datetime objects that span old dates
• #6660: cell focus/cursor issue when plotting to nbagg
• #6775: Last figure in http://matplotlib.org/users/pyplot_tutorial.html is not displayed correctly
• #5981: Increased tick width in 3D plots looks odd
• #6771: ImportWarning: No module named artist
• #6289: Grids are not rendered in backend implementation
• #6621: Change in the result of test_markevery_linear_scales_zoomed
• #6515: Dotted grid lines in v2.0.0b1
• #6511: Dependencies in installation of 2.0.0b1
• #6668: “Bachelor’s degrees…” picture in the gallery is cropped
• #6751: Tableau style
• #6742: import matplotlib.pyplot as plt throws an erro
• #6097: anaconda package missing nose dependency
• #6299: savefig() to eps/pdf does not work
• #6387: import matplotlib causes UnicodeDecodeError
• #6471: Colorbar label position different when executing a block of code
• #6732: Adding pairplot functionality?
• #6749: Step diagram does not support xlim() and ylim()
• #6748: Step diagram does not suppot
• #6615: Bad event index for step plots
• #6588: Different line styles between PNG and PDF exports.
• #6693: linestyle=”None” argument for fill_between() doesn’t work
• #6592: Linestyle pattern depends on current style, not style set at creation
• #5430: Linestyle: dash tuple with offset
• #6728: Can’t install matplotlib with specific python version
• #6546: Recommendation to install packages for various OS
• #6536: get_sample_data() in cbook.py duplicates code from _get_data_path() __init__.py
• #3631: Better document meaning of notches in boxplots
• #6705: The test suite spends 20% of it’s time in gc.collect()
• #6698: Axes3D scatter crashes without alpha keyword
• #5860: Computer Modern Roman should be the default serif when using TeX backend
• #6702: Bad fonts crashes matplotlib on startup
• #6671: Issue plotting big endian images
• #6196: Qt properties editor discards color alpha
• #6509: pylab image_masked is broken
• #6657: appveyor is failing on pre-install
• #6610: Icons for Tk are not antialiased.
• #6687: Small issues with the example polar_scatter_demo.py
• #6541: Time to deprecate the GTK backend
• #6680: Minor typo in the docstring of IdentityTransform?
• #6670: plt.text object updating incorrectly with blit=False
• #6646: Incorrect fill_between chart when use set_xscale(‘log’)
• #6540: imshow(..., alpha=0.5) produces different results in 2.x
• #6650: fill_between() not working properly
• #6566: Regression: Path.contains_points now returns uint instead of bool
• #6624: bus error: fc-list
• #6655: Malware found on matplotlib components
• #6623: RectangleSelector disappears after resizing
• #6629: matplotlib version error
• #6638: get_ticklabels returns ‘’ in ipython/python interpreter
• #6631: can’t build matplotlib on smartos system(open solaris)
• #6562: 2.x: Cairo backends cannot render images
• #6507: custom scatter marker demo broken
• #6591: DOC: update static image for interpolation_none_vs_nearest.py example
• #6607: BUG: saving image to png changes colors
• #6587: please copy http://matplotlib.org/devdocs/users/colormaps.html to http://matplotlib.org/users
• #6594: Documentation Typo
• #5784: dynamic ticking (#5588) should avoid (if possible) single ticks
• #6492: mpl_toolkits.mplot3d has a null byte somewhere
• #5862: Some Microsoft fonts produce unreadable EPS
• #6537: bundled six 1.9.0 causes ImportError: No module named ‘winreg’ in Pympler
• #6563: pyplot.errorbar attempts to plot 0 on a log axis in SVGs
• #6571: Unexpected behavior with ttk.Notebook - graph not loaded unless tab preselected
• #6570: Unexpected behavior with ttk.Notebook - graph not loaded unless tab preselected
• #6539: network tests are not skipped when running tests.py with –no-network
• #6567: qt_compat fails to identify PyQt5
• #6559: mpl 1.5.1 requires pyqt even with a wx backend
• #6009: No space before unit symbol when there is no SI prefix in ticker.EngFormatter
• #6528: Fail to install matplotlib by “pip install” on SmartOS(like open solaris system)
• #6531: Segmentation fault with any backend (matplotlib 1.4.3 and 1.5.1) when calling pyplot.show()
• #6513: Using gray shade from string ignores alpha parameters
• #6477: Savefig() to pdf renders markers differently than show()
• #6525: PS export issue with custom font
• #6514: LaTeX axis labels can no longer have custom fonts
• #2663: Multi Cursor disable broken
• #6083: Figure linewidth default in rcparams
• #1069: Add a donation information page
• #6035: Issue(?): head size of FancyArrowPatch changes between interactive figure and picture export
• #6495: new figsize is bad for subplots with fontsize 12
• #6493: Stepfilled color cycle for background and edge different
• #6380: Implicit addition of “color” to property_cycle breaks semantics
• #6447: Line2D.contains does not take drawstyle into account.
• #6257: option for default space between title and axes
• #5868: tight_layout doesn’t leave enough space between outwards ticks and axes title
• #5987: Outward ticks cause labels to be clipped by default
• #5269: Default changes: legend
• #6489: Test errors with numpy 1.11.1rc1
• #5960: Misplaced shadows when using FilteredArtistList
• #6452: Please add a generic “seaborn” style
• #6469: Test failures testing matplotlib 1.5.1 manylinux wheels
• #5854: New cycler does not work with bar plots
• #5977: legend needs logic to deal with new linestyle scaling by linewidth
• #6365: Default format time series xtick labels changed
• #6104: docs: latex required for PDF plotting?
• #6451: Inequality error on web page http://matplotlib.org/faq/howto_faq.html
• #6459: use conda already installed on appveyor
• #6043: Advanced hillshading example looks strange with new defaults.
• #6440: BUG: set_tick_params labelcolor should apply to offset
• #6458: Wrong package name in INSTALL file
• #2842: matplotlib.tests.test_basic.test_override_builtins() fails with Python >=3.4
• #2375: matplotlib 1.3.0 doesn’t compile with Solaris Studio 12.1 CC
• #2667: matplotlib.tests.test_mathtext.test_mathtext_{cm,stix,sticksans}_{37,53}.test are failing
• #2243: axes limits with aspect='equal'
• #1758: y limit with dashed or dotted lines hangs with somewhat big data
• #5994: Points annotation coords not working in 2.x
• #6444: matplotlib.path.contains_points is a LOT slower in 1.51
• #5461: Feature request: allow a default line alpha to be set in mpl.rcParams
• #5132: ENH: Set the alpha value for plots in rcParams
• #6449: axhline and axvline linestyle as on-off seq doesn’t work if set directly in function call
• #6416: animation with ‘ffmpeg’ backend and ‘savefig.bbox = tight’ garbles video
• #6437: Improperly spaced time axis
• #5974: scatter is not changing color in Axes3D
• #6436: clabels plotting outside of projection limb
• #6438: Cant get emoji working in Pie chart legend with google app engine. Need help.
• #6362: greyscale scatter points appearing blue
• #6301: tricky bug in ticker due to special behaviour of numpy
• #6276: Ticklabel format not preserved after editing plot limits
• #6173: linestyle parameter does not support default cycler through None, crashes instead.
• #6109: colorbar _ticker + _locate bug
• #6231: Segfault when figures are deleted in random order
• #6432: micro sign doesn’t show in EngFormatter
• #6057: Infinite Loop: LogLocator Colorbar & update_ticks
• #6270: pyplot.contour() not working with matplotlib.ticker.LinearLocator()
• #6058: “Configure subplots” tool is initialized very inefficiently in the Qt backends
• #6363: Change legend to accept alpha instead of (only) framealpha.
• #6394: Severe bug in `imshow` when plotting images with small values
• #6368: Bug: matplotlib.pyplot.spy: does not work correctly for sparse matrices with many entries (>= 2**32)
• #6419: Imshow does not copy data array but determines colormap values upon call
• #3615: mouse scroll event in Gtk3 backend
• #3373: add link to gtk embedding cookbook to website
• #6121: opening the configure subplots menu moves the axes by a tiny amount
• #2511: NavigationToolbar breaks if axes are added during use.
• #6349: Down arrow on GTK3 backends selects toolbar, which eats further keypress events
• #6408: minor ticks don’t respect rcParam xtick.top / ytick.right
• #6398: sudden install error with pip (pyparsing 2.1.2 related)
• #5819: 1.5.1rc1: dont use absolute links in the “new updates” on the homepage
• #5969: urgent bug after 1.5.0: offset of LineCollection when apply agg_filter
• #5767: axes limits (in old “round_numbers” mode) affected by floating point issues
• #5755: Better choice of axes offset value
• #5938: possible bug with ax.set_yscale(‘log’) when all values in array are zero
• #6399: pyparsing version 2.1.2 not supported (2.1.1 works though)
• #5884: numpy as no Attribute string
• #6395: Deprecation warning for axes.color_cycle
• #6385: Possible division by zero in new get_tick_space() methods; is rotation ignored?
• #6344: Installation issue
• #6315: Qt properties editor could sort lines labels using natsort
• #5219: Notebook backend: possible to remove javascript/html when figure is closed?
• #5111: nbagg backend captures exceptions raised by callbacks
• #4940: NBAgg figure management issues
• #4582: Matplotlib IPython Widget
• #6142: matplotlib.ticker.LinearLocator view_limits algorithm improvement?
• #6326: Unicode invisible after image saved
• #5980: Gridlines on top of plot by default in 2.0?
• #6272: Ability to set default scatter marker in matplotlibrc
• #6335: subplots animation example is broken on OS X with qt4agg
• #6357: pyplot.hist: normalization fails
• #6352: clim doesn’t update after draw
• #6353: hist won’t norm for small numbers
• #6343: prop_cycle breaks keyword aliases
• #6226: Issue saving figure as eps when using gouraud shaded triangulation
• #6330: ticklabel_format reset to default by ScalarFormatter
• #4975: Non-default color_cycle not working in Pie plot
• #5990: Scatter markers do not follow new colour cycle
• #5577: Handling of “next color in cycle” should be handled differently
• #5489: Special color names to pull colors from the currently active color cycle
• #6325: Master requires cycler 0.10.0
• #6278: imshow with pgf backend does not render transparency
• #5945: Figures in the notebook backend are too large following DPI changes
• #6332: Animation with blit broken
• #6331: matplotlib pcolormesh seems to slide some data around on the plot
• #6307: Seaborn style sheets don’t edit patch.facecolor
• #6294: Zero size ticks show up as single pixels in rendered pdf
• #6318: Cannot import mpl_toolkits in Python3
• #6316: Viridis exists but not in plt.cm.datad.keys()
• #6082: Cannot interactively edit axes limits using Qt5 backend
• #6309: Make CheckButtons based on subplots automatically
• #6306: Can’t show images when plt.show() was executed
• #2527: Vertical alignment of text is too high
• #4827: Pickled Figure Loses sharedx Properties
• #5998: math??{} font styles are ignored in 2.x
• #6293: matplotlib notebook magic cells with output plots - skips next cell for computation
• #235: hatch linewidth patch
• #5875: Manual linestyle specification ignored if ‘prop_cycle’ contains ‘ls’
• #5959: imshow rendering issue
• #6237: MacOSX agg version: doesn’t redraw after keymap.grid keypress
• #6266: Better fallback when color is a float
• #6002: Potential bug with ‘start_points’ argument of ‘pyplot.streamplot’
• #6265: Document how to set viridis as default colormap in mpl 1.x
• #6258: Rendering vector graphics: parsing polygons?
• #1702: Bug in 3D histogram documentation
• #5937: xticks/yticks default behaviour
• #4706: Documentation - Basemap
• #6255: Can’t build matplotlib.ft2font in cygwin
• #5792: Not easy to get colorbar tick mark locations
• #6233: ImportError from Sphinx plot_directive from Cython
• #6235: Issue with building docs with Sphinx 1.4.0
• #4383: xkcd color names
• #6219: Example embedding_in_tk.py freezes in Python3.5.1
• #5067: improve whats_new entry for prop cycler
• #4614: Followup items from the matplotlib 2.0 BoF
• #5986: mac osx backend does not scale dashes by linewidth
• #4680: Set forward=True by default when setting the figure size
• #4597: use mkdtemp in _create_tmp_config_dir
• #3437: Interactive save should respect ‘savefig.facecolor’ rcParam.
• #2467: Improve default colors and layouts
• #4194: matplotlib crashes on OS X when saving to JPEG and then displaying the plot
• #4320: Pyplot.imshow() “None” interpolation is not supported on Mac OSX
• #1266: Draggable legend results RuntimeError and AttributeError on Mac OS 10.8.1
• #5442: xkcd plots rendered as regular plots on Mac OS X
• #2697: Path snapping does not respect quantization scale appropriate for Retina displays
• #6049: Incorrect TextPath display under interactive mode
• #1319: macOSx backend lacks support for cursor-type widgets
• #531: macosx backend does not work with blitting
• #5964: slow rendering with backend_macosx on El Capitan
• #5847: macosx backend color rendering
• #6224: References to non-existing class FancyBoxPatch
• #781: macosx backend doesn’t find fonts the same way as other backends
• #4271: general colormap reverser
• #6201: examples svg_histogram.html failes with UnicodeEncodeError
• #6212: ENH? BUG? pyplot.setp/Artist.setp does not accept non-indexable iterables of handles.
• #4445: Two issues with the axes offset indicator
• #6209: Qt4 backend uses Qt5 backend
• #6136: Feature request: configure default scatter plot marker size
• #6180: Minor typos in the style sheets users’ guide
• #5517: “interactive example” not working with PySide
• #4607: bug in font_manager.FontManager.score_family()
• #4400: Setting annotation background covers arrow
• #596: Add “bring window to front” functionality
- #4674: Default marker edge width in plot vs. scatter
- #5988: rainbow_text example is missing some text
- #6165: MacOSX backend hangs drawing lines with many dashes/dots
- #6155: Deprecation warnings with Dateutil 2.5
- #6003: In ‘pyplot.streamplot’, starting points near the same streamline raise ‘InvalidIndexError’
- #6105: Accepting figure argument in subplot2grid
- #6184: csv2rec handles dates differently to datetimes when datefirst is specified.
- #6164: Unable to use PySide with gui=qt
- #6166: legends do not refresh
- #3897: bug: inconsistent types accepted in DateLocator subclasses
- #6160: EPS issues with rc parameters used in seaborn library on Win 8.1
- #6163: Can’t make matplotlib run in my computer
- #5331: Boxplot with zero IQR sets whiskers to max and min and leaves no outliers
- #5575: plot_date() ignores timezone
- #6143: drawstyle accepts anything as default rather than raising
- #6151: Matplotlib 1.5.1 ignores annotation_clip parameter
- #6147: colormaps issue
- #5916: Headless get_window_extent or equivalent
- #6141: Matplotlib subplots and datetime x-axis functionality not working as intended?
- #6138: No figure shows, no error
- #6134: Cannot plot a line of width=1 without antialiased
- #6120: v2.x failures on travis
- #6092: %matplotlib notebook broken with current matplotlib master
- #1235: Legend placement bug
- #2499: Showing np.uint16 images of the form (h,w,3) is broken
- #5479: Table: auto_set_column_width not working
- #6028: Appearance of non-math hyphen changes with math in text
- #6113: ValueError after moving legend and rcParams.update
- #6111: patches fails when data are array, not list
- #6108: Plot update issue within event callback for multiple updates
- #6069: imshow no longer correctly handles ‘bad’ (nan) values
- #6103: ticklabels empty when not interactive
• #6084: Despined figure is cropped
• #6067: pyplot.savefig doesn’t expand ~ (tilde) in path
• #4754: Change default color cycle
• #6063: Axes.relim() seems not to work when copying Line2D objects
• #6065: Proposal to change color – ‘indianred’
• #6056: quiver plot in polar projection - how to make the quiver density latitude-dependent?
• #6051: Matplotlib v1.5.1 apparently not compatible with python-dateutil 2.4.2
• #5513: Call get_backend in pylab_setup
• #5983: Option to Compress Graphs for pgf-backend
• #5895: Polar Projection PDF Issue
• #5948: tilted line visible in generated pdf file
• #5737: matplotlib 1.5 compatibility with wxPython
• #5645: Missing line in a self-sufficient example in navigation_toolbar.rst :: a minor bug in docs
• #6037: Matplotlib xtick appends .%f after %H:%M%:%S on chart
• #6025: Exception in Tkinter/to_rgb with new colormaps
• #6034: colormap name is broken for ListedColormap?
• #5982: Styles need update after default style changes
• #6017: Include tests.py in archive of release
• #5520: ‘nearest’ interpolation not working with low dpi
• #4280: imsave reduces 1 row from the image
• #3057: DPI-connected bug of imshow when using multiple masked arrays
• #5490: Don’t interpolate images in RGB space
• #5996: 2.x: Figure.add_axes(..., facecolor='color') does not set axis background colour
• #4760: Default linewidth thicker than axes linewidth
• #2698: ax.text() fails to draw a box if the text content is full of blank spaces and linefeeds.
• #3948: a weird thing in the source code comments
• #5921: test_backend.pgf.check_for(texsystem) does not do what it says...
• #4295: Draggable annotation position wrong with negative x/y
• #1986: Importing pyplot messes with command line argument parsing
• #5885: matplotlib stepfilled histogram breaks at the value 10^-1 on xubuntu
• #5050: pandas v0.17.0rc1
• #3658: axes.locator_params fails with LogLocator (and most Locator subclasses)
• #3742: Square plots
• #3900: debugging Segmentation fault with Qt5 backend
• #4192: Error when color value is None
• #4210: segfault: fill_between with Python3
• #4325: FancyBboxPatch wrong size
• #4340: Histogram gap artifacts
• #5096: Add xtick.top.visible, xtick.bottom.visible, ytick.left.visible, ytick.right.visible to rcParams
• #5120: custom axis scale doesn’t work in 1.4.3
• #5212: shifted(?) bin positions when plotting multiple histograms at the same time
• #5293: Qt4Agg: RuntimeError: call __init__ twice
• #5971: Add support for PySide2 (Qt5)
• #5993: Basemap readshapefile should read shapefile for the long/lat specified in the Basemap instance.
• #5991: basemap crashes with no error message when passed numpy nan’s
• #5883: New colormaps: Inferno, Viridis, ...
• #5841: extra label for non-existent tick
• #4502: Default style proposal: outward tick marks
• #875: Replace “jet” as the default colormap
• #5047: Don’t draw end caps on error bars by default
• #4700: Overlay blend mode
• #4671: Change default legend location to ‘best’.
• #5419: Default setting of figure transparency in NbAgg is a performance problem
• #4815: Set default axis limits in 2D-plots to the limits of the data
• #4854: set numpoints to 1
• #5917: improved dash styles
• #5900: Incorrect Image Tutorial Inline Sample Code
• #5965: xkcd example in gallery
• #5616: Better error message if no animation writer is available
• #5920: How to rotate secondary y axis label so it doesn’t overlap with y-ticks, matplotlib
• #5966: SEGFAULT if pyplot is imported
• #5967: savefig SVG and PDF output for scatter plots is excessively complex, crashes Inkscape
• #1943: legend doesn’t work with stackplot
• #5923: Windows usetex=True error in long usernames
• #5940: KeyError: ‘getpwuid(): uid not found: 5001’
• #5748: Windows test failures on appveyor
• #5944: Notebook backend broken on Master
• #5946: Calling subplots_adjust breaks savefig output
• #5929: Fallback font doesn’t work on windows?
• #5925: Data points beyond axes range plotted when saved to SVG
• #5918: Pyplot.savefig is very slow with some combinations of data/ylim scales
• #5919: Error when trying to import matplotlib into IPython notebook
• #5803: Barbs broken
• #5846: setupext.py: problems parsing setup.cfg (not updated to changes in configparser)
• #5309: Differences between function and keywords for savefig.bbox and axes.facecolor
• #5889: Factual errors in HowTo FAQ Box Plot Image
• #5618: New rcParams requests
• #5810: Regression in test_remove_shared_axes
• #5281: plt.tight_layout(pad=0) cuts away outer ticks
• #5909: The documentation for LinearLocator’s presets keyword is unclear
• #5864: mathtext mishandling of certain exponents
• #5869: doc build fails with mpl-1.5.1 and sphinx-1.3.4 (sphinx-1.3.3 is fine)
• #5835: gridspec.Gridspec doesn’t check for consistency in arguments
• #5867: No transparency in *.pgf file when using pgf Backend.
• #5863: left(... right) are too small
• #5850: prop_cycler for custom dashes – linestyle such as (<offset>, (<on>, <off>)) throws error
• #5861: Marker style request
• #5851: Bar and box plots use the ‘default’ matplotlib colormap, even if the style is changed
• #5857: FAIL: matplotlib.tests.test_coding_standards.test_pep8_conformance_examples
• #5831: tests.py is missing from pypi tarball
• #5829: test_rasterize_dpi fails with 1.5.1
• #5843: what is the source code of ax.pcolormesh(T, R, Z,vmin=0,vmax=255,cmap=’jet’) ?
• #5799: mathtext kerning around comma
• #2841: There is no set_linestyle_cycle in the matplotlib axes API
• #5821: Consider using an offline copy of Raleway font
• #5822: FuncAnimation.save() only saving 1 frame
• #5449: Incomplete dependency list for installation from source
• #5793: GTK backends
• #5814: Adding colorbars to row subplots doesn’t render the main plots when saving to .eps in 1.5.0
• #5816: matplotlib.pyplot.boxplot ignored showmeans keyword
• #5086: Default date format for axis formatting
• #4808: AutoDateFormatter shows too much precision
• #5812: Widget event issue
• #5794: –no-network not recognized as valid option for tests.py
• #5801: No such file or directory: ‘/usr/share/mplstylelib’
• #5777: Using default style raises warnings about non style parameters
• #5738: Offset text should be computed based on lowest and highest ticks, not actual axes limits
• #5403: Document minimal MovieWriter sub-class
• #5558: The link to the John Hunter Memorial fund is a 404
• #5757: Several axes_grid1 and axisartist examples broken on master
• #5557: plt.hist throws KeyError when passed a pandas.Series without 0 in index
• #5550: Plotting datetime values from Pandas dataframe
• #4855: Limit what style.use can affect?
• #5765: import matplotlib._png as _png ImportError: libpng16.so.16: cannot open shared object
• #5753: Handling of zero in log shared axes depends on whether axes are shared
• #5756: 3D rendering, scatterpoints disappear near edges of surfaces
• #5747: Figure.suptitle does not respect size argument
• #5641: plt.errorbar error with empty list
• #5476: annotate doesn’t trigger redraw
• #5572: Matplotlib 1.5 broken_barh fails on empty data.
• #5089: axes.properties calls get_axes internally
• #5745: Using internal qhull despite the presence of pyqhull installed in the system
• #5744: cycler is required, is missing, yet build succeeds.
• #5592: Problem with _init_func in ArtistAnimation
• #5729: Test matplotlib.tests.test_backend_svg.test_determinism fails on OSX in virtual envs.
• #4756: font_manager.py takes multiple seconds to import
• #5435: Unable to upgrade matplotlib 1.5.0 through pip
• #5636: Generating legend from figure options panel of qt backend raise exception for large number of plots
• #5365: Warning in test_lines.test_nan_is_sorted
• #5646: Version the font cache
• #5692: Can’t remove StemContainer
• #5635: RectangleSelector creates not wanted lines in axes
• #5427: BUG? Normalize modifies pandas Series inplace
• #5693: Invalid caching of long lines with nans
• #5705: doc/users/plotting/examples/axes_zoom_effect.py is not a Python file
• #4359: savefig crashes with malloc error on os x
• #5715: Minor error in set up fork
• #5687: Segfault on plotting with PySide as backend.qt4
• #5708: Segfault with Qt4Agg backend in 1.5.0
• #5704: Issue with xy and xytex
• #5673: colorbar labelling bug (1.5 regression)
• #4491: Document how to get a framework build in a virtual env
• #5468: axes selection in axes editor
• #5684: AxesGrid demo exception with LogNorm: ‘XAxis’ object has no attribute ‘set_scale’
• #5663: AttributeError: ‘NoneType’ object has no attribute ‘canvas’
• #5573: Support HiDPI (retina) displays in docs
• #5680: SpanSelector span_stays fails with use_blit=True
• #5679: Y-axis switches to log scale when an X-axis is shared multiple times.
• #5655: Problems installing basemap behind a proxy
• #5670: Doubling of coordinates in polygon clipping
• #4725: change set_adjustable for share axes with aspect ratio of 1
• #5488: The default number of ticks should be based on the length of the axis
• #5543: num2date ignoring tz in v1.5.0
• #305: Change canvas.print_figure default resolution
• #5660: Cannot raise FileNotFoundError in python2
• #5658: A way to remove the image of plt.figimage()?
• #5495: Something fishy in png reading
• #5549: test_streamplot: test_colormap test broke unintentionally
- #5381: HiDPI support in Notebook backend
- #5531: test_mplot3d:test_quiver3d broke unintentionally
- #5530: test_axes:test_polar_unit broke unintentionally
- #5525: Comparison failure in text_axes:test_phase_spectrum_freqs
- #5650: Wrong backend selection with PyQt4
- #5649: Documentation metadata (release version) does not correspond with some of the ‘younger’ documentation content
- #5648: Some tests require non-zero tolerance
- #3980: zoom in wx with retina behaves badly
- #5642: Mistype in pyplot_scales.py of pyplot_tutorial.rst :: a minor bug in docs
- #3316: wx crashes on exit if figure not shown and not explicitly closed
- #5624: Cannot manually close matplotlib plot window in Mac OS X Yosemite
- #4891: Better auto-selection of axis limits
- #5633: No module named externals
- #5634: No module named ‘matplotlib.tests’
- #5473: Strange OS warning when import pyplot after upgrading to 1.5.0
- #5524: Change in colorbar extensions
- #5627: Followup for Windows CI stuff
- #5613: Quiverkey() positions arrow incorrectly with labelpos ‘N’ or ‘S’
- #5615: tornado now a requirement?
- #5582: FuncAnimation crashes the interpreter (win7, 64bit)
- #5610: Testfailures on windows
- #5595: automatically build windows conda packages and wheels in master
- #5535: test_axes:test_rc_grid image comparison test has always been broken
- #4396: Qt5 is not mentioned in backends list in doc
- #5205: pcolor does not handle non-array C data
- #4839: float repr in axes parameter editing window (aka the green tick button)
- #5542: Bad superscript positioning for some fonts
- #3791: Update colormap examples.
- #4679: Relationship between line-art markers and the markeredgewidth parameter
- #5601: Scipy/matplotlib recipe with plt.connect() has trouble in python 3 (AnnoteFinder)
- #4211: Axes3D quiver: variable length arrows
• #773: mplot3d enhancement
• #395: need 3D examples for tricontour and tricontourf
• #186: Axes3D with PolyCollection broken
• #178: Incorrect mplot3d contourf rendering
• #5508: Animation.to_html5_video requires python3 base64 module
• #5576: Improper reliance upon pkg-config when C_INCLUDE_PATH is set
• #5369: Change in zorder of streamplot between 1.3.1 and 1.4.0
• #5569: Stackplot does not handle NaNs
• #5565: label keyword is not interpreted properly in errorbar() for pandas.DataFrame-like objects
• #5561: interactive mode doesn’t display images with standard python interpreter
• #5559: Setting window titles when in interactive mode
• #5554: Cropping text to axes
• #5545: EllipseCollection renders incorrectly when passed a sequence of widths
• #5475: artist picker tolerance has no effect
• #5529: Wrong image/code for legend_demo (pylab)
• #5139: plt.subplots for already existing Figure
• #5497: violin{,plot} return value
• #5441: boxplot rcParams are not in matplotlibrc.template
• #5522: axhline fails on custom scale example
• #5528: $\rho$ in text for plots erroring
• #4799: Probability axes scales
• #5487: Trouble importing image_comparison decorator in v1.5
• #5464: figaspect not working with numpy floats
• #4487: Should default hist() bins be changed in 2.0?
• #5499: UnicodeDecodeError in IPython Notebook caused by negative numbers in plt.legend()
• #5498: Labels’ collisions while plotting named DataFrame iterrows
• #5491: clippedline.py example should be removed
• #5482: RuntimeError: could not open display
• #5481: value error : unknown locale: UTF-8
• #4780: Non-interactive backend calls draw more than 100 times
• #5470: colorbar values could take advantage of offsetting and/or scientific notation
• #5471: FuncAnimation video saving results in one frame file
• #5457: Example of new colormaps is misleading
• #3920: Please fix pip install, so that plt.show() etc works correctly
• #5418: install backend gtk in Cygwin
• #5368: New axes.set_prop_cycle method cannot handle any generic iterable
• #5446: Tests fail to run (killed manually after 7000 sec)
• #5225: Rare race condition in makedirs with parallel processes
• #5444: overline and subscripts/superscripts in mathtext
• #4859: Call tight_layout() by default
• #5429: Segfault in matplotlib.tests.test_image:test_get_window_extent_for_AxisImage on python3.5
• #5431: Matplotlib 1.4.3 broken on Windows
• #5409: Match zdata cursor display scalling with colorbar?
• #5128: ENH: Better default font
• #5420: [Mac OS X 10.10.5] Macports install error :unknown locale: UTF-8
• #3867: OSX compile broken since CXX removal (conda only?)
• #5411: XKCD style fails except for inline mode
• #5406: Hangs on OS X 10.11.1: No such file or directory: ‘~/.matplotlib/fontList.json’
• #3116: mplot3d: argument checking in plot_surface should be improved.
• #347: Faster Text drawing needed
• #5399: FuncAnimation w/o init_func breaks when saving
• #5395: Style changes doc has optimistic release date
• #5393: wrong legend in errorbar plot for pandas series
• #5396: fill_between() with gradient
• #5221: infinite range for hist(histtype=“step”)’
• #4901: Error running double pendulum animation example
• #3314: assert mods.pop(0) == ‘tests’ errors for multiprocess tests on OSX
• #5337: Remove --nocapture from nosetests on .travis.yml?
• #5378: errorbar fails with pandas data frame
• #5367: histogram and digitize do not agree on the definition of a bin
• #5314: ValueError: insecure string pickle
• #5347: Problem with importing matplotlib.animation
• #4788: Modified axes patch will not re-clip artists
• #4968: Lasso-ing in WxAgg causes flickering of the entire figure
Matplotlib only uses BSD compatible code, and its license is based on the PSF license. See the Open Source Initiative licenses page for details on individual licenses. Non-BSD compatible licenses (e.g., LGPL) are acceptable in matplotlib toolkits. For a discussion of the motivations behind the licencing choice, see Licenses.

11.1 Copyright Policy

John Hunter began matplotlib around 2003. Since shortly before his passing in 2012, Michael Droettboom has been the lead maintainer of matplotlib, but, as has always been the case, matplotlib is the work of many.

Prior to July of 2013, and the 1.3.0 release, the copyright of the source code was held by John Hunter. As of July 2013, and the 1.3.0 release, matplotlib has moved to a shared copyright model.

matplotlib uses a shared copyright model. Each contributor maintains copyright over their contributions to matplotlib. But, it is important to note that these contributions are typically only changes to the repositories. Thus, the matplotlib source code, in its entirety, is not the copyright of any single person or institution. Instead, it is the collective copyright of the entire matplotlib Development Team. If individual contributors want to maintain a record of what changes/contributions they have specific copyright on, they should indicate their copyright in the commit message of the change, when they commit the change to one of the matplotlib repositories.

The Matplotlib Development Team is the set of all contributors to the matplotlib project. A full list can be obtained from the git version control logs.

11.2 License agreement for matplotlib 2.0.2

1. This LICENSE AGREEMENT is between the Matplotlib Development Team (“MDT”), and the Individual or Organization (“Licensee”) accessing and otherwise using matplotlib software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, MDT hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use matplotlib 2.0.2 alone or in any derivative version, provided, however, that MDT’s License Agreement and MDT’s notice of copyright, i.e., “Copyright (c) 2012-2013	 
Matplotlib Development Team; All Rights Reserved” are retained in matplotlib 2.0.2 alone or in any derivative version prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or incorporates matplotlib 2.0.2 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to matplotlib 2.0.2.

4. MDT is making matplotlib 2.0.2 available to Licensee on an “AS IS” basis. MDT MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, MDT MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF MATPLOTLIB 2.0.2 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. MDT SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF MATPLOTLIB 2.0.2 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING MATPLOTLIB 2.0.2, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between MDT and Licensee. This License Agreement does not grant permission to use MDT trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8. By copying, installing or otherwise using matplotlib 2.0.2, Licensee agrees to be bound by the terms and conditions of this License Agreement.

11.3 License agreement for matplotlib versions prior to 1.3.0

1. This LICENSE AGREEMENT is between John D. Hunter (“JDH”), and the Individual or Organization (“Licensee”) accessing and otherwise using matplotlib software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, JDH hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use matplotlib 2.0.2 alone or in any derivative version, provided, however, that JDH’s License Agreement and JDH’s notice of copyright, i.e., “Copyright (c) 2002-2009 John D. Hunter; All Rights Reserved” are retained in matplotlib 2.0.2 alone or in any derivative version prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or incorporates matplotlib 2.0.2 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to matplotlib 2.0.2.

4. JDH is making matplotlib 2.0.2 available to Licensee on an “AS IS” basis. JDH MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, JDH MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF MATPLOTLIB 2.0.2 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. JDH SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF MATPLOTLIB 2.0.2 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING MATPLOTLIB 2.0.2, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between JDH and Licensee. This License Agreement does not grant permission to use JDH trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8. By copying, installing or otherwise using matplotlib 2.0.2, Licensee agrees to be bound by the terms and conditions of this License Agreement.
Matplotlib was written by John D. Hunter, with contributions from an ever-increasing number of users and developers. The current co-lead developers are Michael Droettboom and Thomas A. Caswell; they are assisted by many active developers.

The following is a list of contributors extracted from the git revision control history of the project:

Some earlier contributors not included above are (with apologies to any we have missed):


We also thank all who have reported bugs, commented on proposed changes, or otherwise contributed to Matplotlib’s development and usefulness.

422 Chapter 12. Credits
Part II

The Matplotlib API
### PLOTTING COMMANDS SUMMARY

```python
matplotlib.pyplot.plotting()
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>acorr</code></td>
<td>Plot the autocorrelation of x.</td>
</tr>
<tr>
<td><code>angle_spectrum</code></td>
<td>Plot the angle spectrum.</td>
</tr>
<tr>
<td><code>annotate</code></td>
<td>Annotate the point xy with text s.</td>
</tr>
<tr>
<td><code>arrow</code></td>
<td>Add an arrow to the axes.</td>
</tr>
<tr>
<td><code>autoscale</code></td>
<td>Autoscale the axis view to the data (toggle).</td>
</tr>
<tr>
<td><code>axes</code></td>
<td>Add an axes to the figure.</td>
</tr>
<tr>
<td><code>axhline</code></td>
<td>Add a horizontal line across the axis.</td>
</tr>
<tr>
<td><code>axhspan</code></td>
<td>Add a horizontal span (rectangle) across the axis.</td>
</tr>
<tr>
<td><code>axis</code></td>
<td>Convenience method to get or set axis properties.</td>
</tr>
<tr>
<td><code>axvline</code></td>
<td>Add a vertical line across the axes.</td>
</tr>
<tr>
<td><code>axvspan</code></td>
<td>Add a vertical span (rectangle) across the axes.</td>
</tr>
<tr>
<td><code>bar</code></td>
<td>Make a bar plot.</td>
</tr>
<tr>
<td><code>barbs</code></td>
<td>Plot a 2-D field of barbs.</td>
</tr>
<tr>
<td><code>barh</code></td>
<td>Make a horizontal bar plot.</td>
</tr>
<tr>
<td><code>box</code></td>
<td>Turn the axes box on or off.</td>
</tr>
<tr>
<td><code>boxplot</code></td>
<td>Make a box and whisker plot.</td>
</tr>
<tr>
<td><code>broken_barh</code></td>
<td>Plot horizontal bars.</td>
</tr>
<tr>
<td><code>cla</code></td>
<td>Clear the current axes.</td>
</tr>
<tr>
<td><code>clabel</code></td>
<td>Label a contour plot.</td>
</tr>
<tr>
<td><code>clf</code></td>
<td>Clear the current figure.</td>
</tr>
<tr>
<td><code>clim</code></td>
<td>Set the color limits of the current image.</td>
</tr>
<tr>
<td><code>close</code></td>
<td>Close a figure window.</td>
</tr>
<tr>
<td><code>cohere</code></td>
<td>Plot the coherence between x and y.</td>
</tr>
<tr>
<td><code>colorbar</code></td>
<td>Add a colorbar to a plot.</td>
</tr>
<tr>
<td><code>contour</code></td>
<td>Plot contours.</td>
</tr>
<tr>
<td><code>contourf</code></td>
<td>Plot contours.</td>
</tr>
<tr>
<td><code>csd</code></td>
<td>Plot the cross-spectral density.</td>
</tr>
<tr>
<td><code>delaxes</code></td>
<td>Remove an axes from the current figure.</td>
</tr>
<tr>
<td><code>draw</code></td>
<td>Redraw the current figure.</td>
</tr>
<tr>
<td><code>errorbar</code></td>
<td>Plot an errorbar graph.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>eventplot</code></td>
<td>Plot identical parallel lines at specific positions.</td>
</tr>
<tr>
<td><code>figimage</code></td>
<td>Adds a non-resampled image to the figure.</td>
</tr>
<tr>
<td><code>figlegend</code></td>
<td>Place a legend in the figure.</td>
</tr>
<tr>
<td><code>fignum_exists</code></td>
<td>Add text to figure.</td>
</tr>
<tr>
<td><code>figtext</code></td>
<td>Creates a new figure.</td>
</tr>
<tr>
<td><code>fill</code></td>
<td>Plot filled polygons.</td>
</tr>
<tr>
<td><code>fill_between</code></td>
<td>Make filled polygons between two curves.</td>
</tr>
<tr>
<td><code>fill_betweenx</code></td>
<td>Make filled polygons between two horizontal curves.</td>
</tr>
<tr>
<td><code>findobj</code></td>
<td>Find artist objects.</td>
</tr>
<tr>
<td><code>gca</code></td>
<td>Get the current <code>Axes</code> instance on the current figure matching the given keyword args, or create one.</td>
</tr>
<tr>
<td><code>gcf</code></td>
<td>Get a reference to the current figure.</td>
</tr>
<tr>
<td><code>gci</code></td>
<td>Get the current colorable artist.</td>
</tr>
<tr>
<td><code>get_fignums</code></td>
<td>Return a list of existing figure numbers.</td>
</tr>
<tr>
<td><code>get_figlabels</code></td>
<td>Return a list of existing figure labels.</td>
</tr>
<tr>
<td><code>grid</code></td>
<td>Turn the axes grids on or off.</td>
</tr>
<tr>
<td><code>hexbin</code></td>
<td>Make a hexagonal binning plot.</td>
</tr>
<tr>
<td><code>hist</code></td>
<td>Plot a histogram.</td>
</tr>
<tr>
<td><code>hist2d</code></td>
<td>Make a 2D histogram plot.</td>
</tr>
<tr>
<td><code>hlines</code></td>
<td>Plot horizontal lines at each y from xmin to xmax.</td>
</tr>
<tr>
<td><code>hold</code></td>
<td></td>
</tr>
<tr>
<td><code>imread</code></td>
<td>Read an image from a file into an array.</td>
</tr>
<tr>
<td><code>imsave</code></td>
<td>Save an array as in image file.</td>
</tr>
<tr>
<td><code>imshow</code></td>
<td>Display an image on the axes.</td>
</tr>
<tr>
<td><code>install_repl_displayhook</code></td>
<td>Install a repl display hook so that any stale figure are automatically redrawn when control is returned to the repl.</td>
</tr>
<tr>
<td><code>ioff</code></td>
<td>Turn interactive mode off.</td>
</tr>
<tr>
<td><code>ion</code></td>
<td>Turn interactive mode on.</td>
</tr>
<tr>
<td><code>ishold</code></td>
<td></td>
</tr>
<tr>
<td><code>isinteractive</code></td>
<td>Return status of interactive mode.</td>
</tr>
<tr>
<td><code>legend</code></td>
<td>Places a legend on the axes.</td>
</tr>
<tr>
<td><code>locator_params</code></td>
<td>Control behavior of tick locators.</td>
</tr>
<tr>
<td><code>loglog</code></td>
<td>Make a plot with log scaling on both the x and y axis.</td>
</tr>
<tr>
<td><code>magnitude_spectrum</code></td>
<td>Plot the magnitude spectrum.</td>
</tr>
<tr>
<td><code>margins</code></td>
<td>Set or retrieve autoscaling margins.</td>
</tr>
<tr>
<td><code>matshow</code></td>
<td>Display an array as a matrix in a new figure window.</td>
</tr>
<tr>
<td><code>minorticks_off</code></td>
<td>Remove minor ticks from the current plot.</td>
</tr>
<tr>
<td><code>minorticks_on</code></td>
<td>Display minor ticks on the current plot.</td>
</tr>
<tr>
<td><code>over</code></td>
<td></td>
</tr>
<tr>
<td><code>pause</code></td>
<td>Pause for <code>interval</code> seconds.</td>
</tr>
<tr>
<td><code>pcolor</code></td>
<td>Create a pseudocolor plot of a 2-D array.</td>
</tr>
<tr>
<td><code>pcolormesh</code></td>
<td>Plot a quadrilateral mesh.</td>
</tr>
<tr>
<td><code>phase_spectrum</code></td>
<td>Plot the phase spectrum.</td>
</tr>
<tr>
<td><code>pie</code></td>
<td>Plot a pie chart.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>plot</code></td>
<td>Plot lines and/or markers to the <em>Axes</em>.</td>
</tr>
<tr>
<td><code>plot_date</code></td>
<td>A plot with data that contains dates.</td>
</tr>
<tr>
<td><code>plotfile</code></td>
<td>Plot the data in in a file.</td>
</tr>
<tr>
<td><code>polar</code></td>
<td>Make a polar plot.</td>
</tr>
<tr>
<td><code>psd</code></td>
<td>Plot the power spectral density.</td>
</tr>
<tr>
<td><code>quiver</code></td>
<td>Plot a 2-D field of arrows.</td>
</tr>
<tr>
<td><code>quiverkey</code></td>
<td>Add a key to a quiver plot.</td>
</tr>
<tr>
<td><code>rc</code></td>
<td>Set the current rc params.</td>
</tr>
<tr>
<td><code>rc_context</code></td>
<td>Return a context manager for managing rc settings.</td>
</tr>
<tr>
<td><code>rcdefaults</code></td>
<td>Restore the rc params from Matplotlib’s internal defaults.</td>
</tr>
<tr>
<td><code>rgrids</code></td>
<td>Get or set the radial gridlines on a polar plot.</td>
</tr>
<tr>
<td><code>savefig</code></td>
<td>Save the current figure.</td>
</tr>
<tr>
<td><code>sca</code></td>
<td>Set the current Axes instance to <code>ax</code>.</td>
</tr>
<tr>
<td><code>scatter</code></td>
<td>Make a scatter plot of <code>x</code> vs <code>y</code> Marker size is scaled by <code>s</code> and marker color is mapped to <code>c</code>.</td>
</tr>
<tr>
<td><code>sci</code></td>
<td>Set the current image.</td>
</tr>
<tr>
<td><code>semilogx</code></td>
<td>Make a plot with log scaling on the x axis.</td>
</tr>
<tr>
<td><code>semilogy</code></td>
<td>Make a plot with log scaling on the y axis.</td>
</tr>
<tr>
<td><code>set_cmap</code></td>
<td>Set the default colormap.</td>
</tr>
<tr>
<td><code>setp</code></td>
<td>Set a property on an artist object.</td>
</tr>
<tr>
<td><code>show</code></td>
<td>Display a figure.</td>
</tr>
<tr>
<td><code>specgram</code></td>
<td>Plot a spectrogram.</td>
</tr>
<tr>
<td><code>spy</code></td>
<td>Plot the sparsity pattern on a 2-D array.</td>
</tr>
<tr>
<td><code>stackplot</code></td>
<td>Draws a stacked area plot.</td>
</tr>
<tr>
<td><code>stem</code></td>
<td>Create a stem plot.</td>
</tr>
<tr>
<td><code>step</code></td>
<td>Make a step plot.</td>
</tr>
<tr>
<td><code>streamplot</code></td>
<td>Draws streamlines of a vector flow.</td>
</tr>
<tr>
<td><code>subplot</code></td>
<td>Return a subplot axes positioned by the given grid definition.</td>
</tr>
<tr>
<td><code>subplot2grid</code></td>
<td>Create a subplot in a grid.</td>
</tr>
<tr>
<td><code>subplot_tool</code></td>
<td>Launch a subplot tool window for a figure.</td>
</tr>
<tr>
<td><code>subplots</code></td>
<td>Create a figure and a set of subplots This utility wrapper makes it convenient to create</td>
</tr>
<tr>
<td><code>subplots_adjust</code></td>
<td>Tune the subplot layout.</td>
</tr>
<tr>
<td><code>suptitle</code></td>
<td>Add a centered title to the figure.</td>
</tr>
<tr>
<td><code>switch_backend</code></td>
<td>Switch the default backend.</td>
</tr>
<tr>
<td><code>table</code></td>
<td>Add a table to the current axes.</td>
</tr>
<tr>
<td><code>text</code></td>
<td>Add text to the axes.</td>
</tr>
<tr>
<td><code>thetagrids</code></td>
<td>Get or set the theta locations of the gridlines in a polar plot.</td>
</tr>
<tr>
<td><code>tick_params</code></td>
<td>Change the appearance of ticks and tick labels.</td>
</tr>
<tr>
<td><code>ticklabel_format</code></td>
<td>Change the <em>ScalarFormatter</em> used by default for linear axes.</td>
</tr>
<tr>
<td><code>tight_layout</code></td>
<td>Automatically adjust subplot parameters to give specified padding.</td>
</tr>
<tr>
<td><code>title</code></td>
<td>Set a title of the current axes.</td>
</tr>
<tr>
<td><code>tricontour</code></td>
<td>Draw contours on an unstructured triangular grid.</td>
</tr>
<tr>
<td><code>tricontourf</code></td>
<td>Draw contours on an unstructured triangular grid.</td>
</tr>
<tr>
<td><code>tripcolor</code></td>
<td>Create a pseudocolor plot of an unstructured triangular grid.</td>
</tr>
</tbody>
</table>
Table 13.1 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>triplot</code></td>
<td>Draw a unstructured triangular grid as lines and/or markers.</td>
</tr>
<tr>
<td><code>twinx</code></td>
<td>Make a second axes that shares the x-axis.</td>
</tr>
<tr>
<td><code>twiny</code></td>
<td>Make a second axes that shares the y-axis.</td>
</tr>
<tr>
<td><code>uninstall_repl_displayhook</code></td>
<td>Uninstalls the matplotlib display hook.</td>
</tr>
<tr>
<td><code>violinplot</code></td>
<td>Make a violin plot.</td>
</tr>
<tr>
<td><code>vlines</code></td>
<td>Plot vertical lines.</td>
</tr>
<tr>
<td><code>xcorr</code></td>
<td>Plot the cross correlation between x and y.</td>
</tr>
<tr>
<td><code>xkcd</code></td>
<td>Turns on xkcd sketch-style drawing mode.</td>
</tr>
<tr>
<td><code>xlabel</code></td>
<td>Set the x axis label of the current axis.</td>
</tr>
<tr>
<td><code>xlim</code></td>
<td>Get or set the x limits of the current axes.</td>
</tr>
<tr>
<td><code>xscale</code></td>
<td>Set the scaling of the x-axis.</td>
</tr>
<tr>
<td><code>xticks</code></td>
<td>Get or set the x-limits of the current tick locations and labels.</td>
</tr>
<tr>
<td><code>ylabel</code></td>
<td>Set the y axis label of the current axis.</td>
</tr>
<tr>
<td><code>ylim</code></td>
<td>Get or set the y-limits of the current axes.</td>
</tr>
<tr>
<td><code>yscale</code></td>
<td>Set the scaling of the y-axis.</td>
</tr>
<tr>
<td><code>yticks</code></td>
<td>Get or set the y-limits of the current tick locations and labels.</td>
</tr>
</tbody>
</table>

`matplotlib.pyplot.colormaps()`

Matplotlib provides a number of colormaps, and others can be added using `register_cmap()`. This function documents the built-in colormaps, and will also return a list of all registered colormaps if called.

You can set the colormap for an image, pcolor, scatter, etc, using a keyword argument:

```
imshow(X, cmap=cm.hot)
```

or using the `set_cmap()` function:

```
imshow(X)
pyplot.set_cmap('hot')
pyplot.set_cmap('jet')
```

In interactive mode, `set_cmap()` will update the colormap post-hoc, allowing you to see which one works best for your data.

All built-in colormaps can be reversed by appending _r: For instance, `gray_r` is the reverse of `gray`.

There are several common color schemes used in visualization:

- **Sequential schemes** for unipolar data that progresses from low to high
- **Diverging schemes** for bipolar data that emphasizes positive or negative deviations from a central value
- **Cyclic schemes** meant for plotting values that wrap around at the endpoints, such as phase angle, wind direction, or time of day
- **Qualitative schemes** for nominal data that has no inherent ordering, where color is used only to distinguish categories
The base colormaps are derived from those of the same name provided with Matlab:

<table>
<thead>
<tr>
<th>Colormap</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>autumn</td>
<td>sequential linearly-increasing shades of red-orange-yellow</td>
</tr>
<tr>
<td>bone</td>
<td>sequential increasing black-white color map with a tinge of blue, to emulate X-ray film</td>
</tr>
<tr>
<td>cool</td>
<td>linearly-decreasing shades of cyan-magenta</td>
</tr>
<tr>
<td>copper</td>
<td>sequential increasing shades of black-copper</td>
</tr>
<tr>
<td>flag</td>
<td>repetitive red-white-blue-black pattern (not cyclic at endpoints)</td>
</tr>
<tr>
<td>gray</td>
<td>sequential linearly-increasing black-to-white grayscale</td>
</tr>
<tr>
<td>hot</td>
<td>sequential black-red-yellow-white, to emulate blackbody radiation from an object at increasing temperatures</td>
</tr>
<tr>
<td>hsv</td>
<td>cyclic red-yellow-green-cyan-blue-magenta-red, formed by changing the hue component in the HSV color space</td>
</tr>
<tr>
<td>inferno</td>
<td>perceptually uniform shades of black-red-yellow</td>
</tr>
<tr>
<td>jet</td>
<td>a spectral map with dark endpoints, blue-cyan-yellow-red; based on a fluid-jet simulation by NCSA(^1)</td>
</tr>
<tr>
<td>magma</td>
<td>perceptually uniform shades of black-red-white</td>
</tr>
<tr>
<td>pink</td>
<td>sequential increasing pastel black-pink-white, meant for sepia tone colorization of photographs</td>
</tr>
<tr>
<td>plasma</td>
<td>perceptually uniform shades of blue-red-yellow</td>
</tr>
<tr>
<td>prism</td>
<td>repetitive red-yellow-green-blue-purple-...-green pattern (not cyclic at endpoints)</td>
</tr>
<tr>
<td>spring</td>
<td>linearly-increasing shades of magenta-yellow</td>
</tr>
<tr>
<td>summer</td>
<td>sequential linearly-increasing shades of green-yellow</td>
</tr>
<tr>
<td>viridis</td>
<td>perceptually uniform shades of blue-green-yellow</td>
</tr>
<tr>
<td>winter</td>
<td>linearly-increasing shades of blue-green</td>
</tr>
</tbody>
</table>

For the above list only, you can also set the colormap using the corresponding `pylab` shortcut interface function, similar to Matlab:

```python
imshow(X)
hot()
jet()
```

The next set of palettes are from the Yorick scientific visualisation package, an evolution of the GIST package, both by David H. Munro:

---

\(^1\) Rainbow colormaps, jet in particular, are considered a poor choice for scientific visualization by many researchers: Rainbow Color Map (Still) Considered Harmful
<table>
<thead>
<tr>
<th>Colormap</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gist_earth</td>
<td>mapmaker’s colors from dark blue deep ocean to green lowlands to brown</td>
</tr>
<tr>
<td></td>
<td>highlands to white mountains</td>
</tr>
<tr>
<td>gist_heat</td>
<td>sequential increasing black-red-orange-white, to emulate blackbody</td>
</tr>
<tr>
<td></td>
<td>radiation from an iron bar as it grows hotter</td>
</tr>
<tr>
<td>gist_ncar</td>
<td>pseudo-spectral black-blue-green-yellow-red-purple-white colormap from</td>
</tr>
<tr>
<td></td>
<td>National Center for Atmospheric Research ²</td>
</tr>
<tr>
<td>gist_rainbow</td>
<td>turns through the colors in spectral order from red to violet at full</td>
</tr>
<tr>
<td></td>
<td>saturation (like hsv but not cyclic)</td>
</tr>
<tr>
<td>gist_stern</td>
<td>“Stern special” color table from Interactive Data Language software</td>
</tr>
</tbody>
</table>

The following colormaps are based on the ColorBrewer color specifications and designs developed by Cynthia Brewer:

ColorBrewer Diverging (luminance is highest at the midpoint, and decreases towards differently-colored endpoints):

<table>
<thead>
<tr>
<th>Colormap</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BrBG</td>
<td>brown, white, blue-green</td>
</tr>
<tr>
<td>PiYG</td>
<td>pink, white, yellow-green</td>
</tr>
<tr>
<td>PRGn</td>
<td>purple, white, green</td>
</tr>
<tr>
<td>PuOr</td>
<td>orange, white, purple</td>
</tr>
<tr>
<td>RdBu</td>
<td>red, white, blue</td>
</tr>
<tr>
<td>RdGy</td>
<td>red, white, gray</td>
</tr>
<tr>
<td>RdYlBu</td>
<td>red, yellow, blue</td>
</tr>
<tr>
<td>RdYlGn</td>
<td>red, yellow, green</td>
</tr>
<tr>
<td>Spectral</td>
<td>red, orange, yellow, green,</td>
</tr>
<tr>
<td></td>
<td>blue</td>
</tr>
</tbody>
</table>

ColorBrewer Sequential (luminance decreases monotonically):

² Resembles “BkBlAqGrYeOrReViWh200” from NCAR Command Language. See Color Table Gallery
<table>
<thead>
<tr>
<th>Colormap</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blues</td>
<td>white to dark blue</td>
</tr>
<tr>
<td>BuGn</td>
<td>white, light blue, dark green</td>
</tr>
<tr>
<td>BuPu</td>
<td>white, light blue, dark purple</td>
</tr>
<tr>
<td>GnBu</td>
<td>white, light green, dark blue</td>
</tr>
<tr>
<td>Greens</td>
<td>white to dark green</td>
</tr>
<tr>
<td>Greys</td>
<td>white to black (not linear)</td>
</tr>
<tr>
<td>Oranges</td>
<td>white, orange, dark brown</td>
</tr>
<tr>
<td>OrRd</td>
<td>white, orange, dark red</td>
</tr>
<tr>
<td>PuBu</td>
<td>white, light purple, dark blue</td>
</tr>
<tr>
<td>PuBuGn</td>
<td>white, light purple, dark green</td>
</tr>
<tr>
<td>PuRd</td>
<td>white, light purple, dark red</td>
</tr>
<tr>
<td>Purples</td>
<td>white to dark purple</td>
</tr>
<tr>
<td>RdPu</td>
<td>white, pink, dark purple</td>
</tr>
<tr>
<td>Reds</td>
<td>white to dark red</td>
</tr>
<tr>
<td>YlGn</td>
<td>light yellow, dark green</td>
</tr>
<tr>
<td>YlGnBu</td>
<td>light yellow, light green, dark blue</td>
</tr>
<tr>
<td>YlOrBr</td>
<td>light yellow, orange, dark brown</td>
</tr>
<tr>
<td>YlOrRd</td>
<td>light yellow, orange, dark red</td>
</tr>
</tbody>
</table>

ColorBrewer Qualitative:

(For plotting nominal data, ListedColormap is used, not LinearSegmentedColormap. Different sets of colors are recommended for different numbers of categories.)

- Accent
- Dark2
- Paired
- Pastel1
- Pastel2
- Set1
- Set2
- Set3

Other miscellaneous schemes:
<table>
<thead>
<tr>
<th>Colormap</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>afmhot</td>
<td>sequential black-orange-yellow-white blackbody spectrum, commonly used in 3D atomic force microscopy</td>
</tr>
<tr>
<td>brg</td>
<td>blue-red-green</td>
</tr>
<tr>
<td>bwr</td>
<td>diverging blue-white-red</td>
</tr>
<tr>
<td>cool-warm</td>
<td>diverging blue-gray-red, meant to avoid issues with 3D shading, color blindness, and ordering of colors[^3]</td>
</tr>
<tr>
<td>CM-Rmap</td>
<td>“Default colormaps on color images often reproduce to confusing grayscale images. The proposed colormap maintains an aesthetically pleasing color image that automatically reproduces to a monotonic grayscale with discrete, quantifiable saturation levels.”[^4]</td>
</tr>
<tr>
<td>cube-helix</td>
<td>Unlike most other color schemes cubehelix was designed by D.A. Green to be monotonically increasing in terms of perceived brightness. Also, when printed on a black and white postscript printer, the scheme results in a greyscale with monotonically increasing brightness. This color scheme is named cubehelix because the r,g,b values produced can be visualised as a squashed helix around the diagonal in the r,g,b color cube.</td>
</tr>
<tr>
<td>gnu-plot</td>
<td>gnuplot’s traditional pm3d scheme (black-blue-red-yellow)</td>
</tr>
<tr>
<td>gnu-plot2</td>
<td>sequential color printable as gray (black-blue-violet-yellow-white)</td>
</tr>
<tr>
<td>ocean</td>
<td>green-blue-white</td>
</tr>
<tr>
<td>rainbow</td>
<td>spectral purple-blue-green-yellow-orange-red colormap with diverging luminance</td>
</tr>
<tr>
<td>seismic</td>
<td>diverging blue-white-red</td>
</tr>
<tr>
<td>nipy_spectral</td>
<td>purple-blue-green-yellow-red-white spectrum, originally from the Neuroimaging in Python project</td>
</tr>
<tr>
<td>terrain</td>
<td>mapmaker’s colors, blue-green-yellow-brown-white, originally from IGOR Pro</td>
</tr>
</tbody>
</table>

The following colormaps are redundant and may be removed in future versions. It’s recommended to use the names in the descriptions instead, which produce identical output:

<table>
<thead>
<tr>
<th>Colormap</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gist_gray</td>
<td>identical to gray</td>
</tr>
<tr>
<td>gist_yarg</td>
<td>identical to gray_r</td>
</tr>
<tr>
<td>binary</td>
<td>identical to gray_r</td>
</tr>
<tr>
<td>spectral</td>
<td>identical to nipy_spectral[^5]</td>
</tr>
</tbody>
</table>

[^3]: See Diverging Color Maps for Scientific Visualization by Kenneth Moreland.
[^4]: See A Color Map for Effective Black-and-White Rendering of Color-Scale Images by Carey Rappaport
[^5]: Changed to distinguish from ColorBrewer’s Spectral map. spectral() still works, but set_cmap('nipy_spectral') is recommended for clarity.
API CHANGES

Log of changes to Matplotlib that affect the outward-facing API. If updating Matplotlib breaks your scripts, this list may help you figure out what caused the breakage and how to fix it by updating your code.

For new features that were added to Matplotlib, please see What’s new in matplotlib.

14.1 API Changes in 2.0.1

14.1.1 Extensions to matplotlib.backend_bases.GraphicsContextBase

To better support controlling the color of hatches, the method matplotlib.backend_bases.GraphicsContextBase.set_hatch_color was added to the expected API of GraphicsContext classes. Calls to this method are currently wrapped with a try:...except Attribute: block to preserve back-compatibility with any third-party backends which do not extend GraphicsContextBase.

This value can be accessed in the backends via matplotlib.backend_bases.GraphicsContextBase.get_hatch_color (which was added in 2.0 see Extension to matplotlib.backend_bases.GraphicsContextBase) and should be used to color the hatches.

In the future there may also be hatch_linewidth and hatch_density related methods added. It is encouraged, but not required that third-party backends extend GraphicsContextBase to make adapting to these changes easier.

14.1.2 afm.get_fontconfig_fonts returns a list of paths and does not check for existence

afm.get_fontconfig_fonts used to return a set of paths encoded as a {key: 1, ...} dict, and checked for the existence of the paths. It now returns a list and dropped the existence check, as the same check is performed by the caller (afm.findSystemFonts) as well.

14.1.3 bar now returns rectangles of negative height or width if the corresponding input is negative

plt.bar used to normalize the coordinates of the rectangles that it created, to keep their height and width positives, even if the corresponding input was negative. This normalization has been removed to permit a
simpler computation of the correct sticky_edges to use.

14.1.4 Do not clip line width when scaling dashes

The algorithm to scale dashes was changed to no longer clip the scaling factor: the dash patterns now continue to shrink at thin line widths. If the line width is smaller than the effective pixel size, this may result in dashed lines turning into solid gray-ish lines. This also required slightly tweaking the default patterns for ‘–’, ‘:’, and ‘.-’ so that with the default line width the final patterns would not change.

There is no way to restore the old behavior.

14.1.5 Deprecate ‘Vega’ color maps

The “Vega” colormaps are deprecated in Matplotlib 2.0.1 and will be removed in Matplotlib 2.2. Use the “tab” colormaps instead: “tab10”, “tab20”, “tab20b”, “tab20c”.

14.2 API Changes in 2.0.0

14.2.1 Deprecation and removal

Color of Axes

The axisbg and axisbgcolor properties on Axes have been deprecated in favor of facecolor.

GTK and GDK backends deprecated

The GDK and GTK backends have been deprecated. These obsolete backends allow figures to be rendered via the GDK API to files and GTK2 figures. They are untested and known to be broken, and their use has been discouraged for some time. Instead, use the GTKAgg and GTKCairo backends for rendering to GTK2 windows.

WX backend deprecated

The WX backend has been deprecated. It is untested, and its use has been discouraged for some time. Instead, use the WXAgg backend for rendering figures to WX windows.

CocoaAgg backend removed

The deprecated and not fully functional CocoaAgg backend has been removed.
round removed from TkAgg Backend

The TkAgg backend had its own implementation of the `round` function. This was unused internally and has been removed. Instead, use either the `round` builtin function or `numpy.round`.

‘hold’ functionality deprecated

The ‘hold’ keyword argument and all functions and methods related to it are deprecated, along with the ‘axes.hold’ rcParams entry. The behavior will remain consistent with the default `hold=True` state that has long been in place. Instead of using a function or keyword argument (`hold=False`) to change that behavior, explicitly clear the axes or figure as needed prior to subsequent plotting commands.

14.2.2 Artist.update has return value

The methods `matplotlib.artist.Artist.set`, `matplotlib.Artist.update`, and the function `matplotlib.artist.Artist.setp` now use a common codepath to look up how to update the given artist properties (either using the setter methods or an attribute/property).

The behavior of `matplotlib.Artist.update` is slightly changed to return a list of the values returned from the setter methods to avoid changing the API of `matplotlib.Artist.set` and `matplotlib.artist.setp`.

The keys passed into `matplotlib.Artist.update` are now converted to lower case before being processed, to match the behavior of `matplotlib.Artist.set` and `matplotlib.artist.setp`. This should not break any user code because there are no set methods with capitals in their names, but this puts a constraint on naming properties in the future.

14.2.3 Legend initializers gain edgecolor and facecolor kwargs

The `Legend` background patch (or ‘frame’) can have its `edgecolor` and `facecolor` determined by the corresponding keyword arguments to the `matplotlib.legend.Legend` initializer, or to any of the methods or functions that call that initializer. If left to their default values of `None`, their values will be taken from `matplotlib.rcParams`. The previously-existing `framealpha` kwarg still controls the alpha transparency of the patch.

14.2.4 Qualitative colormaps

Colorbrewer’s qualitative/discrete colormaps (“Accent”, “Dark2”, “Paired”, “Pastel1”, “Pastel2”, “Set1”, “Set2”, “Set3”) are now implemented as `ListedColormap` instead of `LinearSegmentedColormap`.

To use these for images where categories are specified as integers, for instance, use:

```python
plt.imshow(x, cmap='Dark2', norm=colors.NoNorm())
```
14.2.5 Change in the draw_image backend API

The `draw_image` method implemented by backends has changed its interface.

This change is only relevant if the backend declares that it is able to transform images by returning `True` from `option_scale_image`. See the `draw_image` docstring for more information.

14.2.6 `matplotlib.ticker.LinearLocator` algorithm update

The `matplotlib.ticker.LinearLocator` is used to define the range and location of axis ticks when the user wants an exact number of ticks. `LinearLocator` thus differs from the default locator `MaxNLocator`, for which the user specifies a maximum number of intervals rather than a precise number of ticks.

The view range algorithm in `matplotlib.ticker.LinearLocator` has been changed so that more convenient tick locations are chosen. The new algorithm returns a plot view range that is a multiple of the user-requested number of ticks. This ensures tick marks will be located at whole integers more consistently. For example, when both y-axes of a “twinx” plot use `matplotlib.ticker.LinearLocator` with the same number of ticks, their y-tick locations and grid lines will coincide.

14.2.7 `matplotlib.ticker.LogLocator` gains numticks kwarg

The maximum number of ticks generated by the `LogLocator` can now be controlled explicitly via setting the new ‘numticks’ kwarg to an integer. By default the kwarg is `None` which internally sets it to the ‘auto’ string, triggering a new algorithm for adjusting the maximum according to the axis length relative to the ticklabel font size.

14.2.8 `matplotlib.ticker.LogFormatter`: two new kwargs

Previously, minor ticks on log-scaled axes were not labeled by default. An algorithm has been added to the `LogFormatter` to control the labeling of ticks between integer powers of the base. The algorithm uses two parameters supplied in a kwarg tuple named ‘minor_thresholds’. See the docstring for further explanation.

To improve support for axes using `SymmetricLogLocator`, a ‘linthresh’ kwarg was added.

14.2.9 New defaults for 3D quiver function in `mpl_toolkits.mplot3d.axes3d.py`

Matplotlib has both a 2D and a 3D `quiver` function. These changes affect only the 3D function and make the default behavior of the 3D function match the 2D version. There are two changes:

1. The 3D quiver function previously normalized the arrows to be the same length, which makes it unusable for situations where the arrows should be different lengths and does not match the behavior of the 2D function. This normalization behavior is now controlled with the `normalize` keyword, which defaults to `False`.

2. The `pivot` keyword now defaults to `tail` instead of `tip`. This was done in order to match the default behavior of the 2D quiver function.

To obtain the previous behavior with the 3D quiver function, one can call the function with
ax.quiver(x, y, z, u, v, w, normalize=True, pivot='tip')

where “ax” is an Axes3d object created with something like

```python
import mpl_toolkits.mplot3d.axes3d
ax = plt.subplot(111, projection='3d')
```

### 14.2.10 Stale figure behavior

Attempting to draw the figure will now mark it as not stale (independent if the draw succeeds). This change is to prevent repeatedly trying to re-draw a figure which is raising an error on draw. The previous behavior would only mark a figure as not stale after a full re-draw succeeded.

### 14.2.11 The spectral colormap is now nipy_spectral

The colormaps formerly known as spectral and spectral_r have been replaced by nipy_spectral and nipy_spectral_r since Matplotlib 1.3.0. Even though the colormap was deprecated in Matplotlib 1.3.0, it never raised a warning. As of Matplotlib 2.0.0, using the old names raises a deprecation warning. In the future, using the old names will raise an error.

### 14.2.12 Default install no longer includes test images

To reduce the size of wheels and source installs, the tests and baseline images are no longer included by default.

To restore installing the tests and images, use a `setup.cfg` with

```ini
[packages]
tests = True
toolkits_tests = True
```

in the source directory at build/install time.

### 14.3 Changes in 1.5.3

#### 14.3.1 ax.plot(..., marker=None) gives default marker

Prior to 1.5.3 kwargs passed to `plot` were handled in two parts – default kwargs generated internal to `plot` (such as the cycled styles) and user supplied kwargs. The internally generated kwargs were passed to the `matplotlib.lines.Line2D.__init__` and the user kwargs were passed to `ln.set(**kwargs)` to update the artist after it was created. Now both sets of kwargs are merged and passed to `__init__`. This change was made to allow `None` to be passed in via the user kwargs to mean ‘do the default thing’ as is the convention through out mpl rather than raising an exception.
Unlike most Line2D setter methods set_marker did accept None as a valid input which was mapped to ‘no marker’. Thus, by routing this marker=None through __init__ rather than set(...) the meaning of ax.plot(..., marker=None) changed from ‘no markers’ to ‘default markers from rcparams’.

This is change is only evident if mpl.rcParams['lines.marker'] has a value other than 'None' (which is string 'None' which means ‘no marker’).

14.4 Changes in 1.5.2

14.4.1 Default Behavior Changes

Changed default autorange behavior in boxplots

Prior to v1.5.2, the whiskers of boxplots would extend to the minimum and maximum values if the quartiles were all equal (i.e., Q1 = median = Q3). This behavior has been disabled by default to restore consistency with other plotting packages.

To restore the old behavior, simply set autorange=True when calling plt.boxplot.

14.5 Changes in 1.5.0

14.5.1 Code Changes

Reversed matplotlib.cbook.ls_mapper, added ls_mapper_r

Formerly, matplotlib.cbook.ls_mapper was a dictionary with the long-form line-style names ("solid") as keys and the short forms ("-") as values. This long-to-short mapping is now done by ls_mapper_r, and the short-to-long mapping is done by the ls_mapper.

Prevent moving artists between Axes, Property-ify Artist.axes, deprecate Artist.{get,set}_axes

This was done to prevent an Artist that is already associated with an Axes from being moved/addded to a different Axes. This was never supported as it causes havoc with the transform stack. The apparent support for this (as it did not raise an exception) was the source of multiple bug reports and questions on SO.

For almost all use-cases, the assignment of the axes to an artist should be taken care of by the axes as part of the Axes.add_* method, hence the deprecation of {get,set}_axes.

Removing the set_axes method will also remove the ‘axes’ line from the ACCEPTS kwarg tables (assuming that the removal date gets here before that gets overhauled).

Tightened input validation on ‘pivot’ kwarg to quiver

Tightened validation so that only {‘tip’, ‘tail’, ‘mid’, and ‘middle’} (but any capitalization) are valid values for the ‘pivot’ kwarg in the Quiver.__init__ (and hence Axes.quiver and plt.quiver which both
fully delegate to Quiver). Previously any input matching ‘mid.*’ would be interpreted as ‘middle’, ‘tip.*’ as ‘tip’ and any string not matching one of those patterns as ‘tail’.

The value of Quiver.pivot is normalized to be in the set {‘tip’, ‘tail’, ‘middle’} in Quiver.__init__.

Reordered Axes.get_children

The artist order returned by Axes.get_children did not match the one used by Axes.draw. They now use the same order, as Axes.draw now calls Axes.get_children.

Changed behaviour of contour plots

The default behaviour of contour() and contourf() when using a masked array is now determined by the new keyword argument corner_mask, or if this is not specified then the new rcParam contour.corner_mask instead. The new default behaviour is equivalent to using corner_mask=True; the previous behaviour can be obtained using corner_mask=False or by changing the rcParam. The example http://matplotlib.org/examples/pylab_examples/contour_corner_mask.html demonstrates the difference.

Contour labels may now appear in different places than in earlier versions of Matplotlib.

In addition, the keyword argument nchunk now applies to contour() as well as contourf(), and it subdivides the domain into subdomains of exactly nchunk by nchunk quads, whereas previously it was only roughly nchunk by nchunk quads.

The C/C++ object that performs contour calculations used to be stored in the public attribute QuadContourSet.Cntr, but is now stored in a private attribute and should not be accessed by end users.

Added set_params function to all Locator types

This was a bug fix targeted at making the api for Locators more consistent.

In the old behavior, only locators of type MaxNLocator have set_params() defined, causing its use on any other Locator to raise an AttributeError ( aside: set_params(args) is a function that sets the parameters of a Locator instance to be as specified within args). The fix involves moving set_params() to the Locator class such that all subtypes will have this function defined.

Since each of the Locator subtypes have their own modifiable parameters, a universal set_params() in Locator isn’t ideal. Instead, a default no-operation function that raises a warning is implemented in Locator. Subtypes extending Locator will then override with their own implementations. Subtypes that do not have a need for set_params() will fall back onto their parent’s implementation, which raises a warning as intended.

In the new behavior, Locator instances will not raise an AttributeError when set_params() is called. For Locators that do not implement set_params(), the default implementation in Locator is used.

Disallow None as x or y value in ax.plot

Do not allow None as a valid input for the x or y args in ax.plot. This may break some user code, but this was never officially supported (ex documented) and allowing None objects through can lead to confusing
exceptions downstream.

To create an empty line use

```python
ln1, = ax.plot([], [], ...)
ln2, = ax.plot([], ...)
```

In either case to update the data in the Line2D object you must update both the x and y data.

**Removed args and kwargs from MicrosecondLocator.__call__**

The call signature of `__call__()` has changed from `__call__(self, *args, **kwargs)` to `__call__(self)`. This is consistent with the superclass `Locator` and also all the other Locators derived from this superclass.

**No ValueError for the MicrosecondLocator and YearLocator**

The `MicrosecondLocator` and `YearLocator` objects when called will return an empty list if the axes have no data or the view has no interval. Previously, they raised a `ValueError`. This is consistent with all the Date Locators.

**‘OffsetBox.DrawingArea’ respects the ‘clip’ keyword argument**

The call signature was `OffsetBox.DrawingArea(..., clip=True)` but nothing was done with the clip argument. The object did not do any clipping regardless of that parameter. Now the object can and does clip the child Artists if they are set to be clipped.

You can turn off the clipping on a per-child basis using `child.set_clip_on(False)`.

**Add salt to clipPath id**

Add salt to the hash used to determine the id of the clipPath nodes. This is to avoid conflicts when two svg documents with the same clip path are included in the same document (see https://github.com/ipython/ipython/issues/8133 and https://github.com/matplotlib/matplotlib/issues/4349), however this means that the svg output is no longer deterministic if the same figure is saved twice. It is not expected that this will affect any users as the current ids are generated from an md5 hash of properties of the clip path and any user would have a very difficult time anticipating the value of the id.

**Changed snap threshold for circle markers to inf**

When drawing circle markers above some marker size (previously 6.0) the path used to generate the marker was snapped to pixel centers. However, this ends up distorting the marker away from a circle. By setting the snap threshold to inf snapping is never done on circles.

This change broke several tests, but is an improvement.
Preserve units with Text position

Previously the ‘get_position’ method on Text would strip away unit information even though the units were still present. There was no inherent need to do this, so it has been changed so that unit data (if present) will be preserved. Essentially a call to ‘get_position’ will return the exact value from a call to ‘set_position’.

If you wish to get the old behaviour, then you can use the new method called ‘get_unitless_position’.

New API for custom Axes view changes

Interactive pan and zoom were previously implemented using a Cartesian-specific algorithm that was not necessarily applicable to custom Axes. Three new private methods, _get_view(), _set_view(), and _set_view_from_bbox(), allow for custom Axes classes to override the pan and zoom algorithms. Implementors of custom Axes who override these methods may provide suitable behaviour for both pan and zoom as well as the view navigation buttons on the interactive toolbars.

14.5.2 MathTex visual changes

The spacing commands in mathtext have been changed to more closely match vanilla TeX.

Improved spacing in mathtext

The extra space that appeared after subscripts and superscripts has been removed.

No annotation coordinates wrap

In #2351 for 1.4.0 the behavior of ['axes points', 'axes pixel', 'figure points', 'figure pixel'] as coordinates was change to no longer wrap for negative values. In 1.4.3 this change was reverted for 'axes points' and 'axes pixel' and in addition caused 'axes fraction' to wrap. For 1.5 the behavior has been reverted to as it was in 1.4.0-1.4.2, no wrapping for any type of coordinate.

14.5.3 Deprecation

Deprecated GraphicsContextBase.set_graylevel

The GraphicsContextBase.set_graylevel function has been deprecated in 1.5 and will be removed in 1.6. It has been unused. The GraphicsContextBase.set_foreground could be used instead.

deprecated idle_event

The idle_event was broken or missing in most backends and causes spurious warnings in some cases, and its use in creating animations is now obsolete due to the animations module. Therefore code involving it has been removed from all but the wx backend (where it partially works), and its use is deprecated. The animations module may be used instead to create animations.
color_cycle deprecated

In light of the new property cycling feature, the Axes method `set_color_cycle` is now deprecated. Calling this method will replace the current property cycle with one that cycles just the given colors.

Similarly, the rc parameter `axes.color_cycle` is also deprecated in lieu of the new `axes.prop_cycle` parameter. Having both parameters in the same rc file is not recommended as the result cannot be predicted. For compatibility, setting `axes.color_cycle` will replace the cycler in `axes.prop_cycle` with a color cycle. Accessing `axes.color_cycle` will return just the color portion of the property cycle, if it exists.

Timeline for removal has not been set.

14.5.4 Bundled jquery

The version of jquery bundled with the webagg backend has been upgraded from 1.7.1 to 1.11.3. If you are using the version of jquery bundled with webagg you will need to update your html files as such

```
- <script src="_static/jquery/js/jquery-1.7.1.min.js"></script>
+ <script src="_static/jquery/js/jquery-1.11.3.min.js"></script>
```

14.5.5 Code Removed

Removed Image from main namespace

Image was imported from PIL/pillow to test if PIL is available, but there is no reason to keep `Image` in the namespace once the availability has been determined.

Removed lod from Artist

Removed the method `set_lod` and all references to the attribute `_lod` as the are not used anywhere else in the code base. It appears to be a feature stub that was never built out.

Removed threading related classes from cbook

The classes Scheduler, Timeout, and Idle were in cbook, but are not used internally. They appear to be a prototype for the idle event system which was not working and has recently been pulled out.

Removed Lena images from sample_data

The `lena.png` and `lena.jpg` images have been removed from Matplotlib’s sample_data directory. The images are also no longer available from `matplotlib.cbook.get_sample_data`. We suggest using `matplotlib.cbook.get_sample_data('grace_hopper.png')` or `matplotlib.cbook.get_sample_data('grace_hopper.jpg')` instead.
Legend

Removed handling of `loc` as a positional argument to `Legend`.

Legend handlers

Remove code to allow legend handlers to be callable. They must now implement a method `legend_artist`.

Axis

Removed method `set_scale`. This is now handled via a private method which should not be used directly by users. It is called via `Axes.set_{x,y}scale` which takes care of ensuring the related changes are also made to the `Axes` object.

finance.py

Removed functions with ambiguous argument order from `finance.py`.

Annotation

Removed `textcoords` and `xytext` proprieties from Annotation objects.

spinxext.ipython_* .py

Both `ipython_console_highlighting` and `ipython_directive` have been moved to IPython.
Change your import from `matplotlib.sphinxext.ipython_directive` to `IPython.sphinxext.ipython_directive` and from `matplotlib.sphinxext.ipython_directive` to `IPython.sphinxext.ipython_directive`.

LineCollection.color

Deprecated in 2005, use `set_color`.

`remove 'faceted' as a valid value for shading in tri.tripcolor`

Use `edgecolor` instead. Added validation on `shading` to only be valid values.

Remove `faceted` kwarg from scatter

Remove support for the `faceted` kwarg. This was deprecated in d48b34288e9651ff95c3b8a071ef5ac5cf50bae7 (2008-04-18!) and replaced by `edgecolor`. 
Remove set_colorbar method from ScalarMappable

Remove set_colorbar method, use colorbar attribute directly.

patheffects.svg

- remove get_proxy_renderer method from AbstractPathEffect class
- remove patch_alpha and offset_xy from SimplePatchShadow

Remove testing.image_util.py

Contained only a no-longer used port of functionality from PIL

Remove mlab.FIFOBuffer

Not used internally and not part of core mission of mpl.

Remove mlab.prepca

Deprecated in 2009.

Remove NavigationToolbar2QTAgg

Added no functionality over the base NavigationToolbar2Qt

mpl.py

Remove the module matplotlib.mpl. Deprecated in 1.3 by PR #1670 and commit 78ce67d161625833cacff23cfe5d74920248c5b2

14.6 Changes in 1.4.x

14.6.1 Code changes

- A major refactoring of the axes module was made. The axes module has been split into smaller modules:
  - the _base module, which contains a new private _AxesBase class. This class contains all methods except plotting and labelling methods.
  - the axes module, which contains the Axes class. This class inherits from _AxesBase, and contains all plotting and labelling methods.
the `subplot` module, with all the classes concerning subplotting.

There are a couple of things that do not exist in the `axes` module’s namespace anymore. If you use them, you need to import them from their original location:

- `math` - `import math`
- `ma` - `from numpy import ma`
- `cbook` - `from matplotlib import cbook`
- `docstring` - `from matplotlib import docstring`
- `is_sequence_of_strings` - `from matplotlib.cbook import is_sequence_of_strings`
- `is_string_like` - `from matplotlib.cbook import is_string_like`
- `iterable` - `from matplotlib.cbook import iterable`
- `itertools` - `import itertools`
- `artist` - `from matplotlib import artist as artist`
- `matplotlib` - `import matplotlib`
- `collections` - `from matplotlib import collections as collections`
- `colors` - `from matplotlib import colors as colors`
- `contour` - `from matplotlib import contour as contour`
- `patches` - `from matplotlib import patches as patches`
- `path` - `from matplotlib import path as path`
- `quiver` - `from matplotlib import quiver as quiver`
- `stack` - `from matplotlib import stack as stack`
- `stream` - `from matplotlib import stream as stream`
- `table` - `from matplotlib import table as table`

As part of the refactoring to enable Qt5 support, the module `matplotlib.backends.qt4_compat` was renamed to `matplotlib.qt_compat`. `qt4_compat` is deprecated in 1.4 and will be removed in 1.5.

- The `errorbar()` method has been changed such that the upper and lower limits (`lolims, uplims, xlolims, xuplims`) now point in the correct direction.
- The `fmt` kwarg for `plot()` defaults.
- A bug has been fixed in the path effects rendering of fonts, which now means that the font size is consistent with non-path effect fonts. See https://github.com/matplotlib/matplotlib/issues/2889 for more detail.
- The Sphinx extensions `ipython_directive` and `ipython_console_highlighting` have been moved to the IPython project itself. While they remain in Matplotlib for this release, they have been deprecated. Update your extensions in `conf.py` to point to `IPython.sphinxext.ipython_directive` instead of `matplotlib.sphinxext.ipython_directive`.
In finance, almost all functions have been deprecated and replaced with a pair of functions name *_ochl and *_ohlc. The former is the ‘open-close-high-low’ order of quotes used previously in this module, and the latter is the ‘open-high-low-close’ order that is standard in finance.

For consistency the face_alpha keyword to matplotlib.path_effects.SimplePatchShadow has been deprecated in favour of the alpha keyword. Similarly, the keyword offset_xy is now named offset across all _Base has been renamed to matplotlib.path_effects.AbstractPathEffect. matplotlib.path_effects.ProxyRenderer has been renamed to matplotlib.path_effects.PathEffectRenderer and is now a full RendererBase subclass.

The artist used to draw the outline of a colorbar has been changed from a matplotlib.path.Line2D to matplotlib.path.Polygon, thus colorbar.ColorbarBase.outline is now a matplotlib.path.Polygon object.

The legend handler interface has changed from a callable, to any object which implements the legend_artists method (a deprecation phase will see this interface be maintained for v1.4). See Legend guide for further details. Further legend changes include:

- matplotlib.axes.Axes._get_legend_handles() now returns a generator of handles, rather than a list.
- The legend() function’s “loc” positional argument has been deprecated. Use the “loc” keyword instead.

The rcParams savefig.transparent has been added to control default transparency when saving figures.

Slightly refactored the Annotation family. The text location in Annotation is now handled entirely handled by the underlying Text object so set_position works as expected. The attributes xytext and textcoords have been deprecated in favor of xyann and anncoords so that Annotation and AnnotationBbox can share a common sensibly named api for getting/setting the location of the text or box.

- xyann -> set the location of the annotation
- xy -> set where the arrow points to
- anncoords -> set the units of the annotation location
- xycoords -> set the units of the point location
- set_position() -> Annotation only set location of annotation

matplotlib.mlab.specgram, matplotlib.mlab.psd, matplotlib.mlab.csd, matplotlib.mlab.cohere, matplotlib.mlab.cohere_pairs, matplotlib.pyplot.specgram, matplotlib.pyplot.psd, matplotlib.pyplot.csd, and matplotlib.pyplot.cohere now raise ValueError where they previously raised AssertionError.

For matplotlib.mlab.psd, matplotlib.mlab.csd, matplotlib.mlab.cohere, matplotlib.mlab.cohere_pairs, matplotlib.pyplot.specgram, matplotlib.pyplot.psd, matplotlib.pyplot.csd, and matplotlib.pyplot.cohere, in cases where a shape (n, 1) array is returned, this is now converted to a (n, ) array. Previously, (n, m) arrays were averaged to an (n, ) array, but (n, 1) arrays were return unchanged. This change makes the dimensions consistent in both cases.
- Added the rcParam `axes.formatter.useoffset` to control the default value of `useOffset` in `ticker.ScalarFormatter`.
- Added `Formatter` sub-class `StrMethodFormatter` which does the exact same thing as `FormatStrFormatter`, but for new-style formatting strings.
- Deprecated `matplotlib.testing.image_util` and the only function within, `matplotlib.testing.image_util.autocontrast`. These will be removed completely in v1.5.0.
- The `fmt` argument of `plot_date()` has been changed from `bo` to just `o`, so color cycling can happen by default.
- Removed the class `FigureManagerQTAgg` and deprecated `NavigationToolbar2QTAgg` which will be removed in 1.5.
- Removed formerly public (non-prefixed) attributes `rect` and `drawRect` from `FigureCanvasQTAgg`; they were always an implementation detail of the (preserved) `drawRectangle()` function.
- The function signatures of `tight_bbox.adjust_bbox` and `tight_bbox.process_figure_for_rasterizing` have been changed. A new `fixed_dpi` parameter allows for overriding the `figure.dpi` setting instead of trying to deduce the intended behaviour from the file format.
- Added support for horizontal/vertical axes padding to `mpl_toolkits.axes_grid1.ImageGrid` — argument `axes_pad` can now be tuple-like if separate axis padding is required. The original behavior is preserved.
- Added support for skewed transforms to `matplotlib.transforms.Affine2D`, which can be created using the `skew` and `skew_deg` methods.
- Added clockwise parameter to control sectors direction in `axes.pie`.
- In `matplotlib.lines.Line2D` the `markevery` functionality has been extended. Previously an integer start-index and stride-length could be specified using either a two-element-list or a two-element-tuple. Now this can only be done using a two-element-tuple. If a two-element-list is used then it will be treated as numpy fancy indexing and only the two markers corresponding to the given indexes will be shown.
- removed prop kwarg from `mpl_toolkits.axes_grid1.anchored_artists.AnchoredSizeBar` call. It was passed through to the base-class `__init__` and is only used for setting padding. Now `fontproperties` (which is what is really used to set the font properties of `AnchoredSizeBar`) is passed through in place of `prop`. If `fontproperties` is not passed in, but `prop` is, then `prop` is used inplace of `fontproperties`. If both are passed in, `prop` is silently ignored.
- The use of the index 0 in `plt.subplot` and related commands is deprecated. Due to a lack of validation calling `plt.subplots(2, 2, 0)` does not raise an exception, but puts an axes in the `_last_` position. This is due to the indexing in subplot being 1-based (to mirror MATLAB) so before indexing into the `GridSpec` object used to determine where the axes should go, 1 is subtracted off. Passing in 0 results in passing -1 to `GridSpec` which results in getting the last position back. Even though this behavior is clearly wrong and not intended, we are going through a deprecation cycle in an abundance of caution that any users are exploiting this ‘feature’. The use of 0 as an index will raise a warning in 1.4 and an exception in 1.5.
- Clipping is now off by default on offset boxes.
• Matplotlib now uses a less-aggressive call to gc.collect(1) when closing figures to avoid major delays with large numbers of user objects in memory.
• The default clip value of all pie artists now defaults to False.

14.6.2 Code removal

• Removed mlab.levypdf. The code raised a numpy error (and has for a long time) and was not the standard form of the Levy distribution. scipy.stats.levy should be used instead

14.7 Changes in 1.3.x

14.7.1 Changes in 1.3.1

It is rare that we make an API change in a bugfix release, however, for 1.3.1 since 1.3.0 the following change was made:

• text.Text.cached (used to cache font objects) has been made into a private variable. Among the obvious encapsulation benefit, this removes this confusing-looking member from the documentation.
• The method hist() now always returns bin occupancies as an array of type float. Previously, it was sometimes an array of type int, depending on the call.

14.7.2 Code removal

• The following items that were deprecated in version 1.2 or earlier have now been removed completely.
  – The Qt 3.x backends (qt and qtagg) have been removed in favor of the Qt 4.x backends (qt4 and qt4agg).
  – The FltkAgg and Emf backends have been removed.
  – The matplotlib.nxutils module has been removed. Use the functionality on matplotlib.path.Path.contains_point and friends instead.
  – The following kwargs to the legend function have been renamed:
    * pad -> borderpad
    * labelsep -> labelspaceing
    * handlelen -> handlelength
    * handletextsep -> handletextpad
    * axespad -> borderaxespad
  Related to this, the following rcParams have been removed:
• *legend.pad, legend.labelsep, legend.handlelen, legend.handletextsep and legend.axespad*

  – For the hist function, instead of width, use rwidth (relative width).

  – On patches.Circle, the resolution kwarg has been removed. For a circle made up of line segments, use patches.CirclePolygon.

  – The printing functions in the Wx backend have been removed due to the burden of keeping them up-to-date.

  – mlab.liaupunov has been removed.

  – mlab.save, mlab.load, pylab.save and pylab.load have been removed. We recommend using numpy.savetxt and numpy.loadtxt instead.

  – widgets.HorizontalSpanSelector has been removed. Use widgets.SpanSelector instead.

14.7.3 Code deprecation

• The CocoaAgg backend has been deprecated, with the possibility for deletion or resurrection in a future release.

• The top-level functions in matplotlib.path that are implemented in C++ were never meant to be public. Instead, users should use the Pythonic wrappers for them in the path.Path and collections.Collection classes. Use the following mapping to update your code:

  – point_in_path -> path.Path.contains_point

  – get_path_extents -> path.Path.get_extents

  – point_in_path_collection -> collection.Collection.contains

  – path_in_path -> path.Path.contains_path

  – path_intersects_path -> path.Path.intersects_path

  – convert_path_to_polygons -> path.Path.to_polygons

  – cleanup_path -> path.Path.cleaned

  – points_in_path -> path.Path.contains_points

  – clip_path_to_rect -> path.Path.clip_to_bbox

• matplotlib.colors.normalize and matplotlib.colors.no_norm have been deprecated in favour of matplotlib.colors.Normalize and matplotlib.colors.NoNorm respectively.

• The ScalarMappable class’ set_colorbar is now deprecated. Instead, the matplotlib.cm.ScalarMappable.colorbar attribute should be used. In previous Matplotlib versions this attribute was an undocumented tuple of (colorbar_instance, colorbar_axes) but is now just colorbar_instance. To get the colorbar axes it is possible to just use the ax attribute on a colorbar instance.
- The `mpl` module is now deprecated. Those who relied on this module should transition to simply using `import matplotlib as mpl`.

### 14.7.4 Code changes

- **Patch** now fully supports using RGBA values for its `facecolor` and `edgecolor` attributes, which enables faces and edges to have different alpha values. If the `Patch` object’s alpha attribute is set to anything other than `None`, that value will override any alpha-channel value in both the face and edge colors. Previously, if `Patch` had `alpha=None`, the alpha component of `edgecolor` would be applied to both the edge and face.

- The optional `isRGB` argument to `set_foreground()` (and the other GraphicsContext classes that descend from it) has been renamed to `isRGBA`, and should now only be set to `True` if the `fg` color argument is known to be an RGBA tuple.

- For `Patch`, the `capstyle` used is now `butt`, to be consistent with the default for most other objects, and to avoid problems with non-solid `linestyle` appearing solid when using a large `linewidth`. Previously, `Patch` used `capstyle='projecting'`.

- Path objects can now be marked as `readonly` by passing `readonly=True` to its constructor. The built-in path singletons, obtained through `Path.unit*` class methods return readonly paths. If you have code that modified these, you will need to make a deepcopy first, using either:

```python
import copy
path = copy.deepcopy(Path.unit_circle())
# or
path = Path.unit_circle().deepcopy()
```

Deep copying a Path always creates an editable (i.e. non-readonly) Path.

- The list at `Path.NUM_VERTICES` was replaced by a dictionary mapping Path codes to the number of expected vertices at `NUM_VERTICES_FOR_CODE`.

- To support XKCD style plots, the `matplotlib.path.cleanup_path()` method’s signature was updated to require a sketch argument. Users of `matplotlib.path.cleanup_path()` are encouraged to use the new `cleaned()` Path method.

- Data limits on a plot now start from a state of having “null” limits, rather than limits in the range (0, 1). This has an effect on artists that only control limits in one direction, such as `axvline` and `axhline`, since their limits will not longer also include the range (0, 1). This fixes some problems where the computed limits would be dependent on the order in which artists were added to the axes.

- Fixed a bug in setting the position for the right/top spine with data position type. Previously, it would draw the right or top spine at +1 data offset.

- In `FancyArrow`, the default arrow head width, `head_width`, has been made larger to produce a visible arrow head. The new value of this kwarg is `head_width = 20 * width`.

- It is now possible to provide `number of levels + 1` colors in the case of `extend='both'` for `contourf` (or just `number of levels` colors for an extend value `min` or `max`) such that the resulting
colormap's `set_under` and `set_over` are defined appropriately. Any other number of colors will continue to behave as before (if more colors are provided than levels, the colors will be unused). A similar change has been applied to contour, where `extend='both'` would expect `number of levels + 2` colors.

- A new keyword `extendrect` in `colorbar()` and `ColorbarBase` allows one to control the shape of colorbar extensions.
- The extension of `MultiCursor` to both vertical (default) and/or horizontal cursor implied that `self.line` is replaced by `self.vline` for vertical cursors lines and `self.hline` is added for the horizontal cursors lines.
- On POSIX platforms, the `report_memory()` function raises `NotImplementedError` instead of `OSError` if the `ps` command cannot be run.
- The `matplotlib.cbook.check_output()` function has been moved to `matplotlib.compat.subprocess()`.

### 14.7.5 Configuration and rcParams

- On Linux, the user-specific `matplotlibrc` configuration file is now located in `config/matplotlibrc` to conform to the XDG Base Directory Specification.
- The `font.*` rcParams now affect only text objects created after the rcParam has been set, and will not retroactively affect already existing text objects. This brings their behavior in line with most other rcParams.
- Removed call of `grid()` in `plotfile()`. To draw the axes grid, set the `axes.grid` rcParam to `True`, or explicitly call `grid()`.

### 14.8 Changes in 1.2.x

- The classic option of the rc parameter `toolbar` is deprecated and will be removed in the next release.
- The `isvector()` method has been removed since it is no longer functional.
- The `rasterization_zorder` property on `Axes` a zorder below which artists are rasterized. This has defaulted to `-30000.0`, but it now defaults to `None`, meaning no artists will be rasterized. In order to rasterize artists below a given zorder value, `set_rasterization_zorder` must be explicitly called.
- In `scatter()`, and `scatter`, when specifying a marker using a tuple, the angle is now specified in degrees, not radians.
- Using `twinx()` or `twiny()` no longer overrides the current locaters and formatters on the axes.
- In `contourf()`, the handling of the `extend` kwarg has changed. Formerly, the extended ranges were mapped after to 0, 1 after being normed, so that they always corresponded to the extreme values of the colormap. Now they are mapped outside this range so that they correspond to the special colormap values determined by the `set_under()` and `set_over()` methods, which default to the colormap end points.
• The new rc parameter `savefig.format` replaces `cairo.format` and `savefig.extension`, and sets the default file format used by `matplotlib.figure.Figure.savefig()`.

• In `pie()` and `pie()`, one can now set the radius of the pie; setting the `radius` to ‘None’ (the default value), will result in a pie with a radius of 1 as before.

• Use of `projection_factory()` is now deprecated in favour of axes class identification using `process_projection_requirements()` followed by direct axes class invocation (at the time of writing, functions which do this are: `add_axes()`, `add_subplot()` and `gca()`). Therefore:

```python
class MyTransform(mtrans.Transform):
    def transform_non_affine(self, xy):
        ...
        transform_non_affine = transform
```

This change means that third party objects can expose themselves as Matplotlib axes by providing a `_as_mpl_axes` method. See *Developer’s guide for creating scales and transformations* for more detail.

• A new keyword `extendfrac` in `colorbar()` and `ColorbarBase` allows one to control the size of the triangular minimum and maximum extensions on colorbars.

• A new keyword `capthick` in `errorbar()` has been added as an intuitive alias to the `markeredgewidth` and `mew` keyword arguments, which indirectly controlled the thickness of the caps on the errorbars. For backwards compatibility, specifying either of the original keyword arguments will override any value provided by `capthick`.

• Transform subclassing behaviour is now subtly changed. If your transform implements a non-affine transformation, then it should override the `transform_non_affine` method, rather than the generic `transform` method. Previously transforms would define `transform` and then copy the method into `transform_non_affine`:

```python
class MyTransform(mtrans.Transform):
    def transform(self, xy):
        ...
```
• Artists no longer have `x_isdata` or `y_isdata` attributes; instead any artist’s transform can be interrogated with `artist_instance.get_transform().contains_branch(ax.transData)`.

• Lines added to an axes now take into account their transform when updating the data and view limits. This means transforms can now be used as a pre-transform. For instance:

```python
>>> import matplotlib.pyplot as plt
>>> import matplotlib.transforms as mtrans

>>> ax = plt.axes()
>>> ax.plot(range(10), transform=mtrans.Affine2D().scale(10) + ax.transData)
>>> print(ax.viewLim)
Bbox('array([[  0.,   0.],
          [ 90.,  90.]],
         dtype=float64)')
```

• One can now easily get a transform which goes from one transform’s coordinate system to another, in an optimized way, using the new `subtract` method on a transform. For instance, to go from data coordinates to axes coordinates:

```python
>>> import matplotlib.pyplot as plt

>>> ax = plt.axes()
>>> data2ax = ax.transData - ax.transAxes
>>> print(ax.transData.depth, ax.transAxes.depth)
3, 1
>>> print(data2ax.depth)
2
```

for versions before 1.2 this could only be achieved in a sub-optimal way, using `ax.transData + ax.transAxes.inverted()` (depth is a new concept, but had it existed it would return 4 for this example).

• `twinx` and `twiny` now returns an instance of `SubplotBase` if parent axes is an instance of `SubplotBase`.

• All Qt3-based backends are now deprecated due to the lack of py3k bindings. Qt and QtAgg backends will continue to work in v1.2.x for py2.6 and py2.7. It is anticipated that the Qt3 support will be completely removed for the next release.

• `ColorConverter`, `Colormap` and `Normalize` now subclasses `object`.

• `ContourSet` instances no longer have a `transform` attribute. Instead, access the transform with the `get_transform` method.

### 14.9 Changes in 1.1.x

• Added new `matplotlib.sankey.Sankey` for generating Sankey diagrams.

• In `imshow()`, setting `interpolation` to ‘nearest’ will now always mean that the nearest-neighbor interpolation is performed. If you want the no-op interpolation to be performed, choose ‘none’.

• There were errors in how the tri-functions were handling input parameters that had to be fixed. If your tri-plots are not working correctly anymore, or you were working around apparent mistakes, please see issue #203 in the github tracker. When in doubt, use `kwags`.
• The ‘symlog’ scale had some bad behavior in previous versions. This has now been fixed and users should now be able to use it without frustrations. The fixes did result in some minor changes in appearance for some users who may have been depending on the bad behavior.

• There is now a common set of markers for all plotting functions. Previously, some markers existed only for `scatter()` or just for `plot()`. This is now no longer the case. This merge did result in a conflict. The string ‘d’ now means “thin diamond” while ‘D’ will mean “regular diamond”.

14.10 Changes beyond 0.99.x

• The default behavior of `matplotlib.axes.Axes.set_xlim()`, `matplotlib.axes.Axes.set_ylim()`, and `matplotlib.axes.Axes.axis()` has been changed: when view limits are set explicitly with one of these methods, autoscaling is turned off for the matching axis. A new `auto` kwarg is available to control this behavior. The limit kwargs have been renamed to `left` and `right` instead of `xmin` and `xmax`, and `bottom` and `top` instead of `ymin` and `ymax`. The old names may still be used, however.

• There are five new Axes methods with corresponding pyplot functions to facilitate autoscaling, tick location, and tick label formatting, and the general appearance of ticks and tick labels:
  – `matplotlib.axes.Axes.autoscale()` turns autoscaling on or off, and applies it.
  – `matplotlib.axes.Axes.margins()` sets margins used to autoscale the `matplotlib.axes.Axes.viewLim` based on the `matplotlib.axes.Axes.dataLim`.
  – `matplotlib.axes.Axes.locator_params()` allows one to adjust axes locator parameters such as `nbins`.
  – `matplotlib.axes.Axes.ticklabel_format()` is a convenience method for controlling the `matplotlib.ticker.ScalarFormatter` that is used by default with linear axes.
  – `matplotlib.axes.Axes.tick_params()` controls direction, size, visibility, and color of ticks and their labels.

• The `matplotlib.axes.Axes.bar()` method accepts a `error_kw` kwarg; it is a dictionary of kwargs to be passed to the errorbar function.

• The `matplotlib.axes.Axes.hist()` `color` kwarg now accepts a sequence of color specs to match a sequence of datasets.

• The `EllipseCollection` has been changed in two ways:
  – There is a new `units` option, ‘xy’, that scales the ellipse with the data units. This matches the `~matplotlib.patches.Ellipse` scaling.
  – The `height` and `width` kwargs have been changed to specify the height and width, again for consistency with `Ellipse`, and to better match their names; previously they specified the half-height and half-width.

• There is a new rc parameter `axes.color_cycle`, and the color cycle is now independent of the rc parameter `lines.color`. `matplotlib.Axes.set_default_color_cycle()` is deprecated.
• You can now print several figures to one pdf file and modify the document information dictionary of a pdf file. See the docstrings of the class `matplotlib.backends.backend_pdf.PdfPages` for more information.

• Removed configobj and enthought.traits packages, which are only required by the experimental traited config and are somewhat out of date. If needed, install them independently.

• The new rc parameter `savefig.extension` sets the filename extension that is used by `matplotlib.figure.Figure.savefig()` if its `fname` argument lacks an extension.

• In an effort to simplify the backend API, all clipping rectangles and paths are now passed in using GraphicsContext objects, even on collections and images. Therefore:

```python
define_path_collection(self, master_transform, cliprect, clippath, 
    clipath_trans, paths, all_transforms, offsets, 
    offsetTrans, facecolors, edgecolors, linewidths, 
    linestyles, antialiaseds, urls)

# is now
define_path_collection(self, gc, master_transform, paths, all_transforms, 
    offsets, offsetTrans, facecolors, edgecolors, 
    linewidths, linestyles, antialiaseds, urls)

define_quad_mesh(self, master_transform, cliprect, clippath, 
    clipath_trans, meshWidth, meshHeight, coordinates, 
    offsets, offsetTrans, facecolors, antialiased, 
    showedges)

# is now
define_quad_mesh(self, gc, master_transform, meshWidth, meshHeight, 
    coordinates, offsets, offsetTrans, facecolors, 
    antialiased, showedges)

define_image(self, x, y, im, bbox, clippath=None, clippath_trans=None)

# is now
define_image(self, gc, x, y, im)
```

• There are four new Axes methods with corresponding pyplot functions that deal with unstructured triangular grids:
  
  - `matplotlib.axes.Axes.tricontour()` draws contour lines on a triangular grid.
  - `matplotlib.axes.Axes.tricontourf()` draws filled contours on a triangular grid.
  - `matplotlib.axes.Axes.tripcolor()` draws a pseudocolor plot on a triangular grid.
  - `matplotlib.axes.Axes.triplot()` draws a triangular grid as lines and/or markers.
14.11 Changes in 0.99

- pylab no longer provides a load and save function. These are available in matplotlib.mlab, or you can use numpy.loadtxt and numpy.savetxt for text files, or np.save and np.load for binary numpy arrays.

- User-generated colormaps can now be added to the set recognized by matplotlib.cm.get_cmap(). Colormaps can be made the default and applied to the current image using matplotlib.pyplot.set_cmap().

- Changed use_mrecords default to False in mlab.csv2rec since this is partially broken.

- Axes instances no longer have a “frame” attribute. Instead, use the new “spines” attribute. Spines is a dictionary where the keys are the names of the spines (e.g., ‘left’, ‘right’ and so on) and the values are the artists that draw the spines. For normal (rectilinear) axes, these artists are Line2D instances. For other axes (such as polar axes), these artists may be Patch instances.

- Polar plots no longer accept a resolution kwarg. Instead, each Path must specify its own number of interpolation steps. This is unlikely to be a user-visible change – if interpolation of data is required, that should be done before passing it to Matplotlib.

14.12 Changes for 0.98.x

- psd(), csd(), and cohere() will now automatically wrap negative frequency components to the beginning of the returned arrays. This is much more sensible behavior and makes them consistent with specgram(). The previous behavior was more of an oversight than a design decision.

- Added new keyword parameters nonposx, nonposy to matplotlib.axes.Axes methods that set log scale parameters. The default is still to mask out non-positive values, but the kwargs accept ‘clip’, which causes non-positive values to be replaced with a very small positive value.

- Added new matplotlib.pyplot.fignum_exists() and matplotlib.pyplot.get_fignums(); they merely expose information that had been hidden in matplotlib._pylab_helpers.

- Deprecated numerix package.

- Added new matplotlib.image.imsave() and exposed it to the matplotlib.pyplot interface.

- Remove support for pyExcelerator in exceltools – use xlwt instead.

- Changed the defaults of acorr and xcorr to use usevlines=True, maxlags=10 and normed=True since these are the best defaults.

- Following keyword parameters for matplotlib.label.Label are now deprecated and new set of parameters are introduced. The new parameters are given as a fraction of the font-size. Also, scatteryoffsets, fancybox and columnspacing are added as keyword parameters.

<table>
<thead>
<tr>
<th>Deprecated</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td>pad</td>
<td>borderpad</td>
</tr>
<tr>
<td>labelsep</td>
<td>labelspacing</td>
</tr>
<tr>
<td>handlelen</td>
<td>handlelength</td>
</tr>
<tr>
<td>handlestextsep</td>
<td>handlestextpad</td>
</tr>
<tr>
<td>axespad</td>
<td>borderaxespad</td>
</tr>
</tbody>
</table>
• Removed the configobj and experimental traits rc support

• Modified 
  
  `matplotlib.mlab.psd()`, `matplotlib.mlab.csd()`, `matplotlib.mlab.cohere()`, and 
  `matplotlib.mlab.specgram()` to scale one-sided densities by a factor of 2. Also, optionally 
  scale the densities by the sampling frequency, which gives true values of densities that can be 
  integrated by the returned frequency values. This also gives better MATLAB compatibility. The 
  corresponding `matplotlib.axes.Axes` methods and `matplotlib.pyplot` functions were updated as well.

• Font lookup now uses a nearest-neighbor approach rather than an exact match. Some fonts may be 
  different in plots, but should be closer to what was requested.

• `matplotlib.axes.Axes.set_xlim()`, `matplotlib.axes.Axes.set_ylim()` now return a copy 
  of the viewlim array to avoid modify-in-place surprises.

• `matplotlib.mlab.AFM.get_fullname()` and `matplotlib.mlab.AFM.get_familyname()` no 
  longer raise an exception if the AFM file does not specify these optional attributes, but returns a 
  guess based on the required FontName attribute.

• Changed precision kwarg in `matplotlib.pyplot.spy()`: default is 0, and the string value ‘present’ 
  is used for sparse arrays only to show filled locations.

• `matplotlib.collections.EllipseCollection` added.

• Added angles kwarg to `matplotlib.pyplot.quiver()` for more flexible specification of the arrow 
  angles.

• Deprecated (raise Not Implemented Error) all the mlab2 functions from `matplotlib.mlab` out of 
  concern that some of them were not clean room implementations.

• Methods `matplotlib.collections.Collection.get_offsets()` and `matplotlib.collections.Collection.set_offsets()` added to `Collection` base class.

• `matplotlib.figure.Figure.figurePatch` renamed `matplotlib.figure.Figure.patch`; 
  get_frame()`, which returns `matplotlib.axes.Axes.patch`, is deprecated.

• Changes in the `matplotlib.contour.ContourLabeler` attributes (`matplotlib.pyplot. 
  clabel()` function) so that they all have a form like .labelAttribute. The three attributes that 
  are most likely to be used by end users, .cl, .cl_xy and .cl_cvalues have been maintained for 
  the moment (in addition to their renamed versions), but they are deprecated and will eventually be 
  removed.

• Moved several functions in `matplotlib.mlab` and `matplotlib.cbook` into a separate module 
  `matplotlib.numerical_methods` because they were unrelated to the initial purpose of mlab or 
  cbook and appeared more coherent elsewhere.

### 14.13 Changes for 0.98.1

• Removed broken `matplotlib.axes3d` support and replaced it with a non-implemented error pointing 
  to 0.91.x
14.14 Changes for 0.98.0

- `matplotlib.image.imread()` now no longer always returns RGBA data—if the image is luminance or RGB, it will return a MxN or MxNx3 array if possible. Also uint8 is no longer always forced to float.
- New axes function and Axes method provide control over the plot color cycle: `matplotlib.axes.set_default_color_cycle()` and `matplotlib.axes.Axes.set_color_cycle()`.
- Matplotlib now requires Python 2.4, so `matplotlib.cbook` will no longer provide `set`, `enumerate()`, `reversed()` or `izip()` compatibility functions.
- In Numpy 1.0, bins are specified by the left edges only. The axes method `matplotlib.axes.Axes.hist()` now uses future Numpy 1.3 semantics for histograms. Providing `binedges`, the last value gives the upper-right edge now, which was implicitly set to `+infinity` in Numpy 1.0. This also means that the last bin doesn’t contain upper outliers any more by default.
- New axes method and pyplot function, `hexbin()`, is an alternative to `scatter()` for large datasets. It makes something like a `pcolor()` of a 2-D histogram, but uses hexagonal bins.
- New kwarg, `symmetric`, in `matplotlib.ticker.MaxNLocator` allows one require an axis to be centered around zero.
- Toolkits must now be imported from `mpl_toolkits` (not `matplotlib.toolkits`)

14.14.1 Notes about the transforms refactoring

A major new feature of the 0.98 series is a more flexible and extensible transformation infrastructure, written in Python/Numpy rather than a custom C extension.

The primary goal of this refactoring was to make it easier to extend matplotlib to support new kinds of projections. This is mostly an internal improvement, and the possible user-visible changes it allows are yet to come.

See `matplotlib.transforms` for a description of the design of the new transformation framework.

For efficiency, many of these functions return views into Numpy arrays. This means that if you hold on to a reference to them, their contents may change. If you want to store a snapshot of their current values, use the Numpy array method `copy()`.

The view intervals are now stored only in one place — in the `matplotlib.axes.Axes` instance, not in the locator instances as well. This means locators must get their limits from their `matplotlib.axis.Axis`, which in turn looks up its limits from the `Axes`. If a locator is used temporarily and not assigned to an Axis or Axes, (e.g., in `matplotlib.contour`), a dummy axis must be created to store its bounds. Call `matplotlib.ticker.Locator.create_dummy_axis()` to do so.
The functionality of Pbox has been merged with Bbox. Its methods now all return copies rather than modifying in place.

The following lists many of the simple changes necessary to update code from the old transformation framework to the new one. In particular, methods that return a copy are named with a verb in the past tense, whereas methods that alter an object in place are named with a verb in the present tense.
### matplotlib.transforms

<table>
<thead>
<tr>
<th>Old method</th>
<th>New method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bbox.get_bounds()</td>
<td>transforms.Bbox.bounds</td>
</tr>
<tr>
<td>Bbox.width()</td>
<td>transforms.Bbox.width</td>
</tr>
<tr>
<td>Bbox.height()</td>
<td>transforms.Bbox.height</td>
</tr>
<tr>
<td>Bbox.intervalx().get_bounds()</td>
<td>Bbox.intervalx is now a property.</td>
</tr>
<tr>
<td>Bbox.intervalx().set_bounds()</td>
<td>transforms.Bbox.intervalx</td>
</tr>
<tr>
<td>Bbox.intervaly().get_bounds()</td>
<td>Bbox.intervaly is now a property.</td>
</tr>
<tr>
<td>Bbox.intervaly().set_bounds()</td>
<td>Bbox.intervaly is now a property.</td>
</tr>
<tr>
<td>Bbox.xmin()</td>
<td>transforms.Bbox.x0 or transforms.Bbox.xmin</td>
</tr>
<tr>
<td>Bbox.ymin()</td>
<td>transforms.Bbox.y0 or transforms.Bbox.ymin</td>
</tr>
<tr>
<td>Bbox.xmax()</td>
<td>transforms.Bbox.x1 or transforms.Bbox.xmax</td>
</tr>
<tr>
<td>Bbox.ymax()</td>
<td>transforms.Bbox.y1 or transforms.Bbox.ymax</td>
</tr>
<tr>
<td>Bbox.overlaps(bboxes)</td>
<td>Bbox.count_overlaps(bboxes)</td>
</tr>
<tr>
<td>bbox_all(bboxes)</td>
<td>Bbox.union(bboxes) [transforms.Bbox.union() is a staticmethod.]</td>
</tr>
<tr>
<td>lbwh_to_bbox(l, b, w, h)</td>
<td>Bbox.from_bounds(x0, y0, w, h)</td>
</tr>
<tr>
<td></td>
<td>Bbox.from_bounds() is a staticmethod.</td>
</tr>
<tr>
<td>inverse_transform_bbox(bbox)</td>
<td>Bbox.inverse_transformed(trans)</td>
</tr>
<tr>
<td></td>
<td>Bbox.inverse_transformed(trans)</td>
</tr>
<tr>
<td>Interval.contains_open(v)</td>
<td>interval_contains_open(tuple, v)</td>
</tr>
<tr>
<td>Interval.contains(v)</td>
<td>interval_contains(tuple, v)</td>
</tr>
<tr>
<td>identity_transform()</td>
<td>matplotlib.transforms.IDENTITYTransform</td>
</tr>
<tr>
<td>blend_xy_sep_transform(xtrans, ytrans)</td>
<td>blended_transform_factory(xtrans, ytrans)</td>
</tr>
<tr>
<td>scale_transform(xs, ys)</td>
<td>Affine2D().scale(xs[, ys])</td>
</tr>
<tr>
<td>get_bbox_transform(boxin, boxout)</td>
<td>BboxTransform(boxin, boxout) or BboxTransformFrom(boxin) or BboxTransformTo(boxout)</td>
</tr>
<tr>
<td>Transform.seq_xy_tup(points)</td>
<td>Transform.transform(points)</td>
</tr>
<tr>
<td>Transform.inverse_xy_tup(points)</td>
<td>Transform.inverted().transform(points)</td>
</tr>
</tbody>
</table>

1 The Bbox is bound by the points (x0, y0) to (x1, y1) and there is no defined order to these points, that is, x0 is not necessarily the left edge of the box. To get the left edge of the Bbox, use the read-only property xmin.
```plaintext
matplotlib.axes

<table>
<thead>
<tr>
<th>Old method</th>
<th>New method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Axes.get_position()</td>
<td>matplotlib.axes.Axes.get_position()²</td>
</tr>
<tr>
<td>Axes.set_position()</td>
<td>matplotlib.axes.Axes.set_position()³</td>
</tr>
<tr>
<td>Axes.toggle_log_lineary()</td>
<td>matplotlib.axes.Axes.set_yscale()⁴</td>
</tr>
<tr>
<td>Subplot class</td>
<td>removed.</td>
</tr>
</tbody>
</table>

The Polar class has moved to matplotlib.projections.polar.

matplotlib.artist

<table>
<thead>
<tr>
<th>Old method</th>
<th>New method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Artist.set_clip_path(path)</td>
<td>Artist.set_clip_path(path, transform)⁵</td>
</tr>
</tbody>
</table>

matplotlib.collections

<table>
<thead>
<tr>
<th>Old method</th>
<th>New method</th>
</tr>
</thead>
<tbody>
<tr>
<td>linestyle</td>
<td>linestyles⁶</td>
</tr>
</tbody>
</table>

matplotlib.colors

<table>
<thead>
<tr>
<th>Old method</th>
<th>New method</th>
</tr>
</thead>
<tbody>
<tr>
<td>ColorConvertor.to_rgba_list(c)</td>
<td>ColorConvertor.to_rgba_array(c) [matplotlib.colors.ColorConvertor.to_rgba_array() returns an N×4 Numpy array of RGBA color quadruples.]</td>
</tr>
</tbody>
</table>

matplotlib.contour

<table>
<thead>
<tr>
<th>Old method</th>
<th>New method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contour._segments</td>
<td>matplotlib.contour.Contour.get_paths() [Returns a list of matplotlib.path.Path instances.]</td>
</tr>
</tbody>
</table>

matplotlib.figure

<table>
<thead>
<tr>
<th>Old method</th>
<th>New method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure.dpi.get() / Figure.dpi.set()</td>
<td>matplotlib.figure.Figure.dpi (a property)</td>
</tr>
</tbody>
</table>

² matplotlib.axes.Axes.get_position() used to return a list of points, now it returns a matplotlib.transforms.Bbox instance.

³ matplotlib.axes.Axes.set_position() now accepts either four scalars or a matplotlib.transforms.Bbox instance.

⁴ Since the refactoring allows for more than two scale types ('log' or 'linear'), it no longer makes sense to have a toggle. Axes.toggle_log_lineary() has been removed.

⁵ matplotlib.artist.Artist.set_clip_path() now accepts a matplotlib.path.Path instance and a matplotlib.transforms.Transform that will be applied to the path immediately before clipping.

⁶ Linestyles are now treated like all other collection attributes, i.e. a single value or multiple values may be provided.

14.14. Changes for 0.98.0
```
**Matplotlib, Release 2.0.2**

**matplotlib.patches**

<table>
<thead>
<tr>
<th>Old method</th>
<th>New method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Patch. get_verts()</td>
<td>matplotlib.patches.Patch.get_path() [Returns a matplotlib.path.Path instance]</td>
</tr>
</tbody>
</table>

**matplotlib.backend_bases**

<table>
<thead>
<tr>
<th>Old method</th>
<th>New method</th>
</tr>
</thead>
<tbody>
<tr>
<td>GraphicsContext.set_clip_rectangle(tuple)</td>
<td>GraphicsContext.set_clip_rectangle(bbox)</td>
</tr>
<tr>
<td>GraphicsContext.get_clip_path()</td>
<td>GraphicsContext.get_clip_path()</td>
</tr>
<tr>
<td>GraphicsContext.set_clip_path()</td>
<td>GraphicsContext.set_clip_path()</td>
</tr>
</tbody>
</table>

**RendererBase**

New methods:

- `draw_path(self, gc, path, transform, rgbFace)`
- `draw_markers(self, gc, marker_path, marker_trans, path, trans, rgbFace)`
- `draw_path_collection(self, master_transform, cliprect, clippath, clippath_trans, paths, all_transforms, offsets, offsetTrans, facecolors, edgecolors, linwidths, linestyles, antialiaseds) [optional]`

Changed methods:

- `draw_image(self, x, y, im, bbox)` is now `draw_image(self, x, y, im, bbox, clippath, clippath_trans)`

Removed methods:

- `draw_arc`
- `draw_line_collection`
- `draw_line`
- `draw_lines`
- `draw_point`
- `draw_quad_mesh`
- `draw_poly_collection`
- `draw_polygon`

---

7 matplotlib.backend_bases.GraphicsContext.get_clip_path() returns a tuple of the form (path, affine_transform), where path is a matplotlib.path.Path instance and affine_transform is a matplotlib.transforms.Affine2D instance.

8 matplotlib.backend_bases.GraphicsContext.set_clip_path() now only accepts a matplotlib.transforms.TransformedPath instance.
• draw_rectangle
• draw_regpoly_collection

14.15 Changes for 0.91.2

• For csv2rec(), checkrows=0 is the new default indicating all rows will be checked for type inference
• A warning is issued when an image is drawn on log-scaled axes, since it will not log-scale the image data.
• Moved rec2gtxk() to matplotlib.toolkits.gtktools
• Moved rec2excel() to matplotlib.toolkits.exceltools
• Removed, dead/experimental ExampleInfo, Namespace and Importer code from matplotlib.__init__

14.16 Changes for 0.91.1

14.17 Changes for 0.91.0

• Changed cbook.is_file_like() to cbook.is_writable_file_like() and corrected behavior.
• Added ax kwarg to pyplot.colorbar() and Figure.colorbar() so that one can specify the axes object from which space for the colorbar is to be taken, if one does not want to make the colorbar axes manually.
• Changed cbook.reversed() so it yields a tuple rather than a (index, tuple). This agrees with the python reversed builtin, and cbook only defines reversed if python doesn’t provide the builtin.
• Made skiprows=1 the default on csv2rec()
• The gd and paint backends have been deleted.
• The errorbar method and function now accept additional kwargs so that upper and lower limits can be indicated by capping the bar with a caret instead of a straight line segment.
• The matplotlib.dviread file now has a parser for files like psfonts.map and pdftex.map, to map TeX font names to external files.
• The file matplotlib.type1font contains a new class for Type 1 fonts. Currently it simply reads pfa and pfb format files and stores the data in a way that is suitable for embedding in pdf files. In the future the class might actually parse the font to allow e.g., subsetting.
• matplotlib.FT2Font now supports FT_Attach_File(). In practice this can be used to read an afm file in addition to a pfa/pfb file, to get metrics and kerning information for a Type 1 font.
• The AFM class now supports querying CapHeight and stem widths. The get_name_char method now has an isord kwarg like get_width_char.
• Changed `pcolor()` default to shading='flat'; but as noted now in the docstring, it is preferable to simply use the edgecolor kwarg.

• The mathtext font commands (\cal, \rm, \it, \tt) now behave as TeX does: they are in effect until the next font change command or the end of the grouping. Therefore uses of $\cal{R}\$ should be changed to $\{\cal R\}$. Alternatively, you may use the new LaTeX-style font commands \mathcal, \mathrm, \mathit, \mathtt) which do affect the following group, e.g., $\mathcal{R}\$.

• Text creation commands have a new default linespacing and a new linespacing kwarg, which is a multiple of the maximum vertical extent of a line of ordinary text. The default is 1.2; linespacing=2 would be like ordinary double spacing, for example.

• Changed default kwarg in `matplotlib.colors.Normalize.__init__` to clip=False; clipping silently defeats the purpose of the special over, under, and bad values in the colormap, thereby leading to unexpected behavior. The new default should reduce such surprises.

• Made the emit property of `set_xlim()` and `set_ylim()` True by default; removed the Axes custom callback handling into a ‘callbacks’ attribute which is a CallbackRegistry instance. This now supports the ’xlim_changed’ and ’ylim_changed’ Axes events.

### 14.18 Changes for 0.90.1

The file dviread.py has a (very limited and fragile) dvi reader for usetex support. The API might change in the future so don’t depend on it yet.

Removed deprecated support for a float value as a gray-scale; now it must be a string, like ’0.5’. Added alpha kwarg to ColorConverter.to_rgba_list.

New method `set_bounds(vmin, vmax)` for formatters, locators sets the viewInterval and dataInterval from floats.

Removed deprecated `colorbar_classic`.

`Line2D.get_xdata` and `get_ydata valid_only=False` kwarg is replaced by `orig=True`. When True, it returns the original data, otherwise the processed data (masked, converted)

Some modifications to the units interface.

`units.ConversionInterface.tickers` renamed to `units.ConversionInterface.axisinfo` and it now returns a `units.AxisInfo object` rather than a tuple. This will make it easier to add axis info functionality (e.g., I added a default label on this iteration) w/o having to change the tuple length and hence the API of the client code every time new functionality is added.

Also, `units.ConversionInterface.convert_to_value` is now simply named `units.ConversionInterface.convert`.

Axes.errorbar uses `Axes.vlines` and `Axes.hlines` to draw its error
limits in the vertical and horizontal direction. As you'll see in the changes below, these functions now return a LineCollection rather than a list of lines. The new return signature for errorbar is ylins, caplines, errorcollections where errorcollections is a xerrcollection, yerrcollection

Axes.vlines and Axes.hlines now create and returns a LineCollection, not a list of lines. This is much faster. The kwarg signature has changed, so consult the docs

MaxNLocator accepts a new Boolean kwarg (integer) to force ticks to integer locations.

Commands that pass an argument to the Text constructor or to Text.set_text() now accept any object that can be converted with '%%s'. This affects xlabel(), title(), etc.

Barh now takes a **kwargs dict instead of most of the old arguments. This helps ensure that bar and barh are kept in sync, but as a side effect you can no longer pass e.g., color as a positional argument.

ft2font.get_charmap() now returns a dict that maps character codes to glyph indices (until now it was reversed)

Moved data files into lib/matplotlib so that setuptools' develop mode works. Re-organized the mpl-data layout so that this source structure is maintained in the installation. (i.e., the 'fonts' and 'images' sub-directories are maintained in site-packages.). Suggest removing site-packages/matplotlib/mpl-data and ~/.matplotlib/ttffont.cache before installing

14.19 Changes for 0.90.0

All artists now implement a “pick” method which users should not call. Rather, set the "picker" property of any artist you want to pick on (the epsilon distance in points for a hit test) and register with the "pick_event" callback. See examples/pick_event_demo.py for details

Bar, barh, and hist have "log" binary kwarg: log=True sets the ordinate to a log scale.

Boxplot can handle a list of vectors instead of just an array, so vectors can have different lengths.

Plot can handle 2-D x and/or y; it plots the columns.

Added linewidth kwarg to bar and barh.
Made the default Artist._transform None (rather than invoking identity_transform for each artist only to have it overridden later). Use artist.get_transform() rather than artist._transform, even in derived classes, so that the default transform will be created lazily as needed.

New LogNorm subclass of Normalize added to colors.py. All Normalize subclasses have new inverse() method, and the __call__() method has a new clip kwarg.

Changed class names in colors.py to match convention: normalize -> Normalize, no_norm -> NoNorm. Old names are still available for now.

Removed obsolete pcolor_classic command and method.

Removed lineprops and markerprops from the Annotation code and replaced them with an arrow configurable with kwarg arrowprops. See examples/annotation_demo.py - JDH

14.20 Changes for 0.87.7

Completely reworked the annotations API because I found the old API cumbersome. The new design is much more legible and easy to read. See matplotlib.text.Annotation and examples/annotation_demo.py

markeredgecolor and markerfacecolor cannot be configured in matplotlibrc any more. Instead, markers are generally colored automatically based on the color of the line, unless marker colors are explicitly set as kwargs - NN

Changed default comment character for load to '#' - JDH

math_parse_s_ft2font_svg from mathtext.py & mathtext2.py now returns width, height, svg_elements. svg_elements is an instance of Bunch (cmbook.py) and has the attributes svg_glyphs and svg_lines, which are both lists.

Renderer.draw_arc now takes an additional parameter, rotation. It specifies to draw the artist rotated in degrees anti-clockwise. It was added for rotated ellipses.

Renamed Figure.set_figsizes to Figure.set_size_inches to better match the get method, Figure.get_size_inches.

Removed the copy_bbox_transform from transforms.py; added shallowcopy methods to all transforms. All transforms already had deepcopy methods.
FigureManager.resize(width, height): resize the window specified in pixels.

barh: x and y args have been renamed to width and bottom respectively, and their order has been swapped to maintain a (position, value) order.

bar and barh: now accept kwarg 'edgecolor'.

bar and barh: The left, height, width and bottom args can now all be scalars or sequences; see docstring.

barh: now defaults to edge aligned instead of center aligned bars.

bar, barh and hist: Added a keyword arg 'align' that controls between edge or center bar alignment.

Collections: PolyCollection and LineCollection now accept vertices or segments either in the original form [(x,y), (x,y), ...] or as a 2D numerix array, with X as the first column and Y as the second. Contour and quiver output the numerix form. The transforms methods Bbox.update() and Transformation.seq_xy_tups() now accept either form.

Collections: LineCollection is now a ScalarMappable like PolyCollection, etc.

Specifying a grayscale color as a float is deprecated; use a string instead, e.g., 0.75 -> '0.75'.

Collections: initializers now accept any mpl color arg, or sequence of such args; previously only a sequence of rgba tuples was accepted.

Colorbar: completely new version and api; see docstring. The original version is still accessible as colorbar_classic, but is deprecated.

Contourf: "extend" kwarg replaces "clip_ends"; see docstring. Masked array support added to pcolormesh.

Modified aspect-ratio handling:
- Removed aspect kwarg from imshow
- Axes methods:
  - set_aspect(self, aspect, adjustable=None, anchor=None)
  - set_adjustable(self, adjustable)
  - set_anchor(self, anchor)
- Pylab interface:
  - axis('image')

Backend developers: ft2font's load_char now takes a flags argument, which you can OR together from the LOAD_XXX
14.21 Changes for 0.86

Matplotlib data is installed into the matplotlib module. This is similar to package_data. This should get rid of having to check for many possibilities in _get_data_path(). The MATPLOTLIBDATA env key is still checked first to allow for flexibility.

1) Separated the color table data from cm.py out into a new file, _cm.py, to make it easier to find the actual code in cm.py and to add new colormaps. Everything from _cm.py is imported by cm.py, so the split should be transparent.
2) Enabled automatic generation of a colormap from a list of colors in contour; see modified examples/contour_demo.py.
3) Support for imshow of a masked array, with the ability to specify colors (or no color at all) for masked regions, and for regions that are above or below the normally mapped region. See examples/image_masked.py.
4) In support of the above, added two new classes, ListedColormap, and no_norm, to colors.py, and modified the Colormap class to include common functionality. Added a clip kwarg to the normalize class.

14.22 Changes for 0.85

Made xtick and ytick separate props in rc

made pos=None the default for tick formaters rather than 0 to indicate "not supplied"

Removed "feature" of minor ticks which prevents them from overlapping major ticks. Often you want major and minor ticks at the same place, and can offset the major ticks with the pad. This could be made configurable

Changed the internal structure of contour.py to a more OO style. Calls to contour or contourf in axes.py or pylab.py now return a ContourSet object which contains references to the LineCollections or PolyCollections created by the call, as well as the configuration variables that were used. The ContourSet object is a "mappable" if a colormap was used.

Added a clip_ends kwarg to contourf. From the docstring:
* clip_ends = True
  If False, the limits for color scaling are set to the minimum and maximum contour levels.
  True (default) clips the scaling limits. Example:
  if the contour boundaries are V = [-100, 2, 1, 0, 1, 2, 100],
  then the scaling limits will be [-100, 100] if clip_ends
  is False, and [-3, 3] if clip_ends is True.

Added kwargs linewidths, antialiased, and nchunk to contourf. These are experimental; see the docstring.

Changed Figure.colorbar():
  kw argument order changed;
  if mappable arg is a non-filled ContourSet, colorbar() shows
  lines instead hof polygons.
  if mappable arg is a filled ContourSet with clip_ends=True,
  the endpoints are not labelled, so as to give the
  correct impression of open-endedness.

Changed LineCollection.get_linewidths to get_linewidth, for
consistency.

14.23 Changes for 0.84

Unified argument handling between hlines and vlines. Both now
take optionally a fmt argument (as in plot) and a keyword args
that can be passed onto Line2D.

Removed all references to "data clipping" in rc and lines.py since
these were not used and not optimized. I'm sure they'll be
resurrected later with a better implementation when needed.

'set' removed - no more deprecation warnings. Use 'setp' instead.

Backend developers: Added flipud method to image and removed it
from to_str. Removed origin kwarg from backend.draw_image.
origin is handled entirely by the frontend now.

14.24 Changes for 0.83

- Made HOME/.matplotlib the new config dir where the matplotlibrc
  file, the ttf.cache, and the tex.cache live. The new default
  filenames in .matplotlib have no leading dot and are not hidden.
  e.g., the new names are matplotlibrc, tex.cache, and ttf.ttf
  font.cache. This is how ipython does it so it must be right.

  If old files are found, a warning is issued and they are moved to
  the new location.
- backends/__init__.py no longer imports new_figure_manager, draw_if_interactive and show from the default backend, but puts these imports into a call to pylab_setup. Also, the Toolbar is no longer imported from WX/WXAgg. New usage:

    from backends import pylab_setup
    new_figure_manager, draw_if_interactive, show = pylab_setup()

- Moved Figure.get_width_height() to FigureCanvasBase. It now returns int instead of float.

### 14.25 Changes for 0.82

- toolbar import change in GTKAgg, GTKCairo and WXAgg

- Added subplot config tool to GTK* backends -- note you must now import the NavigationToolbar2 from your backend of choice rather than from backend_gtk because it needs to know about the backend specific canvas -- see examples/embedding_in_gtk2.py. Ditto for wx backend -- see examples/embedding_in_wxagg.py

- hist bin change

  Sean Richards notes there was a problem in the way we created the binning for histogram, which made the last bin underrepresented. From his post:

  I see that hist uses the linspace function to create the bins and then uses searchsorted to put the values in their correct bin. That's all good but I am confused over the use of linspace for the bin creation. I wouldn't have thought that it does what is needed, to quote the docstring it creates a "Linear spaced array from min to max". For it to work correctly shouldn't the values in the bins array be the same bound for each bin? (i.e. each value should be the lower bound of a bin). To provide the correct bins for hist would it not be something like

  ```python
  def bins(xmin, xmax, N):
      if N==1: return xmax
      dx = (xmax-xmin)/N # instead of N-1
      return xmin + dx*arange(N)
  ```

  This suggestion is implemented in 0.81. My test script with these changes does not reveal any bias in the binning

    from matplotlib.mlab import randn, rand, zeros, Float
    from matplotlib.mlab import import hist, mean
Nbins = 50
Ntests = 200
results = zeros((Ntests,Nbins), typecode=Float)
for i in range(Ntests):
    print 'computing', i
    x = rand(10000)
    n, bins = hist(x, Nbins)
    results[i] = n
print mean(results)

14.26 Changes for 0.81

- pylab and artist "set" functions renamed to setp to avoid clash
  with python2.4 built-in set. Current version will issue a
  deprecation warning which will be removed in future versions

- imshow interpolation arguments changes for advanced interpolation
  schemes. See help imshow, particularly the interpolation,
  filternorm and filterrad kwargs

- Support for masked arrays has been added to the plot command and
  to the Line2D object. Only the valid points are plotted. A
  "valid_only" kwarg was added to the get_xdata() and get_ydata()
  methods of Line2D; by default it is False, so that the original
  data arrays are returned. Setting it to True returns the plottable
  points.

- contour changes:

  Masked arrays: contour and contourf now accept masked arrays as
  the variable to be contoured. Masking works correctly for
  contour, but a bug remains to be fixed before it will work for
  contourf. The "badmask" kwarg has been removed from both
  functions.

  Level argument changes:

  Old version: a list of levels as one of the positional
  arguments specified the lower bound of each filled region; the
  upper bound of the last region was taken as a very large
  number. Hence, it was not possible to specify that z values
  between 0 and 1, for example, be filled, and that values
  outside that range remain unfilled.

  New version: a list of N levels is taken as specifying the
  boundaries of N-1 z ranges. Now the user has more control over
  what is colored and what is not. Repeated calls to contourf
  (with different colormaps or color specifications, for example)
  can be used to color different ranges of z. Values of z
outside an expected range are left uncolored.

Example:
Old: contourf(z, [0, 1, 2]) would yield 3 regions: 0-1, 1-2, and >2.
New: it would yield 2 regions: 0-1, 1-2. If the same 3 regions were desired, the equivalent list of levels would be [0, 1, 2, 1e38].

### 14.27 Changes for 0.80

- xlim/ylim/axis always return the new limits regardless of arguments. They now take kwargs which allow you to selectively change the upper or lower limits while leaving unnamed limits unchanged. See help(xlim) for example

### 14.28 Changes for 0.73

- Removed deprecated ColormapJet and friends
- Removed all error handling from the verbose object
- figure num of zero is now allowed

### 14.29 Changes for 0.72

- Line2D, Text, and Patch copy_properties renamed update_from and moved into artist base class
- LineCollection.color renamed to LineCollections.set_color for consistency with set/get introspection mechanism,
- pylab figure now defaults to num=None, which creates a new figure with a guaranteed unique number
- contour method syntax changed - now it is MATLAB compatible
  
  unchanged: contour(Z)
  old: contour(Z, x=Y, y=Y)
  new: contour(X, Y, Z)

  see http://matplotlib.sf.net/matplotlib.pylab.html#contour

- Increased the default resolution for save command.
- Renamed the base attribute of the ticker classes to _base to avoid conflict with the base method. Sitt for subs

- subs=none now does autosubbing in the tick locator.

- New subplots that overlap old will delete the old axes. If you do not want this behavior, use fig.add_subplot or the axes command

14.30 Changes for 0.71

Significant numerix namespace changes, introduced to resolve namespace clashes between python built-ins and mlab names. Refactored numerix to maintain separate modules, rather than folding all these names into a single namespace. See the following mailing list threads for more information and background:


OLD usage

from matplotlib.numerix import array, mean, fft

NEW usage

from matplotlib.numerix import array
from matplotlib.numerix.mlab import mean
from matplotlib.numerix.fft import fft

numerix dir structure mirrors numarray (though it is an incomplete implementation)

numerix
numerix/mlab
numerix/linear_algebra
numerix/fft
numerix/random_array

but of course you can use 'numerix : Numeric' and still get the symbols.

pylab still imports most of the symbols from Numerix, MLab, fft, etc, but is more cautious. For names that clash with python names (min, max, sum), pylab keeps the builtins and provides the numeric versions with an a* prefix, e.g., (amin, amax, asum)
14.31 Changes for 0.70

MplEvent factored into a base class Event and derived classes
MouseEvent and KeyEvent

Removed definct set_measurement in wx toolbar

14.32 Changes for 0.65.1

removed add_axes and add_subplot from backend_bases. Use
figure.add_axes and add_subplot instead. The figure now manages the
current axes with gca and sca for get and set current axes. If you
have code you are porting which called, e.g., figmanager.add_axes, you
can now simply do figmanager.canvas.figure.add_axes.

14.33 Changes for 0.65

mpl_connect and mpl_disconnect in the MATLAB interface renamed to
connect and disconnect

Did away with the text methods for angle since they were ambiguous.
fontangle could mean fontstyle (oblique, etc) or the rotation of the
text. Use style and rotation instead.

14.34 Changes for 0.63

Dates are now represented internally as float days since 0001-01-01,
UTC.

All date tickers and formatters are now in matplotlib.dates, rather
than matplotlib.tickers

converters have been abolished from all functions and classes.
num2date and date2num are now the converter functions for all date
plots

Most of the date tick locators have a different meaning in their
constructors. In the prior implementation, the first argument was a
base and multiples of the base were ticked. e.g.,

HourLocator(5) # old: tick every 5 minutes

In the new implementation, the explicit points you want to tick are
provided as a number or sequence
HourLocator(range(0,5,61))  # new: tick every 5 minutes

This gives much greater flexibility. I have tried to make the default constructors (no args) behave similarly, where possible.

Note that YearLocator still works under the base/multiple scheme. The difference between the YearLocator and the other locators is that years are not recurrent.

Financial functions:

    matplotlib.finance.quotes_historical_yahoo(ticker, date1, date2)

date1, date2 are now datetime instances. Return value is a list of quotes where the quote time is a float - days since gregorian start, as returned by date2num

See examples/finance_demo.py for example usage of new API

14.35 Changes for 0.61

canvas.connect is now deprecated for event handling. use
mpl_connect and mpl_disconnect instead. The callback signature is
func(event) rather than func(widget, event)

14.36 Changes for 0.60

ColormapJet and Grayscale are deprecated. For backwards
compatibility, they can be obtained either by doing

    from matplotlib.cm import ColormapJet

or

    from matplotlib.matlab import *

They are replaced by cm.jet and cm.grey

14.37 Changes for 0.54.3

removed the set_default_font / get_default_font scheme from the
font_manager to unify customization of font defaults with the rest of
the rc scheme. See examples/font_properties_demo.py and help(rc) in matplotlib.matlab.

14.38 Changes for 0.54

14.38.1 MATLAB interface

dpi

Several of the backends used a PIXELS_PER_INCH hack that I added to try and make images render consistently across backends. This just complicated matters. So you may find that some font sizes and line widths appear different than before. Apologies for the inconvenience. You should set the dpi to an accurate value for your screen to get true sizes.

pcolor and scatter

There are two changes to the MATLAB interface API, both involving the patch drawing commands. For efficiency, pcolor and scatter have been rewritten to use polygon collections, which are a new set of objects from matplotlib.collections designed to enable efficient handling of large collections of objects. These new collections make it possible to build large scatter plots or pcolor plots with no loops at the python level, and are significantly faster than their predecessors. The original pcolor and scatter functions are retained as pcolor_classic and scatter_classic.

The return value from pcolor is a PolyCollection. Most of the properties that are available on rectangles or other patches are also available on PolyCollections, e.g., you can say:

```
c = scatter(blah, blah)
c.set_linewidth(1.0)
c.set_facecolor('r')
c.set_alpha(0.5)
```

or:

```
c = scatter(blah, blah)
set(c, 'linwidth', 1.0, 'facecolor', 'r', 'alpha', 0.5)
```

Because the collection is a single object, you no longer need to loop over the return value of scatter or pcolor to set properties for the entire list.

If you want the different elements of a collection to vary on a property, e.g., to have different line widths, see matplotlib.collections for a discussion on how to set the properties as a sequence.

For scatter, the size argument is now in points^2 (the area of the symbol in points) as in MATLAB and is not in data coords as before. Using sizes in data coords caused several problems. So you will need to adjust your size arguments accordingly or use scatter_classic.
mathtext spacing

For reasons not clear to me (and which I’ll eventually fix) spacing no longer works in font groups. However, I added three new spacing commands which compensate for this ‘ ‘ (regular space), ‘/’ (small space) and ‘hspace{frac}’ where frac is a fraction of fontsize in points. You will need to quote spaces in font strings, is:

```
title(r'$\text{Histogram of IQ:} \ \mu=100, \ \sigma=15$')
```

### 14.38.2 Object interface - Application programmers

#### Autoscaling

The x and y axis instances no longer have autoscale view. These are handled by `axes.autoscale_view`

#### Axes creation

You should not instantiate your own Axes any more using the OO API. Rather, create a Figure as before and in place of:

```
f = Figure(figsize=(5,4), dpi=100)
a = Subplot(f, 111)
f.add_axis(a)
```

use:

```
f = Figure(figsize=(5,4), dpi=100)
a = f.add_subplot(111)
```

That is, `add_axis` no longer exists and is replaced by:

```
add_axes(rect, axisbg=defaultcolor, frameon=True)
add_subplot(num, axisbg=defaultcolor, frameon=True)
```

#### Artist methods

If you define your own Artists, you need to rename the `_draw` method to draw

#### Bounding boxes

`matplotlib.transforms.Bound2D` is replaced by `matplotlib.transforms.Bbox`. If you want to construct a bbox from left, bottom, width, height (the signature for Bound2D), use `matplotlib.transforms.lbwh_to_bbox`, as in

```
bbox = clickBBBox = lbwh_to_bbox(left, bottom, width, height)
```
The Bbox has a different API than the Bound2D. e.g., if you want to get the width and height of the bbox

**OLD::**  
width = fig.bbox.x.interval()  
height = fig.bbox.y.interval()  

**New::**  
width = fig.bbox.width()  
height = fig.bbox.height()

Object constructors

You no longer pass the bbox, dpi, or transforms to the various Artist constructors. The old way or creating lines and rectangles was cumbersome because you had to pass so many attributes to the Line2D and Rectangle classes not related directly to the geometry and properties of the object. Now default values are added to the object when you call axes.add_line or axes.add_patch, so they are hidden from the user.

If you want to define a custom transformation on these objects, call o.set_transform(trans) where trans is a Transformation instance.

In prior versions of you wanted to add a custom line in data coords, you would have to do

```python
l = Line2D(dpi, bbox, x, y, color=color, transx=transx, transy=transy, )
```

now all you need is

```python
l = Line2D(x, y, color=color)
```

and the axes will set the transformation for you (unless you have set your own already, in which case it will eave it unchanged)

Transformations

The entire transformation architecture has been rewritten. Previously the x and y transformations where stored in the xaxis and yaxis instances. The problem with this approach is it only allows for separable transforms (where the x and y transformations don’t depend on one another). But for cases like polar, they do. Now transformations operate on x,y together. There is a new base class matplotlib.transforms.Transformation and two concrete implementations, matplotlib.transforms.SeparableTransformation and matplotlib.transforms.Affine. The SeparableTransformation is constructed with the bounding box of the input (this determines the rectangular coordinate system of the input, i.e., the x and y view limits), the bounding box of the display, and possibly nonlinear transformations of x and y. The 2 most frequently used transformations, data coordinates -> display and axes coordinates -> display are available as ax.transData and ax.transAxes. See alignment_demo.py which uses axes coords.

Also, the transformations should be much faster now, for two reasons

- they are written entirely in extension code
- because they operate on x and y together, they can do the entire transformation in one loop. Earlier I did something along the lines of:

```python
xt = sx*func(x) + tx
yt = sy*func(y) + ty
```
Although this was done in numerix, it still involves 6 length(x) for-loops (the multiply, add, and function evaluation each for x and y). Now all of that is done in a single pass.

If you are using transformations and bounding boxes to get the cursor position in data coordinates, the method calls are a little different now. See the updated examples/coords_demo.py which shows you how to do this.

Likewise, if you are using the artist bounding boxes to pick items on the canvas with the GUI, the bbox methods are somewhat different. You will need to see the updated examples/object_picker.py.

See unit/transforms_unit.py for many examples using the new transformations.

**14.39 Changes for 0.50**

* refactored Figure class so it is no longer backend dependent. FigureCanvasBackend takes over the backend specific duties of the Figure. matplotlib.backend_bases.FigureBase moved to matplotlib.figure.Figure.

* backends must implement FigureCanvasBackend (the thing that controls the figure and handles the events if any) and FigureManagerBackend (wraps the canvas and the window for MATLAB interface). FigureCanvasBase implements a backend switching mechanism

* Figure is now an Artist (like everything else in the figure) and is totally backend independent

* GDFONTPATH renamed to TTFPATH

* backend faceColor argument changed to rgbFace

* colormap stuff moved to colors.py

* arg_to_rgb in backend_bases moved to class ColorConverter in colors.py

* GD users must upgrade to gd-2.0.22 and gdmodule-0.52 since new gd features (clipping, antialiased lines) are now used.

* Renderer must implement points_to_pixels

Migrating code:

MATLAB interface:

The only API change for those using the MATLAB interface is in how you call figure redraws for dynamically updating figures. In the old API, you did

```python
fig.draw()
```
In the new API, you do

```python
manager = get_current_fig_manager()
manager.canvas.draw()
```

See the examples `system_monitor.py`, `dynamic_demo.py`, and `anim.py`

**API**

There is one important API change for application developers. Figure instances used subclass GUI widgets that enabled them to be placed directly into figures. e.g., `FigureGTK` subclassed `gtk.DrawingArea`. Now the Figure class is independent of the backend, and `FigureCanvas` takes over the functionality formerly handled by Figure. In order to include figures into your apps, you now need to do, for example

```python
# gtk example
fig = Figure(figsize=(5,4), dpi=100)
canvas = FigureCanvasGTK(fig)  # a `gtk.DrawingArea`
canvas.show()
 vbox.pack_start(canvas)
```

If you use the NavigationToolbar, this is now initialized with a `FigureCanvas`, not a `Figure`. The examples `embedding_in_gtk.py`, `embedding_in_gtk2.py`, and `mpl_with_glade.py` all reflect the new API so use these as a guide.

All prior calls to

```python
figure.draw() and
figure.print_figure(args)
```

should now be

```python
canvas.draw() and
canvas.print_figure(args)
```

Apologies for the inconvenience. This refactoring brings significant more freedom in developing `matplotlib` and should bring better plotting capabilities, so I hope the inconvenience is worth it.

### 14.40 Changes for 0.42

- Refactoring `AxisText` to be backend independent. Text drawing and `get_window_extent` functionality will be moved to the Renderer.
- `backend_bases.AxisTextBase` is now `text.Text` module
* All the erase and reset functionality removed from `AxisText` - not needed with double buffered drawing. Ditto with state change. Text instances have a `get_prop_tup` method that returns a hashable tuple of text properties which you can use to see if text props have changed, e.g., by caching a font or layout instance in a dict with the prop tup as a key -- see `RendererGTK.get_pango_layout` in `backend_gtk` for an example.

* Text._get_xy_display renamed Text.get_xy_display

* Artist `set_renderer` and `wash_brushes` methods removed

* Moved `Legend` class from `matplotlib.axes` into `matplotlib.legend`

* Moved `Tick`, `XTick`, `YTick`, `Axis`, `XAxis`, `YAxis` from `matplotlib.axes` to `matplotlib.axis`

* moved `process_text_args` to `matplotlib.text`

* After getting Text handled in a backend independent fashion, the import process is much cleaner since there are no longer cyclic dependencies

* `matplotlib.matlab._get_current_fig_manager` renamed to `matplotlib.matlab.get_current_fig_manager` to allow user access to the GUI window attribute, e.g., `figManager.window` for GTK and `figManager.frame` for wx

---

### 14.41 Changes for 0.40

- **Artist**
  - `__init__` takes a DPI instance and a Bound2D instance which is the bounding box of the artist in display coords
  - `get_window_extent` returns a Bound2D instance
  - `set_size` is removed; replaced by `bbox` and `dpi`
  - the `clip_gc` method is removed. Artists now clip themselves with their box
  - added `_clipOn` boolean attribute. If `True`, gc clip to `bbox`.

- **AxisTextBase**
  - Initialized with a transx, transy which are Transform instances
  - `set_drawing_area` removed
  - `get_left_right` and `get_top_bottom` are replaced by `get_window_extent`

- **Line2D Patches** now take transx, transy
  - Initialized with a transx, transy which are Transform instances

- **Patches**
  - Initialized with a transx, transy which are Transform instances
- FigureBase attributes dpi is a DPI instance rather than scalar and new attribute bbox is a Bound2D in display coords, and I got rid of the left, width, height, etc... attributes. These are now accessible as, for example, bbox.x.min is left, bbox.x.interval() is width, bbox.y.max is top, etc...

- GcfBase attribute pagesize renamed to figsize

- Axes
  * removed figbg attribute
  * added fig instance to __init__
  * resizing is handled by figure call to resize.

- Subplot
  * added fig instance to __init__

- Renderer methods for patches now take gcEdge and gcFace instances. gcFace=None takes the place of filled=False

- True and False symbols provided by cbook in a python2.3 compatible way

- new module transforms supplies Bound1D, Bound2D and Transform instances and more

- Changes to the MATLAB helpers API

  * _matlab_helpers.GcfBase is renamed by Gcf. Backends no longer need to derive from this class. Instead, they provide a factory function new_figure_manager(num, figsize, dpi). The destroy method of the GcfDerived from the backends is moved to the derived FigureManager.

  * FigureManagerBase moved to backend_bases

  * Gcf.get_all_figwins renamed to Gcf.get_all_fig_managers

Jeremy:

Make sure to self._reset = False in AxisTextWX._set_font. This was something missing in my backend code.
 CHAPTER
FIFTEEN

THE TOP LEVEL MATPLOTLIB MODULE

```python
matplotlib.use(arg, warn=True, force=False)
```
Set the matplotlib backend to one of the known backends.

The argument is case-insensitive. `warn` specifies whether a warning should be issued if a backend has already been set up. `force` is an experimental flag that tells matplotlib to attempt to initialize a new backend by reloading the backend module.

**Note:** This function must be called before importing pyplot for the first time; or, if you are not using pyplot, it must be called before importing matplotlib.backends. If `warn` is True, a warning is issued if you try and call this after pylab or pyplot have been loaded. In certain black magic use cases, e.g. `pyplot.switch_backend()`, we are doing the reloading necessary to make the backend switch work (in some cases, e.g., pure image backends) so one can set `warn=False` to suppress the warnings.

To find out which backend is currently set, see `matplotlib.get_backend()`.

```python
matplotlib.get_backend()
```
Return the name of the current backend.

```python
matplotlib.rcParams
```
An instance of `RcParams` for handling default matplotlib values.

```python
matplotlib.rc(group, **kwargs)
```
Set the current rc params. Group is the grouping for the rc, e.g., for `lines.linewidth` the group is `lines`, for `axes.facecolor`, the group is `axes`, and so on. Group may also be a list or tuple of group names, e.g., `(xtick, ytick)`. `kwargs` is a dictionary attribute name/value pairs, e.g.,:

```python
rc('lines', linewidth=2, color='r')
```

sets the current rc params and is equivalent to:

```python
rcParams['lines.linewidth'] = 2
rcParams['lines.color'] = 'r'
```

The following aliases are available to save typing for interactive users:
Thus you could abbreviate the above rc command as:

```python
cr('lines', lw=2, c='r')
```

Note you can use python’s kwargs dictionary facility to store dictionaries of default parameters. e.g., you can customize the font rc as follows:

```python
default_font = {'family': 'monospace',
                 'weight': 'bold',
                 'size' : 'larger'}
rc('font', **default_font) # pass in the font dict as kwargs
```

This enables you to easily switch between several configurations. Use `matplotlib.style.use('default')` or `rcdefaults()` to restore the default rc params after changes.

### `matplotlib.matplotlib_fname()`

Get the location of the config file.

The file location is determined in the following order:

- `$PWD/matplotlibrc`
- `$MATPLOTLIBRC/matplotlibrc`
- `$MPLCONFIGDIR/matplotlibrc`
- On Linux, $HOME/.matplotlib/matplotlibrc, if it exists
- or `$XDG_CONFIG_HOME/matplotlib/matplotlibrc` (if `$XDG_CONFIG_HOME` is defined)
- or `$HOME/.config/matplotlib/matplotlibrc` (if `$XDG_CONFIG_HOME` is not defined)
- On other platforms,
  - `$HOME/.matplotlib/matplotlibrc` if `$HOME` is defined.
- Lastly, it looks in `$MATPLOTLIBDATA/matplotlibrc` for a system-defined copy.

### `class matplotlib.RcParams(*args, **kwargs)`

A dictionary object including validation validating functions are defined and associated with rc parameters in `matplotlib.rcsetup`
matplotlib.rc_params(fail_on_error=False)
    Return a matplotlib.RcParams instance from the default matplotlib rc file.

matplotlib.rc_params_from_file(fname, fail_on_error=False, use_default_template=True)
    Return matplotlib.RcParams from the contents of the given file.

    Parameters
    ----------
    fname : str
        Name of file parsed for matplotlib settings.

    fail_on_error : bool
        If True, raise an error when the parser fails to convert a parameter.

    use_default_template : bool
        If True, initialize with default parameters before updating with those in the
given file. If False, the configuration class only contains the parameters spec-
ified in the file. (Useful for updating dicts.)

class matplotlib.rc_context(rc=None, fname=None)
    Return a context manager for managing rc settings.

    This allows one to do:

    .. code::

        with mpl.rc_context(fname='screen.rc'):
            plt.plot(x, a)
        with mpl.rc_context(fname='print.rc'):
            plt.plot(x, b)
        plt.plot(x, c)

    The 'a' vs 'x' and 'c' vs 'x' plots would have settings from 'screen.rc', while the 'b' vs 'x' plot would
have settings from 'print.rc'.

    A dictionary can also be passed to the context manager:

    .. code::

        with mpl.rc_context(rc={'text.usetex': True}, fname='screen.rc'):
            plt.plot(x, a)

    The 'rc' dictionary takes precedence over the settings loaded from 'fname'. Passing a dictionary only
is also valid.
AFM (ADOBE FONT METRICS INTERFACE)

16.1 matplotlib.afm

This is a python interface to Adobe Font Metrics Files. Although a number of other python implementations exist, and may be more complete than this, it was decided not to go with them because they were either:

1. copyrighted or used a non-BSD compatible license
2. had too many dependencies and a free standing lib was needed
3. Did more than needed and it was easier to write afresh rather than figure out how to get just what was needed.

It is pretty easy to use, and requires only built-in python libs:

```python
>>> from matplotlib import rcParams
>>> import os.path
... afm_fname = os.path.join(rcParams['datapath'], ...
... 'fonts', 'afm', 'ptmr8a.afm')
... from matplotlib.afm import AFM
... with open(afm_fname) as fh:
...     afm = AFM(fh)
... afm.string_width_height('What the heck?')
(6220.0, 694)
>>> afm.get_fontname()
'Times-Roman'
>>> afm.get_kern_dist('A', 'f')
0
>>> afm.get_kern_dist('A', 'y')
-92.0
>>> afm.get_bbox_char('!')
[130, -9, 238, 676]
```

class matplotlib.afm.AFM(fh)

    Bases: object

    Parse the AFM file in file object fh

    family_name
get_angle()
    Return the fontangle as float

get_bbox_char(c, isord=False)

get_capheight()
    Return the cap height as float

get_familyname()
    Return the font family name, e.g., ‘Times’

get_fontname()
    Return the font name, e.g., ‘Times-Roman’

get_fullname()
    Return the font full name, e.g., ‘Times-Roman’

get_height_char(c, isord=False)
    Get the height of character c from the bounding box. This is the ink height (space is 0)

get_horizontal_stem_width()
    Return the standard horizontal stem width as float, or None if not specified in AFM file.

get_kern_dist(c1, c2)
    Return the kerning pair distance (possibly 0) for chars c1 and c2

get_kern_dist_from_name(name1, name2)
    Return the kerning pair distance (possibly 0) for chars name1 and name2

get_name_char(c, isord=False)
    Get the name of the character, i.e., ‘;’ is ‘semicolon’

get_str_bbox(s)
    Return the string bounding box

get_str_bbox_and_descent(s)
    Return the string bounding box

get_underline_thickness()
    Return the underline thickness as float

get_vertical_stem_width()
    Return the standard vertical stem width as float, or None if not specified in AFM file.

get_weight()
    Return the font weight, e.g., ‘Bold’ or ‘Roman’

get_width_char(c, isord=False)
    Get the width of the character from the character metric WX field

get_width_from_char_name(name)
    Get the width of the character from a type1 character name

get_xheight()
    Return the xheight as float
**string_width_height(s)**

Return the string width (including kerning) and string height as a \((w, h)\) tuple.

`matplotlib.afm.parse_afm(fh)`

Parse the Adobe Font Metics file in file handle \(fh\). Return value is a \((dhead, dcmetrics, dkernpairs, dcomposite)\) tuple where \(dhead\) is a `_parse_header()` dict, \(dcmetrics\) is a `_parse_composites()` dict, \(dkernpairs\) is a `_parse_kern_pairs()` dict (possibly \(\{\}\)), and \(dcomposite\) is a `_parse_composites()` dict (possibly \(\{\}\))
17.1 Animation

The easiest way to make a live animation in matplotlib is to use one of the `Animation` classes.

<table>
<thead>
<tr>
<th>Animation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>FuncAnimation</code></td>
<td>Makes an animation by repeatedly calling a function <code>func</code>.</td>
</tr>
<tr>
<td><code>ArtistAnimation</code></td>
<td>Animation using a fixed set of <code>Artist</code> objects.</td>
</tr>
</tbody>
</table>

17.1.1 `matplotlib.animation.FuncAnimation`

class `matplotlib.animation.FuncAnimation(fig, func, frames=None, init_func=None, fargs=None, save_count=None, **kwargs)`

Makes an animation by repeatedly calling a function `func`.

**Parameters**

- `fig` : `matplotlib.figure.Figure`
  
  The figure object that is used to get draw, resize, and any other needed events.

- `func` : `callable`
  
  The function to call at each frame. The first argument will be the next value in `frames`. Any additional positional arguments can be supplied via the `fargs` parameter.
The required signature is

```python
def func(fr: object, *fargs) -> iterable_of_artists:
```

**frames** : iterable, int, generator function, or None, optional

Source of data to pass `func` and each frame of the animation

- If an iterable, then simply use the values provided. If the iterable has a length, it will override the `save_count` kwarg.
- If an integer, equivalent to passing `range(frames)`
- If a generator function, then must have the signature

```python
def gen_function() -> obj:
```

In all of these cases, the values in `frames` is simply passed through to the user-supplied `func` and thus can be of any type.

- If `None`, then equivalent to passing `itertools.count`.

**init_func** : callable, optional

A function used to draw a clear frame. If not given, the results of drawing from the first item in the frames sequence will be used. This function will be called once before the first frame.

- If `blit=True`, `init_func` must return an iterable of artists to be re-drawn.

The required signature is

```python
def init_func() -> iterable_of_artists:
```

**fargs** : tuple or None, optional

Additional arguments to pass to each call to `func`

**save_count** : int, optional

The number of values from `frames` to cache.

**interval** : number, optional

Delay between frames in milliseconds. Defaults to 200.

**repeat_delay** : number, optional

- If the animation in repeated, adds a delay in milliseconds before repeating the animation. Defaults to `None`.

**repeat** : bool, optional

- Controls whether the animation should repeat when the sequence of frames is completed. Defaults to `True`.

**blit** : bool, optional

- Controls whether blitting is used to optimize drawing. Defaults to `False`. 
```
__init__(fig, func, frames=None, init_func=None, fargs=None, save_count=None, **kwargs)
```

### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong></td>
<td>(fig[, frames, init_func, ...])</td>
</tr>
<tr>
<td>new_frame_seq()</td>
<td></td>
</tr>
<tr>
<td>new_saved_frame_seq()</td>
<td></td>
</tr>
<tr>
<td>save(filename[, writer, fps, dpi, codec, ...])</td>
<td>Saves a movie file by drawing every frame.</td>
</tr>
<tr>
<td>to_html5_video()</td>
<td>Returns animation as an HTML5 video tag.</td>
</tr>
</tbody>
</table>

#### 17.1.2 matplotlib.animation.ArtistAnimation

**class** `matplotlib.animation.ArtistAnimation(fig, artists, *args, **kwargs)`  
Animation using a fixed set of Artist objects.

Before creating an instance, all plotting should have taken place and the relevant artists saved.

**Parameters**

- **fig** : `matplotlib.figure.Figure`  
The figure object that is used to get draw, resize, and any other needed events.

- **artists** : list  
  Each list entry a collection of artists that represent what needs to be enabled on each frame. These will be disabled for other frames.

- **interval** : number, optional  
  Delay between frames in milliseconds. Defaults to 200.

- **repeat_delay** : number, optional  
  If the animation in repeated, adds a delay in milliseconds before repeating the animation. Defaults to `None`.

- **repeat** : bool, optional  
  Controls whether the animation should repeat when the sequence of frames is completed. Defaults to `True`.

- **blit** : bool, optional  
  Controls whether blitting is used to optimize drawing. Defaults to `False`.

```
__init__(fig, artists, *args, **kwargs)
```

### Methods
In both cases it is critical to keep a reference to the instance object. The animation is advanced by a timer (typically from the host GUI framework) which the Animation object holds the only reference to. If you do not hold a reference to the Animation object, it (and hence the timers), will be garbage collected which will stop the animation.

To save an animation to disk use

```
Animation.save(filename[, writer, fps, dpi, codec, ...])
```

Saves a movie file by drawing every frame.

```
Animation.to_html5_video()
```

Returns animation as an HTML5 video tag.

### 17.1.3 matplotlib.animation.Animation.save

Animation.save(filename, writer=None, fps=None, dpi=None, codec=None, bitrate=None, extra_args=None, metadata=None, extra_anim=None, savefig_kwags=None)

Saves a movie file by drawing every frame.

**Parameters**

- **filename**: str
  - The output filename, e.g., mymovie.mp4

- **writer**: MovieWriter or str, optional
  - A MovieWriter instance to use or a key that identifies a class to use, such as ‘ffmpeg’ or ‘mencoder’. If None, defaults to rcParams['animation.writer']

- **fps**: number, optional
  - frames per second in the movie. Defaults to None, which will use the animation’s specified interval to set the frames per second.

- **dpi**: number, optional
  - Controls the dots per inch for the movie frames. This combined with the figure’s size in inches controls the size of the movie. If None, defaults to rcparam['savefig.dpi']

- **codec**: str, optional
  - The video codec to be used. Not all codecs are supported by a given MovieWriter. If None, default to rcParams['animation.codec']

- **bitrate**: number, optional
Specifies the number of bits used per second in the compressed movie, in kilobits per second. A higher number means a higher quality movie, but at the cost of increased file size. If None, defaults to rcParam['animation.bitrate']

**extra_args**: list, optional

List of extra string arguments to be passed to the underlying movie utility. If None, defaults to rcParams['animation.extra_args']

**metadata**: dict, optional

Dictionary of keys and values for metadata to include in the output file. Some keys that may be of use include: title, artist, genre, subject, copyright, src-form, comment.

**extra_anim**: list, optional

Additional `Animation` objects that should be included in the saved movie file. These need to be from the same `matplotlib.Figure` instance. Also, animation frames will just be simply combined, so there should be a 1:1 correspondence between the frames from the different animations.

**savefig_kwargs**: dict, optional

Is a dictionary containing keyword arguments to be passed on to the `savefig` command which is called repeatedly to save the individual frames.

**Notes**

fps, codec, bitrate, extra_args, metadata are used to construct a `MovieWriter` instance and can only be passed if `writer` is a string. If they are passed as non-None and `writer` is a `MovieWriter`, a `RuntimeError` will be raised.

### 17.1.4 `matplotlib.animation.Animation.to_html5_video`

`Animation.to_html5_video()`

Returns animation as an HTML5 video tag.

This saves the animation as an h264 video, encoded in base64 directly into the HTML5 video tag. This respects the rc parameters for the writer as well as the bitrate. This also makes use of the `interval` to control the speed, and uses the `repeat` parameter to decide whether to loop.

See `Helper Classes` below for details about what movie formats are supported.

### 17.1.5 `FuncAnimation`

The inner workings of `FuncAnimation` is more-or-less:
with details to handle ‘blitting’ (to dramatically improve the live performance), to be non-blocking, handle repeats, multiple animated axes, and easily save the animation to a movie file.

‘Blitting’ is an old technique in computer graphics. The general gist is to take an existing bit map (in our case a mostly rasterized figure) and then ‘blit’ one more artist on top. Thus, by managing a saved ‘clean’ bitmap, we can only re-draw the few artists that are changing at each frame and possibly save significant amounts of time. When using blitting (by passing blit=True) the core loop of FuncAnimation gets a bit more complicated

```python
for d in frames:
    artists = func(d, *fargs)
    fig.canvas.draw_idle()
    plt.pause(interval)
```

```
ax = fig.gca()

def update_blit(artists):
    fig.canvas.restore_region(bg_cache)
    for a in artists:
        a.axes.draw_artist(a)
        ax.figure.canvas.blit(ax.bbox)

artists = init_func()

for a in artists:
    a.set_animated(True)

fig.canvas.draw()
bg_cache = fig.canvas.copy_from_bbox(ax.bbox)

for f in frames:
    artists = func(f, *fargs)
    update_blit(artists)
    plt.pause(interval)
```

This is of course leaving out many details (such as updating the background when the figure is resized or fully re-drawn). However, this hopefully minimalist example gives a sense of how init_func and func are used inside of FuncAnimation and the theory of how ‘blitting’ works.

The expected signature on func and init_func is very simple to keep FuncAnimation out of your book keeping and plotting logic, but this means that the callable objects you pass in must know what artists they should be working on. There are several approaches to handling this, of varying complexity and encapsulation. The simplest approach, which works quite well in the case of a script, is to define the artist at a global scope and let Python sort things out. For example

```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation

fig, ax = plt.subplots()
xdata, ydata = [], []
```
The second method is to use `functools.partial` to 'bind' artists to function. A third method is to use closures to build up the required artists and functions. A fourth method is to create a class.

**Examples**

**animation example code: animate_decay.py**

[source code]

```
```
```
```python
def run(data):
    # update the data
    t, y = data
    xdata.append(t)
    ydata.append(y)
    xmin, xmax = ax.get_xlim()

    if t >= xmax:
        ax.set_xlim(xmin, 2*xmax)
        ax.figure.canvas.draw()
    line.set_data(xdata, ydata)
    return line,

ani = animation.FuncAnimation(fig, run, data_gen, blit=False, interval=10,
    repeat=False, init_func=init)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

animation example code: bayes_update.py

[source code]
class UpdateDist(object):
    def __init__(self, ax, prob=0.5):
        self.success = 0
        self.prob = prob
        self.line, = ax.plot([], [], 'k-')
        self.x = np.linspace(0, 1, 200)
        self.ax = ax

        # Set up plot parameters
        self.ax.set_xlim(0, 1)
        self.ax.set_ylim(0, 15)
        self.ax.grid(True)

        # This vertical line represents the theoretical value, to
        # which the plotted distribution should converge.
        self.ax.axvline(prob, linestyle='--', color='black')

    def init(self):
        self.success = 0
        self.line.set_data([], [])
        return self.line,

    def __call__(self, i):
        # This way the plot can continuously run and we just keep
        # watching new realizations of the process
        if i == 0:
            return self.init()

        # Choose success based on exceed a threshold with a uniform pick
        if np.random.rand(1,) < self.prob:
            self.success += 1
            y = ss.beta.pdf(self.x, self.success + 1, (i - self.success) + 1)
            self.line.set_data(self.x, y)
            return self.line,

fig, ax = plt.subplots()
ud = UpdateDist(ax, prob=0.7)
anim = FuncAnimation(fig, ud, frames=np.arange(100), init_func=ud.init,
                     interval=100, blit=True)
plt.show()
The double pendulum problem
===========================

This animation illustrates the double pendulum problem.

""
# Double pendulum formula translated from the C code at

from numpy import sin, cos
import numpy as np
import matplotlib.pyplot as plt
import scipy.integrate as integrate
import matplotlib.animation as animation

G = 9.8  # acceleration due to gravity, in m/s^2
L1 = 1.0  # length of pendulum 1 in m
L2 = 1.0  # length of pendulum 2 in m
M1 = 1.0  # mass of pendulum 1 in kg
M2 = 1.0  # mass of pendulum 2 in kg

def derivs(state, t):
    dydx = np.zeros_like(state)
    dydx[0] = state[1]
    del_ = state[2] - state[0]
    den1 = (M1 + M2)*L1 - M2*L1*cos(del_)*cos(del_)
    dydx[1] = (M2*L1*state[1]*state[1]*sin(del_)*cos(del_) +
               M2*G*sin(state[2])*cos(del_) +
               M2*L2*state[3]*state[3]*sin(del_) -
               (M1 + M2)*G*sin(state[0]))/den1
    den2 = (L2/L1)*den1
               (M1 + M2)*G*sin(state[0])*cos(del_) -
               (M1 + M2)*L1*state[1]*state[1]*sin(del_) -
               (M1 + M2)*G*sin(state[2]))/den2

    return dydx

# create a time array from 0..100 sampled at 0.05 second steps
dt = 0.05
t = np.arange(0.0, 20, dt)

# th1 and th2 are the initial angles (degrees)
# w10 and w20 are the initial angular velocities (degrees per second)
th1 = 120.0
w1 = 0.0
th2 = -10.0
w2 = 0.0

# initial state
class
state = np.radians([th1, w1, th2, w2])

# integrate your ODE using scipy.integrate.
y = integrate.odeint(derivs, state, t)

x1 = L1*sin(y[:,0])
y1 = -L1*cos(y[:,0])

x2 = L2*sin(y[:,2]) + x1
y2 = -L2*cos(y[:,2]) + y1

fig = plt.figure()
ax = fig.add_subplot(111, autoscale_on=False, xlim=(-2, 2), ylim=(-2, 2))
ax.grid()
line, = ax.plot([], [], 'o-', lw=2)
time_template = 'time = %.1fs'
time_text = ax.text(0.05, 0.9, '', transform=ax.transAxes)

def init():
    line.set_data([], [])
time_text.set_text('')
    return line, time_text

def animate(i):
    thisx = [0, x1[i], x2[i]]
    thisy = [0, y1[i], y2[i]]

    line.set_data(thisx, thisy)
time_text.set_text(time_template % (i*dt))
    return line, time_text

ani = animation.FuncAnimation(fig, animate, np.arange(1, len(y)), interval=25, blit=True, init_func=init)

# ani.save('double_pendulum.mp4', fps=15)
plt.show()
An animated image
===================

This example demonstrates how to animate an image.

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

fig = plt.figure()

def f(x, y):
    return np.sin(x) + np.cos(y)

x = np.linspace(0, 2 * np.pi, 120)
y = np.linspace(0, 2 * np.pi, 100).reshape(-1, 1)

im = plt.imshow(f(x, y), animated=True)

def updatefig(*args):
    global x, y
    x += np.pi / 15.
y += np.pi / 20.
im.set_array(f(x, y))

ani = animation.FuncAnimation(fig, updatefig, interval=50, blit=True)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

animation example code: histogram.py

[source code]

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import matplotlib.path as path

502 Chapter 17. animation module
```
```python
import matplotlib.animation as animation

fig, ax = plt.subplots()

# histogram our data with numpy
data = np.random.randn(1000)
n, bins = np.histogram(data, 100)

# get the corners of the rectangles for the histogram
left = np.array(bins[:-1])
right = np.array(bins[1:])
bottom = np.zeros(len(left))
top = bottom + n
nrects = len(left)

# here comes the tricky part -- we have to set up the vertex and path
# codes arrays using moveto, lineto and closepoly

# for each rect: 1 for the MOVETO, 3 for the LINETO, 1 for the
# CLOSEPOLY; the vert for the closepoly is ignored but we still need
# it to keep the codes aligned with the vertices
nverts = nrects*(1 + 3 + 1)
verts = np.zeros((nverts, 2))
codes = np.ones(nverts, int) * path.Path.LINETO
verts[0::5, 0] = left
verts[0::5, 1] = bottom
verts[1::5, 0] = left
verts[1::5, 1] = top
verts[2::5, 0] = right
verts[2::5, 1] = top
verts[3::5, 0] = right
verts[3::5, 1] = bottom

barpath = path.Path(verts, codes)
patch = patches.PathPatch(
    barpath, facecolor='green', edgecolor='yellow', alpha=0.5)
ax.add_patch(patch)

ax.set_xlim(left[0], right[-1])
ax.set_ylim(bottom.min(), top.max())

def animate(i):
    # simulate new data coming in
    data = np.random.randn(1000)
n, bins = np.histogram(data, 100)
top = bottom + n
verts[1::5, 1] = top
verts[2::5, 1] = top
return [patch,]
```

17.1. Animation
Matplotlib, Release 2.0.2

```python
ani = animation.FuncAnimation(fig, animate, 100, repeat=False, blit=True)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

**animation example code: rain.py**

[source code]

```python
""
===============
Rain simulation
===============

Simulates rain drops on a surface by animating the scale and opacity of 50 scatter points.

Author: Nicolas P. Rougier
""
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation

# Create new Figure and an Axes which fills it.
fig = plt.figure(figsize=(7, 7))
ax = fig.add_axes([0, 0, 1, 1], frameon=False)
ax.set_xlim(0, 1), ax.set_xticks([])
ax.set_ylim(0, 1), ax.set_yticks([])

# Create rain data
n_drops = 50
rain_drops = np.zeros(n_drops, dtype=[('position', float, 2),
                                     ('size', float, 1),
                                     ('growth', float, 1),
                                     ('color', float, 4)])

# Initialize the raindrops in random positions and with
# random growth rates.
rain_drops['position'] = np.random.uniform(0, 1, (n_drops, 2))
rain_drops['growth'] = np.random.uniform(50, 200, n_drops)

# Construct the scatter which we will update during animation
# as the raindrops develop.
scat = ax.scatter(rain_drops['position'][:, 0], rain_drops['position'][:, 1],
                  s=rain_drops['size'], lw=0.5, edgecolors=rain_drops['color'],
                  facecolors='none')

def update(frame_number):
    # Get an index which we can use to re-spawn the oldest raindrop.
```
current_index = frame_number % n_drops

# Make all colors more transparent as time progresses.
rain_drops['color'][current_index] -= 1.0/len(rain_drops)
np.clip(rain_drops['color'][:, 3], 0, 1)

# Make all circles bigger.
rain_drops['size'] += rain_drops['growth']

# Pick a new position for oldest rain drop, resetting its size,
# color and growth factor.
rain_drops['position'][current_index] = np.random.uniform(0, 1, 2)
rain_drops['size'][current_index] = 5
rain_drops['color'][current_index] = (0, 0, 0, 1)
rain_drops['growth'][current_index] = np.random.uniform(50, 200)

# Update the scatter collection, with the new colors, sizes and positions.
scat.set_edgecolors(rain_drops['color'])
scat.set_sizes(rain_drops['size'])
scat.set_offsets(rain_drops['position'])

# Construct the animation, using the update function as the animation
# director.
animation = FuncAnimation(fig, update, interval=10)
plt.show()
```python
def update(data):
    line.set_ydata(data)
    return line,

def data_gen():
    while True:
        yield np.random.rand(10)

ani = animation.FuncAnimation(fig, update, data_gen, interval=100)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

**animation example code: simple_3danim.py**

[source code]

```python
# """
# 3D animation
# """

A simple example of an animated plot... In 3D!

```

```python
import numpy as np
import matplotlib.pyplot as plt
import mpl_toolkits.mplot3d.axes3d as p3
import matplotlib.animation as animation

def Gen_RandLine(length, dims=2):
    # Create a line using a random walk algorithm

    length is the number of points for the line.
dims is the number of dimensions the line has.
    """
    lineData = np.empty((dims, length))
    lineData[:, 0] = np.random.rand(dims)
    for index in range(1, length):
        # scaling the random numbers by 0.1 so
        # movement is small compared to position.
        # subtraction by 0.5 is to change the range to [-0.5, 0.5]
        # to allow a line to move backwards.
        step = ((np.random.rand(dims) - 0.5) * 0.1)
        lineData[:, index] = lineData[:, index - 1] + step

    return lineData
```

506 Chapter 17. animation module
```python
def update_lines(num, dataLines, lines):
    for line, data in zip(lines, dataLines):
        # NOTE: there is no .set_data() for 3 dim data...
        line.set_data(data[0:2, :num])
        line.set_3d_properties(data[2, :num])
    return lines

# Attaching 3D axis to the figure
fig = plt.figure()
ax = p3.Axes3D(fig)

# Fifty lines of random 3-D lines
data = [Gen_RandLine(25, 3) for index in range(50)]

# Creating fifty line objects.
# NOTE: Can't pass empty arrays into 3d version of plot()
lines = [ax.plot(dat[0, 0:1], dat[1, 0:1], dat[2, 0:1])[0] for dat in data]

# Setting the axes properties
ax.set_xlim3d([0.0, 1.0])
ax.set_xlabel('X')
ax.set_ylim3d([0.0, 1.0])
ax.set_ylabel('Y')
ax.set_zlim3d([0.0, 1.0])
ax.set_zlabel('Z')
ax.set_title('3D Test')

# Creating the Animation object
line_ani = animation.FuncAnimation(fig, update_lines, 25, fargs=(data, lines),
    interval=50, blit=False)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

**animation example code: simple_anim.py**

[source code]

```
""
A simple example of an animated plot
""
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

fig, ax = plt.subplots()
```
```python
x = np.arange(0, 2*np.pi, 0.01)
line, = ax.plot(x, np.sin(x))

def animate(i):
    line.set_ydata(np.sin(x + i/10.0))  # update the data
    return line,

# Init only required for blitting to give a clean slate.
def init():
    line.set_ydata(np.ma.array(x, mask=True))
    return line,

ani = animation.FuncAnimation(fig, animate, np.arange(1, 200), init_func=init,
                               interval=25, blit=True)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

**animation example code: strip_chart_demo.py**

[source code]

```python
# Oscilloscope

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

class Scope(object):
    def __init__(self, ax, maxt=2, dt=0.02):
        self.ax = ax
        self.dt = dt
        self.maxt = maxt
        self.tdata = [0]
        self.ydata = [0]
        self.line = Line2D(self.tdata, self.ydata)
        self.ax.add_line(self.line)
        self.ax.set_ylim(-.1, 1.1)
        self.ax.set_xlim(0, self.maxt)

    def update(self, y):
        lastt = self.tdata[-1]
```
if lastt > self.tdata[0] + self.maxt:  # reset the arrays
    self.tdata = [self.tdata[-1]]
    self.ydata = [self.ydata[-1]]
    self.ax.set_xlim(self.tdata[0], self.tdata[0] + self.maxt)
    self.ax.figure.canvas.draw()

    t = self.tdata[-1] + self.dt
    self.tdata.append(t)
    self.ydata.append(y)
    self.line.set_data(self.tdata, self.ydata)
    return self.line,

def emitter(p=0.03):
    'return a random value with probability p, else 0'
    while True:
        v = np.random.rand(1)
        if v > p:
            yield 0.
        else:
            yield np.random.rand(1)

fig, ax = plt.subplots()
scope = Scope(ax)

# pass a generator in "emitter" to produce data for the update func
ani = animation.FuncAnimation(fig, scope.update, emitter, interval=10,
                               blit=True)

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

animation example code: unchained.py

[source code]
# Create new Figure with black background
fig = plt.figure(figsize=(8, 8), facecolor='black')

# Add a subplot with no frame
ax = plt.subplot(111, frameon=False)

# Generate random data
data = np.random.uniform(0, 1, (64, 75))
X = np.linspace(-1, 1, data.shape[-1])
G = 1.5 * np.exp(-4 * X * X)

# Generate line plots
lines = []
for i in range(len(data)):
    # Small reduction of the X extents to get a cheap perspective effect
    xscale = 1 - i / 200.
    # Same for linewidth (thicker strokes on bottom)
    lw = 1.5 - i / 100.0
    line, = ax.plot(xscale * X, i + G * data[i], color="w", lw=lw)
    lines.append(line)

# Set y limit (or first line is cropped because of thickness)
ax.set_ylim(-1, 70)

# No ticks
ax.set_xticks([])
ax.set_yticks([])

# 2 part titles to get different font weights
ax.text(0.5, 1.0, "MATPLOTLIB ", transform=ax.transAxes,
        ha="right", va="bottom", color="w",
        family="sans-serif", fontweight="light", fontsize=16)
ax.text(0.5, 1.0, "UNCHAINED", transform=ax.transAxes,
        ha="left", va="bottom", color="w",
        family="sans-serif", fontweight="bold", fontsize=16)

def update(*args):
    # Shift all data to the right
    data[:, 1:] = data[:, :-1]

    # Fill-in new values
    data[:, 0] = np.random.uniform(0, 1, len(data))

    # Update data
    for i in range(len(data)):
        lines[i].set_ydata(i + G * data[i])

    # Return modified artists
    return lines

# Construct the animation, using the update function as the animation
# director.
anim = animation.FuncAnimation(fig, update, interval=10)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

17.1.6 ArtistAnimation

Examples

animation example code: basic_example.py

[source code]

```python
# Simple animation examples

This example contains two animations. The first is a random walk plot. The second is an image animation.

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

def update_line(num, data, line):
    line.set_data(data[..., :num])
    return line,

fig1 = plt.figure()
data = np.random.rand(2, 25)
l, = plt.plot([], [], 'r-')
plt.xlim(0, 1)
plt.ylim(0, 1)
plt.xlabel('x')
plt.title('test')
line_ani = animation.FuncAnimation(fig1, update_line, 25, fargs=(data, l),
interval=50, blit=True)

# To save the animation, use the command: line_ani.save('lines.mp4')

fig2 = plt.figure()
x = np.arange(-9, 10)
y = np.arange(-9, 10).reshape(-1, 1)
base = np.hypot(x, y)
ims = []
```
Matplotlib, Release 2.0.2

```python
for add in np.arange(15):
    ims.append((plt.pcolor(x, y, base + add, norm=plt.Normalize(0, 30)),))

im_ani = animation.ArtistAnimation(fig2, ims, interval=50, repeat_delay=3000, blit=True)
# To save this second animation with some metadata, use the following command:
# im_ani.save('im.mp4', metadata={'artist':'Guido'})
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

**animation example code: basic_example_writer.py**

```python
# -*- noplot -*-
import numpy as np
import matplotlib
matplotlib.use("Agg")
import matplotlib.pyplot as plt
import matplotlib.animation as animation

def update_line(num, data, line):
    line.set_data(data[..., :num])
    return line,

# Set up formatting for the movie files
Writer = animation.writers['ffmpeg']
writer = Writer(fps=15, metadata=dict(artist='Me'), bitrate=1800)

fig1 = plt.figure()

data = np.random.rand(2, 25)
l, = plt.plot([], [], 'r-')
plt.xlim(0, 1)
plt.ylim(0, 1)
plt.xlabel('x')
plt.title('test')
```

512 Chapter 17. animation module
```python
line_ani = animation.FuncAnimation(fig1, update_line, 25, fargs=(data, l),
                                   interval=50, blit=True)
line_ani.save('lines.mp4', writer=writer)

fig2 = plt.figure()
x = np.arange(-9, 10)
y = np.arange(-9, 10).reshape(-1, 1)
base = np.hypot(x, y)
ims = []
for add in np.arange(15):
    ims.append((plt.pcolor(x, y, base + add, norm=plt.Normalize(0, 30)),))
im_ani = animation.ArtistAnimation(fig2, ims, interval=50, repeat_delay=3000,
                                   blit=True)
im_ani.save('im.mp4', writer=writer)
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

**animation example code:** dynamic_image2.py

[source code]
ims.append([im])

ani = animation.ArtistAnimation(fig, ims, interval=50, blit=True,
                       repeat_delay=1000)

# ani.save('dynamic_images.mp4')
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

## 17.2 Writer Classes

The provided writers fall into two broad categories: pipe-based and file-based. The pipe-based writers stream the captured frames over a pipe to an external process. The pipe-based variants tend to be more performant, but may not work on all systems.

<table>
<thead>
<tr>
<th>Writer</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>FFMpegWriter</code></td>
<td>Pipe-based ffmpeg writer.</td>
</tr>
<tr>
<td><code>ImageMagickFileWriter</code></td>
<td>File-based animated gif writer.</td>
</tr>
<tr>
<td><code>AVConvWriter</code></td>
<td>Pipe-based avconv writer.</td>
</tr>
</tbody>
</table>

### 17.2.1 matplotlib.animation.FFMpegWriter

```python
class matplotlib.animation.FFMpegWriter(fps=5, codec=None, bitrate=None, extra_args=None, metadata=None)
```

Pipe-based ffmpeg writer.

Frames are streamed directly to ffmpeg via a pipe and written in a single pass.

**Parameters**

- **fps**: int
  
  Framerate for movie.

- **codec**: string or None, optional

  The codec to use. If None (the default) the setting in the rcParam animation.codec is used.

- **bitrate**: int or None, optional

  The bitrate for the saved movie file, which is one way to control the output file size and quality. The default value is None, which uses the value stored in the rcParam animation.bitrate. A value of -1 implies that the bitrate should be determined automatically by the underlying utility.

- **extra_args**: list of strings or None

  A list of extra string arguments to be passed to the underlying movie utility. The default is None, which passes the additional arguments in the ‘animation.extra_args’ rcParam.
metadata: dict of string:string or None

A dictionary of keys and values for metadata to include in the output file. Some keys that may be of use include: title, artist, genre, subject, copyright, srcform, comment.

__init__(fps=5, codec=None, bitrate=None, extra_args=None, metadata=None)
MovieWriter

Parameters fps: int

Framerate for movie.

codec: string or None, optional

The codec to use. If None (the default) the setting in the rcParam animation.codec is used.

bitrate: int or None, optional

The bitrate for the saved movie file, which is one way to control the output file size and quality. The default value is None, which uses the value stored in the rcParam animation.bitrate. A value of -1 implies that the bitrate should be determined automatically by the underlying utility.

extra_args: list of strings or None

A list of extra string arguments to be passed to the underlying movie utility. The default is None, which passes the additional arguments in the ‘animation.extra_args’ rcParam.

metadata: dict of string:string or None

A dictionary of keys and values for metadata to include in the output file. Some keys that may be of use include: title, artist, genre, subject, copyright, srcform, comment.

Methods

<table>
<thead>
<tr>
<th><strong>init</strong>(fps, codec, bitrate, extra_args, ...)</th>
<th>MovieWriter</th>
</tr>
</thead>
<tbody>
<tr>
<td>bin_path()</td>
<td>Returns the binary path to the commandline tool used by a specific subclass.</td>
</tr>
<tr>
<td>cleanup()</td>
<td>Clean-up and collect the process used to write the movie file.</td>
</tr>
<tr>
<td>finish()</td>
<td>Finish any processing for writing the movie.</td>
</tr>
<tr>
<td>grab_frame(**savefig_kwvars)</td>
<td>Grab the image information from the figure and save as a movie frame.</td>
</tr>
<tr>
<td>isAvailable()</td>
<td>Check to see if a MovieWriter subclass is actually available by running the commandline tool.</td>
</tr>
<tr>
<td>saving(*args, **kw)</td>
<td>Context manager to facilitate writing the movie file.</td>
</tr>
</tbody>
</table>

Continued on next page
setup(fig, outfile, dpi[, frame_prefix, ...]) Perform setup for writing the movie file.

Attributes

args_key
delay
eexec_key
frame_size
output_args

17.2.2 matplotlib.animation.ImageMagickFileWriter

class matplotlib.animation.ImageMagickFileWriter(*args, **kwargs)

File-based animated gif writer.

Frames are written to temporary files on disk and then stitched together at the end.

__init__(*args, **kwargs)

Methods

__init__(*args, **kwargs)

bin_path() Returns the binary path to the commandline tool used by a specific subclass.

cleanup()

finish()

grab_frame(**savefig_kwargs) Grab the image information from the figure and save as a movie frame.

isAvailable() Check to see if a ImageMagickWriter is actually available.

saving(*args, **kw) Context manager to facilitate writing the movie file.

setup(fig, outfile, dpi[, frame_prefix, ...]) Perform setup for writing the movie file.

Attributes

args_key
delay
eexec_key

Continued on next page
Table 17.9 – continued from previous page

| frame_format | Format (png, jpeg, etc.) to use for saving the frames, which can be decided by the individual subclasses. |
| frame_size   | A tuple (width, height) in pixels of a movie frame. |
| output_args  | supported_formats |

17.2.3 matplotlib.animation.AVConvWriter

class matplotlib.animation.AVConvWriter(fps=5, codec=None, bitrate=None, extra_args=None, metadata=None)

Pipe-based avconv writer.

Frames are streamed directly to avconv via a pipe and written in a single pass.

**Parameters fps: int**

Frame rate for movie.

**codec: string or None, optional**

The codec to use. If None (the default) the setting in the rcParam animation.codec is used.

**bitrate: int or None, optional**

The bitrate for the saved movie file, which is one way to control the output file size and quality. The default value is None, which uses the value stored in the rcParam animation.bitrate. A value of -1 implies that the bitrate should be determined automatically by the underlying utility.

**extra_args: list of strings or None**

A list of extra string arguments to be passed to the underlying movie utility. The default is None, which passes the additional arguments in the ‘animation.extra_args’ rcParam.

**metadata: dict of string: string or None**

A dictionary of keys and values for metadata to include in the output file. Some keys that may be of use include: title, artist, genre, subject, copyright, srcform, comment.

__init__(fps=5, codec=None, bitrate=None, extra_args=None, metadata=None)

MovieWriter

**Parameters fps: int**

Frame rate for movie.

**codec: string or None, optional**

The codec to use. If None (the default) the setting in the rcParam animation.codec is used.
**bitrate**: int or None, optional

The bitrate for the saved movie file, which is one way to control the output file size and quality. The default value is None, which uses the value stored in the rcParam `animation.bitrate`. A value of -1 implies that the bitrate should be determined automatically by the underlying utility.

**extra_args**: list of strings or None

A list of extra string arguments to be passed to the underlying movie utility. The default is None, which passes the additional arguments in the `animation.extra_args` rcParam.

**metadata**: dict of string:string or None

A dictionary of keys and values for metadata to include in the output file. Some keys that may be of use include: title, artist, genre, subject, copyright, srcform, comment.

### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__init__</code>([fps, codec, bitrate, extra_args, ...])</td>
<td>MovieWriter</td>
</tr>
<tr>
<td><code>bin_path()</code></td>
<td>Returns the binary path to the commandline tool used by a specific subclass.</td>
</tr>
<tr>
<td><code>cleanup()</code></td>
<td>Clean-up and collect the process used to write the movie file.</td>
</tr>
<tr>
<td><code>finish()</code></td>
<td>Finish any processing for writing the movie.</td>
</tr>
<tr>
<td><code>grab_frame(**savefig_kwargs)</code></td>
<td>Grab the image information from the figure and save as a movie frame.</td>
</tr>
<tr>
<td><code>isAvailable()</code></td>
<td>Check to see if a MovieWriter subclass is actually available by running the commandline tool.</td>
</tr>
<tr>
<td><code>saving(*args, **kw)</code></td>
<td>Context manager to facilitate writing the movie file.</td>
</tr>
<tr>
<td><code>setup(fig, outfile, dpi)</code></td>
<td>Perform setup for writing the movie file.</td>
</tr>
</tbody>
</table>

### Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>args_key</code></td>
<td></td>
</tr>
<tr>
<td><code>exec_key</code></td>
<td></td>
</tr>
<tr>
<td><code>frame_size</code></td>
<td>A tuple (width,height) in pixels of a movie frame.</td>
</tr>
<tr>
<td><code>output_args</code></td>
<td></td>
</tr>
</tbody>
</table>

Alternatively the file-based writers save temporary files for each frame which are stitched into a single file at the end. Although slower, these writers can be easier to debug.

**FFMpegFileWriter**

File-based ffmpeg writer.

Continued on next page
Matplotlib, Release 2.0.2

Table 17.12 – continued from previous page

<table>
<thead>
<tr>
<th>Writer Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ImageMagickWriter</td>
<td>Pipe-based animated gif.</td>
</tr>
<tr>
<td>AVConvFileWriter</td>
<td>File-based avconv writer.</td>
</tr>
</tbody>
</table>

### 17.2.4 matplotlib.animation.FFMpegFileWriter

**Class**

```python
class matplotlib.animation.FFMpegFileWriter(*args, **kwargs)
```

File-based ffmpeg writer.

Frames are written to temporary files on disk and then stitched together at the end.

```python
__init__(*args, **kwargs)
```

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__init__(*args, **kwargs)</code></td>
<td>Returns the binary path to the commandline tool used by a specific subclass.</td>
</tr>
<tr>
<td><code>bin_path()</code></td>
<td></td>
</tr>
<tr>
<td><code>cleanup()</code></td>
<td></td>
</tr>
<tr>
<td><code>finish()</code></td>
<td></td>
</tr>
<tr>
<td><code>grab_frame(**savefig_kwargs)</code></td>
<td>Grab the image information from the figure and save as a movie frame.</td>
</tr>
<tr>
<td><code>isAvailable()</code></td>
<td>Check to see if a MovieWriter subclass is actually available by running the commandline tool.</td>
</tr>
<tr>
<td><code>saving(*args, **kw)</code></td>
<td>Context manager to facilitate writing the movie file.</td>
</tr>
<tr>
<td><code>setup(fig, outfile, dpi[, frame_prefix, ...])</code></td>
<td>Perform setup for writing the movie file.</td>
</tr>
</tbody>
</table>

**Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>args_key</code></td>
<td></td>
</tr>
<tr>
<td><code>exec_key</code></td>
<td></td>
</tr>
<tr>
<td><code>frame_format</code></td>
<td>Format (png, jpeg, etc.) to use for saving the frames, which can be decided by the individual subclasses.</td>
</tr>
<tr>
<td><code>frame_size</code></td>
<td>A tuple (width, height) in pixels of a movie frame.</td>
</tr>
<tr>
<td><code>output_args</code></td>
<td></td>
</tr>
<tr>
<td><code>supported_formats</code></td>
<td></td>
</tr>
</tbody>
</table>
17.2.5 matplotlib.animation.ImageMagickWriter

class matplotlib.animation.ImageMagickWriter(fps=5, codec=None, bitrate=None, extra_args=None, metadata=None)

Pipe-based animated gif.

Frames are streamed directly to ImageMagick via a pipe and written in a single pass.

**Parameters**

**fps: int**

Framerate for movie.

**codec: string or None, optional**

The codec to use. If None (the default) the setting in the rcParam animation.codec is used.

**bitrate: int or None, optional**

The bitrate for the saved movie file, which is one way to control the output file size and quality. The default value is None, which uses the value stored in the rcParam animation.bitrate. A value of -1 implies that the bitrate should be determined automatically by the underlying utility.

**extra_args: list of strings or None**

A list of extra string arguments to be passed to the underlying movie utility. The default is None, which passes the additional arguments in the ‘animation.extra_args’ rcParam.

**metadata: dict of string:string or None**

A dictionary of keys and values for metadata to include in the output file. Some keys that may be of use include: title, artist, genre, subject, copyright, srcform, comment.

**__init__**(fps=5, codec=None, bitrate=None, extra_args=None, metadata=None)

MovieWriter

**Parameters**

**fps: int**

Framerate for movie.

**codec: string or None, optional**

The codec to use. If None (the default) the setting in the rcParam animation.codec is used.

**bitrate: int or None, optional**

The bitrate for the saved movie file, which is one way to control the output file size and quality. The default value is None, which uses the value stored in the rcParam animation.bitrate. A value of -1 implies that the bitrate should be determined automatically by the underlying utility.

**extra_args: list of strings or None**
A list of extra string arguments to be passed to the underlying movie utility. The default is None, which passes the additional arguments in the ‘animation.extra_args’ rcParam.

**metadata: dict of string:string or None**

A dictionary of keys and values for metadata to include in the output file. Some keys that may be of use include: title, artist, genre, subject, copyright, srcform, comment.

### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong><strong>init</strong></strong>([fps, codec, bitrate, extra_args, ...])</td>
<td>MovieWriter</td>
</tr>
<tr>
<td>bin_path()</td>
<td>Returns the binary path to the commandline tool used by a specific subclass.</td>
</tr>
<tr>
<td>cleanup()</td>
<td>Clean-up and collect the process used to write the movie file.</td>
</tr>
<tr>
<td>finish()</td>
<td>Finish any processing for writing the movie.</td>
</tr>
<tr>
<td>grab_frame(**savefig_kwargs)</td>
<td>Grab the image information from the figure and save as a movie frame.</td>
</tr>
<tr>
<td>isAvailable()</td>
<td>Check to see if a ImageMagickWriter is actually available.</td>
</tr>
<tr>
<td>saving(*args, **kw)</td>
<td>Context manager to facilitate writing the movie file.</td>
</tr>
<tr>
<td>setup(fig, outfile, dpi)</td>
<td>Perform setup for writing the movie file.</td>
</tr>
</tbody>
</table>

### Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>args_key</td>
<td></td>
</tr>
<tr>
<td>delay</td>
<td></td>
</tr>
<tr>
<td>exec_key</td>
<td></td>
</tr>
<tr>
<td>frame_size</td>
<td>A tuple (width, height) in pixels of a movie frame.</td>
</tr>
<tr>
<td>output_args</td>
<td></td>
</tr>
</tbody>
</table>

### 17.2.6 matplotlib.animation.AVConvFileWriter

class matplotlib.animation.AVConvFileWriter(*args, **kwargs)

File-based avconv writer.

Frames are written to temporary files on disk and then stitched together at the end.

**__init__**(args, **kwargs)
Methods

__init__(*args, **kwargs)

bin_path() Returns the binary path to the commandline tool used by a specific subclass.

cleanup()

finish()

grab_frame(**savefig_kwargs) Grab the image information from the figure and save as a movie frame.

isAvailable() Check to see if a MovieWriter subclass is actually available by running the commandline tool.

saving(*args, **kw) Context manager to facilitate writing the movie file.

setup(fig, outfile, dpi[, frame_prefix, ...]) Perform setup for writing the movie file.

Attributes

args_key

exec_key

frame_format Format (png, jpeg, etc.) to use for saving the frames, which can be decided by the individual subclasses.

frame_size A tuple (width,height) in pixels of a movie frame.

output_args

supported_formats

Fundamentally, a MovieWriter provides a way to grab sequential frames from the same underlying Figure object. The base class MovieWriter implements 3 methods and a context manager. The only difference between the pipe-based and file-based writers is in the arguments to their respective setup methods.

MovieWriter.setup Perform setup for writing the movie file.

FileMovieWriter.setup Perform setup for writing the movie file.

MovieWriter.grab_frame Grab the image information from the figure and save as a movie frame.

MovieWriter.finish Finish any processing for writing the movie.

MovieWriter.saving Context manager to facilitate writing the movie file.

17.2.7 matplotlib.animation.MovieWriter.setup

MovieWriter.setup(fig, outfile, dpi)

Perform setup for writing the movie file.

Parameters:

fig : matplotlib.Figure instance

The figure object that contains the information for frames.
outfile : string
    The filename of the resulting movie file

dpi : int
    The DPI (or resolution) for the file. This controls the size in pixels of the resulting movie file.

17.2.8 matplotlib.animation.FileMovieWriter.setup

FileMovieWriter.setup(fig, outfile, dpi, frame_prefix=''_tmp'', clear_temp=True)
    Perform setup for writing the movie file.

    Parameters
    fig : matplotlib.figure.Figure
        The figure to grab the rendered frames from.

    outfile : str
        The filename of the resulting movie file.

    dpi : number
        The dpi of the output file. This, with the figure size, controls the size in pixels of the resulting movie file.

    frame_prefix : str, optional
        The filename prefix to use for temporary files. Defaults to ‘'_tmp''.

    clear_temp : bool, optional
        If the temporary files should be deleted after stitching the final result. Setting this to False can be useful for debugging. Defaults to True.

17.2.9 matplotlib.animation.MovieWriter.grab_frame

MovieWriter.grab_frame(**savefig_kwargs)
    Grab the image information from the figure and save as a movie frame.

    All keyword arguments in savefig_kwargs are passed on to the ‘savefig’ command that saves the figure.

17.2.10 matplotlib.animation.MovieWriter.finish

MovieWriter.finish()
    Finish any processing for writing the movie.
17.2.11 matplotlib.animation.MovieWriter.saving

MovieWriter.saving(*args, **kw)

Context manager to facilitate writing the movie file.

*args, **kw are any parameters that should be passed to setup.

The setup() method is used to prepare the writer (possibly opening a pipe), successive calls to grab_frame() capture a single frame at a time and finish() finalizes the movie and writes the output file to disk. For example

```python
moviewriter = MovieWriter(...)
moviewriter.setup(fig=fig, 'my_movie.ext', dpi=100)
for j in range(n):
    update_figure(n)
    moviewriter.grab_frame()
moviewriter.finish()
```

If using the writer classes directly (not through Animation.save), it is strongly encouraged to use the saving context manager

```python
with moviewriter.saving(fig, 'myfile.mp4', dpi=100):
    for j in range(n):
        update_figure(n)
        moviewriter.grab_frame()
```

to ensures that setup and cleanup are performed as necessary.

*animation example code: moviewriter.py*

17.3 Helper Classes

17.3.1 Animation Base Classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Animation</td>
<td>This class wraps the creation of an animation using matplotlib.</td>
</tr>
<tr>
<td>TimedAnimation</td>
<td>Animation subclass for time-based animation.</td>
</tr>
</tbody>
</table>

matplotlib.animation.Animation

class matplotlib.animation.Animation(fig, event_source=None, blit=False)

This class wraps the creation of an animation using matplotlib.

It is only a base class which should be subclassed to provide needed behavior.

This class is not typically used directly.

Parameters

- **fig** : matplotlib.figure.Figure

  The figure object that is used to get draw, resize, and any other needed events.
**event_source** : object, optional

A class that can run a callback when desired events are generated, as well as be stopped and started.

Examples include timers (see `TimedAnimation`) and file system notifications.

**blit** : bool, optional

controls whether blitting is used to optimize drawing. Defaults to `False`.

**See also:**

`FuncAnimation`, `ArtistAnimation`

```python
__init__(fig, event_source=None, blit=False)
```

### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong>(fig[, event_source, blit])</td>
<td></td>
</tr>
<tr>
<td>new_frame_seq()</td>
<td>Creates a new sequence of frame information.</td>
</tr>
<tr>
<td>new_saved_frame_seq()</td>
<td>Creates a new sequence of saved/cached frame information.</td>
</tr>
<tr>
<td>save(filename[, writer, fps, dpi, codec, ...])</td>
<td>Saves a movie file by drawing every frame.</td>
</tr>
<tr>
<td>to_html5_video()</td>
<td>Returns animation as an HTML5 video tag.</td>
</tr>
</tbody>
</table>

### matplotlib.animation.TimedAnimation

**class matplotlib.animation.TimedAnimation(fig, interval=200, repeat_delay=None, repeat=True, event_source=None, *args, **kwargs)**

*Animation* subclass for time-based animation.

A new frame is drawn every `interval` milliseconds.

**Parameters**

- **fig** : `matplotlib.figure.Figure`
  
  The figure object that is used to get draw, resize, and any other needed events.

- **interval** : number, optional
  
  Delay between frames in milliseconds. Defaults to 200.

- **repeat_delay** : number, optional
  
  If the animation in repeated, adds a delay in milliseconds before repeating the animation. Defaults to `None`.

- **repeat** : bool, optional
  
  Controls whether the animation should repeat when the sequence of frames is completed. Defaults to `True`.

17.3. Helper Classes 525
**blit** : bool, optional
Controls whether blitting is used to optimize drawing. Defaults to False.

```python
__init__(fig, interval=200, repeat_delay=None, repeat=True, event_source=None, **kwargs)
```

## Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__init__(fig[, interval, repeat_delay, ...])</code></td>
<td>Creates a new sequence of frame information.</td>
</tr>
<tr>
<td><code>new_frame_seq()</code></td>
<td>Creates a new sequence of saved/cached frame information.</td>
</tr>
<tr>
<td><code>save(filename[, writer, fps, dpi, codec, ...])</code></td>
<td>Saves a movie file by drawing every frame.</td>
</tr>
<tr>
<td><code>to_html5_video()</code></td>
<td>Returns animation as an HTML5 video tag.</td>
</tr>
</tbody>
</table>

### 17.3.2 Custom Animation classes

*animation example code: subplots.py*

### 17.3.3 Writer Registry

A module-level registry is provided to map between the name of the writer and the class to allow a string to be passed to `Animation.save` instead of a writer instance.

```python
MovieWriterRegistry
```

#### `matplotlib.animation.MovieWriterRegistry`

Class

- Registry of available writer classes by human readable name.

```python
__init__()
```

## Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__init__()</code></td>
<td></td>
</tr>
<tr>
<td><code>ensure_not_dirty()</code></td>
<td>If dirty, reasks the writers if they are available</td>
</tr>
<tr>
<td><code>is_available(name)</code></td>
<td>Check if given writer is available by name.</td>
</tr>
<tr>
<td><code>list()</code></td>
<td>Get a list of available MovieWriters.</td>
</tr>
</tbody>
</table>

Continued on next page
### 17.3.4 Writer Base Classes

To reduce code duplication base classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MovieWriter</td>
<td>Base class for writing movies.</td>
</tr>
<tr>
<td>FileMovieWriter</td>
<td>Sub-class of MovieWriter used for writing to individual files and stitching at the end.</td>
</tr>
</tbody>
</table>

**matplotlib.animation.MovieWriter**

```python
class matplotlib.animation.MovieWriter(fps=5, codec=None, bitrate=None, extra_args=None, metadata=None)
```

Base class for writing movies. This class is set up to provide for writing movie frame data to a pipe. See examples for how to use these classes.

**Attributes**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>frame_format</td>
<td>(str) The format used in writing frame data, defaults to ‘rgba’</td>
</tr>
<tr>
<td>fig</td>
<td>(Figure) The figure to capture data from. This must be provided by the sub-classes.</td>
</tr>
</tbody>
</table>

**Parameters**

- **fps**: int
  - Frame rate for movie.

- **codec**: string or None, optional
  - The codec to use. If None (the default) the setting in the rcParam animation.codec is used.

- **bitrate**: int or None, optional
  - The bitrate for the saved movie file, which is one way to control the output file size and quality. The default value is None, which uses the value stored in the rcParam animation.bitrate. A value of -1 implies that the bitrate should be determined automatically by the underlying utility.

- **extra_args**: list of strings or None
  - A list of extra string arguments to be passed to the underlying movie utility. The default is None, which passes the additional arguments in the ‘animation.extra_args’ rcParam.
metadata: dict of string:string or None

A dictionary of keys and values for metadata to include in the output file. Some keys that may be of use include: title, artist, genre, subject, copyright, srcform, comment.

```
__init__(fps=5, codec=None, bitrate=None, extra_args=None, metadata=None)
```

MovieWriter

**Parameters**

fps: int

Framerate for movie.

codec: string or None, optional

The codec to use. If None (the default) the setting in the rcParam animation.codec is used.

bitrate: int or None, optional

The bitrate for the saved movie file, which is one way to control the output file size and quality. The default value is None, which uses the value stored in the rcParam animation.bitrate. A value of -1 implies that the bitrate should be determined automatically by the underlying utility.

extra_args: list of strings or None

A list of extra string arguments to be passed to the underlying movie utility. The default is None, which passes the additional arguments in the ‘animation.extra_args’ rcParam.

metadata: dict of string:string or None

A dictionary of keys and values for metadata to include in the output file. Some keys that may be of use include: title, artist, genre, subject, copyright, srcform, comment.

**Methods**

```
__init__(fps, codec, bitrate, extra_args, ...)) MovieWriter
bin_path() Returns the binary path to the commandline tool used by a specific subclass.
cleanup() Clean-up and collect the process used to write the movie file.
finish() Finish any processing for writing the movie.
grab_frame(**savefig_kwargs) Grab the image information from the figure and save as a movie frame.
isAvailable() Check to see if a MovieWriter subclass is actually available by running the commandline tool.
```

Continued on next page
**saving(***args, **kw)**  
Context manager to facilitate writing the movie file.

**setup(fig, outfile, dpi[, frame_prefix, ...])**  
Perform setup for writing the movie file.

### Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>frame_size</strong></td>
<td>A tuple (width, height) in pixels of a movie frame.</td>
</tr>
</tbody>
</table>

#### matplotlib.animation.FileMovieWriter

**class** matplotlib.animation.FileMovieWriter(**args**, **kwargs)**  

*MovieWriter* for writing to individual files and stitching at the end.

This must be sub-classed to be useful.

**__init__**(**args**, **kwargs)**

### Methods

**__init__**(**args**, **kwargs)**

- **bin_path()**  
  Returns the binary path to the commandline tool used by a specific subclass.

- **cleanup()**

- **finish()**

- **grab_frame(***savefig_kwargs)**  
  Grab the image information from the figure and save as a movie frame.

- **isAvailable()**  
  Check to see if a MovieWriter subclass is actually available by running the commandline tool.

- **saving(***args, **kw)**  
  Context manager to facilitate writing the movie file.

- **setup(fig, outfile, dpi[, frame_prefix, ...])**  
  Perform setup for writing the movie file.

### Attributes

- **frame_format**  
  Format (png, jpeg, etc.) to use for saving the frames, which can be decided by the individual subclasses.

- **frame_size**  
  A tuple (width, height) in pixels of a movie frame.

and mixins are provided

### 17.3. Helper Classes
AVConvBase Mixin class for avconv output.
FFmpegBase Mixin class for FFmpeg output.
ImageMagickBase Mixin class for ImageMagick output.

matplotlib.animation.AVConvBase

class matplotlib.animation.AVConvBase
    Mixin class for avconv output.
    To be useful this must be multiply-inherited from a MovieWriterBase sub-class.
    __init__()
        Initialize self. See help(type(self)) for accurate signature.

Attributes

    args_key
    exec_key
    output_args

matplotlib.animation.FFmpegBase

class matplotlib.animation.FFmpegBase
    Mixin class for FFmpeg output.
    To be useful this must be multiply-inherited from a MovieWriterBase sub-class.
    __init__()
        Initialize self. See help(type(self)) for accurate signature.

Attributes

    args_key
    exec_key
    output_args

matplotlib.animation.ImageMagickBase

class matplotlib.animation.ImageMagickBase
    Mixin class for ImageMagick output.
    To be useful this must be multiply-inherited from a MovieWriterBase sub-class.
    __init__()
        Initialize self. See help(type(self)) for accurate signature.
17.4 Inheritance Diagrams

See the source code for how to easily implement new MovieWriter classes.

17.5 Deprecated

MencoderBase
MencoderFileWriter  Deprecated since version 2.0.
17.5.1 `matplotlib.animation.MencoderBase`

```python
class matplotlib.animation.MencoderBase

__init__()
    Initialize self. See help(type(self)) for accurate signature.

Attributes

allowed_metadata
args_key
exec_key
output_args
```

17.5.2 `matplotlib.animation.MencoderFileWriter`

```python
class matplotlib.animation.MencoderFileWriter(*args, **kwargs)

__init__(*args, **kwargs)
    Deprecated since version 2.0: Support for mencoder is only partially functional, and will be removed entirely in 2.2. Please use ffmpeg instead.

__init__(*args, **kwargs)
    Deprecated since version 2.0: Support for mencoder is only partially functional, and will be removed entirely in 2.2. Please use ffmpeg instead.

Methods

__init__(*args, **kwargs)
    Deprecated since version 2.0.

bin_path()
    Returns the binary path to the commandline tool used by a specific subclass.

cleanup()

finish()

grab_frame(**savefig_kwargs)
    Grab the image information from the figure and save as a movie frame.

isAvailable()
    Check to see if a MovieWriter subclass is actually available by running the commandline tool.

saving(*args, **kw)
    Context manager to facilitate writing the movie file.
```

Continued on next page
Table 17.37 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>setup(fig, outfile, dpi[, frame_prefix, ...])</td>
<td>Perform setup for writing the movie file.</td>
</tr>
</tbody>
</table>

Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>allowed_metadata</td>
<td></td>
</tr>
<tr>
<td>args_key</td>
<td></td>
</tr>
<tr>
<td>exec_key</td>
<td></td>
</tr>
<tr>
<td>frame_format</td>
<td>Format (png, jpeg, etc.) to use for saving the frames, which can be decided by the individual subclasses.</td>
</tr>
<tr>
<td>frame_size</td>
<td>A tuple (width, height) in pixels of a movie frame.</td>
</tr>
<tr>
<td>output_args</td>
<td></td>
</tr>
<tr>
<td>supported_formats</td>
<td></td>
</tr>
</tbody>
</table>

17.5.3 matplotlib.animation.MencoderWriter

class matplotlib.animation.MencoderWriter(*args, **kwargs)

Deprecated since version 2.0: Support for mencoder is only partially functional, and will be removed entirely in 2.2. Please use ffmpeg instead.

__init__(*args, **kwargs)

Deprecated since version 2.0: Support for mencoder is only partially functional, and will be removed entirely in 2.2. Please use ffmpeg instead.

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong>(*args, **kw)</td>
<td>Deprecated since version 2.0.</td>
</tr>
<tr>
<td>bin_path()</td>
<td>Returns the binary path to the commandline tool used by a specific subclass.</td>
</tr>
<tr>
<td>cleanup()</td>
<td>Clean-up and collect the process used to write the movie file.</td>
</tr>
<tr>
<td>finish()</td>
<td>Finish any processing for writing the movie.</td>
</tr>
<tr>
<td>grab_frame(**savefig_kwargs)</td>
<td>Grab the image information from the figure and save as a movie frame.</td>
</tr>
<tr>
<td>isAvailable()</td>
<td>Check to see if a MovieWriter subclass is actually available by running the commandline tool.</td>
</tr>
<tr>
<td>saving(*args, **kw)</td>
<td>Context manager to facilitate writing the movie file.</td>
</tr>
<tr>
<td>setup(fig, outfile, dpi)</td>
<td>Perform setup for writing the movie file.</td>
</tr>
</tbody>
</table>

Attributes

17.5. Deprecated
<table>
<thead>
<tr>
<th>allowed_metadata</th>
<th>args_key</th>
<th>exec_key</th>
<th>frame_size</th>
<th>outputArgs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>A tuple (width, height) in pixels of a movie frame.</td>
<td></td>
</tr>
</tbody>
</table>
CHAPTER EIGHTEEN

ARTIST MODULE
18.1 Artist class

**class matplotlib.artist.Artist**

Abstract base class for someone who renders into a FigureCanvas.

18.1.1 Interactive

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Artist.add_callback</td>
<td>Adds a callback function that will be called whenever one of the Artist's properties changes.</td>
</tr>
<tr>
<td>Artist.format_cursor_data</td>
<td>Return cursor data string formatted.</td>
</tr>
<tr>
<td>Artist.get_contains</td>
<td>Return the _contains test used by the artist, or None for default.</td>
</tr>
<tr>
<td>Artist.get_cursor_data</td>
<td>Get the cursor data for a given event.</td>
</tr>
<tr>
<td>Artist.get_picker</td>
<td>Return the picker object used by this artist.</td>
</tr>
<tr>
<td>Artist.hitlist</td>
<td>List the children of the artist which contain the mouse event.</td>
</tr>
<tr>
<td>Artist.mouseover</td>
<td>Fire an event when property changed, calling all of the registered callbacks.</td>
</tr>
<tr>
<td>Artist.pchanged</td>
<td>Process pick event</td>
</tr>
<tr>
<td>Artist.pickable</td>
<td>Return True if Artist is pickable.</td>
</tr>
<tr>
<td>Artist.remove_callback</td>
<td>Remove a callback based on its id.</td>
</tr>
<tr>
<td>Artist.set_contains</td>
<td>Replace the contains test used by this artist.</td>
</tr>
<tr>
<td>Artist.set_picker</td>
<td>Set the epsilon for picking used by this artist.</td>
</tr>
<tr>
<td>Artist.contains</td>
<td>Test whether the artist contains the mouse event.</td>
</tr>
</tbody>
</table>

**matplotlib.artist.Artist.add_callback**

*Artist.add_callback(func)*

Adds a callback function that will be called whenever one of the *Artist*'s properties changes.

Returns an id that is useful for removing the callback with `remove_callback()` later.

**matplotlib.artist.Artist.format_cursor_data**

*Artist.format_cursor_data(data)*

Return cursor data string formatted.

**matplotlib.artist.Artist.get_contains**

*Artist.get_contains()*

Return the _contains test used by the artist, or None for default.
**Matplotlib, Release 2.0.2**

* matplotlib.artist.Artist.get_cursor_data

**Artist.get_cursor_data**(event)
Get the cursor data for a given event.

* matplotlib.artist.Artist.get_picker

**Artist.get_picker**()
Return the picker object used by this artist.

* matplotlib.artist.Artist.hitlist

**Artist.hitlist**(event)
List the children of the artist which contain the mouse event event.

* matplotlib.artist.Artist.mouseover

**Artist.mouseover**

* matplotlib.artist.Artist.pchanged

**Artist.pchanged**()
Fire an event when property changed, calling all of the registered callbacks.

* matplotlib.artist.Artist.pick

**Artist.pick**(mouseevent)
Process pick event
each child artist will fire a pick event if mouseevent is over the artist and the artist has picker set

* matplotlib.artist.Artist.pickable

**Artist.pickable**()
Return True if Artist is pickable.

* matplotlib.artist.Artist.remove_callback

**Artist.remove_callback**(oid)
Remove a callback based on its id.

See also:
**add_callback()** For adding callbacks

**matplotlib.artist.Artist.set_contains**

**Artist.set_contains(picker)**
Replace the contains test used by this artist. The new picker should be a callable function which determines whether the artist is hit by the mouse event:

```python
hit, props = picker(artist, mouseevent)
```

If the mouse event is over the artist, return $hit = True$ and $props$ is a dictionary of properties you want returned with the contains test.

ACCEPTS: a callable function

**matplotlib.artist.Artist.set_picker**

**Artist.set_picker(picker)**
Set the epsilon for picking used by this artist

$picker$ can be one of the following:

- **None**: picking is disabled for this artist (default)
- A boolean: if $True$ then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist
- A float: if picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if it’s data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event
- A function: if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```python
hit, props = picker(artist, mouseevent)
```

to determine the hit test. If the mouse event is over the artist, return $hit=True$ and props is a dictionary of properties you want added to the PickEvent attributes.

ACCEPTS: [None|float|boolean|callable]

**matplotlib.artist.Artist.contains**

**Artist.contains(mouseevent)**
Test whether the artist contains the mouse event.

Returns the truth value and a dictionary of artist specific details of selection, such as which points are contained in the pick radius. See individual artists for details.
18.1.2 Margins and Autoscaling

`Artist.sticky_edges`  
x and y sticky edge lists.

`matplotlib.artist.Artist.sticky_edges`

`Artist.sticky_edges`  
x and y sticky edge lists.

When performing autoscaling, if a data limit coincides with a value in the corresponding sticky_edges list, then no margin will be added—the view limit “sticks” to the edge. A typical use case is histograms, where one usually expects no margin on the bottom edge (0) of the histogram.

This attribute cannot be assigned to; however, the x and y lists can be modified in place as needed.

**Examples**

```python
>>> artist.sticky_edges.x[:] = (xmin, xmax)
>>> artist.sticky_edges.y[:] = (ymin, ymax)
```

18.1.3 Clipping

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Artist.get_clip_box</code></td>
<td>Return artist clipbox</td>
</tr>
<tr>
<td><code>Artist.get_clip_on</code></td>
<td>Return whether artist uses clipping</td>
</tr>
<tr>
<td><code>Artist.get_clip_path</code></td>
<td>Return artist clip path</td>
</tr>
<tr>
<td><code>Artist.set_clip_box</code></td>
<td>Set the artist’s clip <code>Bbox</code>.</td>
</tr>
<tr>
<td><code>Artist.set_clip_on</code></td>
<td>Set whether artist uses clipping.</td>
</tr>
<tr>
<td><code>Artist.set_clip_path</code></td>
<td>Set the artist’s clip path, which may be:</td>
</tr>
</tbody>
</table>

`matplotlib.artist.Artist.get_clip_box`

`Artist.get_clip_box()`  
Return artist clipbox

`matplotlib.artist.Artist.get_clip_on`

`Artist.get_clip_on()`  
Return whether artist uses clipping

`matplotlib.artist.Artist.get_clip_path`

`Artist.get_clip_path()`  
Return artist clip path
matplotlib.artist.Artist.set_clip_box

Artist.set_clip_box(clipbox)
Set the artist’s clip Bbox.

ACCEPTS: a matplotlib.transforms.Bbox instance

matplotlib.artist.Artist.set_clip_on

Artist.set_clip_on(b)
Set whether artist uses clipping.

When False artists will be visible out side of the axes which can lead to unexpected results.

ACCEPTS: [True | False]

matplotlib.artist.Artist.set_clip_path

Artist.set_clip_path(path, transform=None)
Set the artist’s clip path, which may be:

• a Patch (or subclass) instance

• a Path instance, in which case an optional Transform instance may be provided, which will be applied to the path before using it for clipping.

• None, to remove the clipping path

For efficiency, if the path happens to be an axis-aligned rectangle, this method will set the clipping box to the corresponding rectangle and set the clipping path to None.

ACCEPTS: [ (Path, Transform) | Patch | None ]

18.1.4 Bulk Properties

<table>
<thead>
<tr>
<th>Artist.update</th>
<th>Update the properties of this Artist from the dictionary prop.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Artist.update_from</td>
<td>Copy properties from other to self.</td>
</tr>
<tr>
<td>Artist.properties</td>
<td>return a dictionary mapping property name -&gt; value for all Artist props</td>
</tr>
<tr>
<td>Artist.set</td>
<td>A property batch setter.</td>
</tr>
</tbody>
</table>

matplotlib.artist.Artist.update

Artist.update(props)
Update the properties of this Artist from the dictionary prop.
Matplotlib, Release 2.0.2

`matplotlib.artist.Artist.update_from`

`Artist.update_from(other)`

Copy properties from `other` to `self`.

`matplotlib.artist.Artist.properties`

`Artist.properties()`

return a dictionary mapping property name - value for all Artist props

`matplotlib.artist.Artist.set`

`Artist.set(**kwargs)`

A property batch setter. Pass `kwargs` to set properties.

### 18.1.5 Drawing

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Artist.draw</code></td>
<td>Derived classes drawing method</td>
</tr>
<tr>
<td><code>Artist.get_animated</code></td>
<td>Return the artist’s animated state</td>
</tr>
<tr>
<td><code>Artist.set_animated</code></td>
<td>Set the artist’s animation state</td>
</tr>
<tr>
<td><code>Artist.get_agg_filter</code></td>
<td>return filter function to be used for agg filter</td>
</tr>
<tr>
<td><code>Artist.get_alpha</code></td>
<td>Return the alpha value used for blending - not supported on all</td>
</tr>
<tr>
<td><code>Artist.get_snap</code></td>
<td>Returns the snap setting which may be:</td>
</tr>
<tr>
<td><code>Artist.get_visible</code></td>
<td>Return the artist’s visibility</td>
</tr>
<tr>
<td><code>Artist.set_zorder</code></td>
<td>Return the <code>Artist</code>’s zorder.</td>
</tr>
<tr>
<td><code>Artist.set_agg_filter</code></td>
<td>set agg_filter fuction.</td>
</tr>
<tr>
<td><code>Artist.set_alpha</code></td>
<td>Set the alpha value used for blending - not supported on all backends.</td>
</tr>
<tr>
<td><code>Artist.set_sketch_params</code></td>
<td>Sets the sketch parameters.</td>
</tr>
<tr>
<td><code>Artist.set_snap</code></td>
<td>Sets the snap setting which may be:</td>
</tr>
<tr>
<td><code>Artist.get_rasterized</code></td>
<td>return True if the artist is to be rasterized</td>
</tr>
<tr>
<td><code>Artist.get_sketch_params</code></td>
<td>Returns the sketch parameters for the artist.</td>
</tr>
<tr>
<td><code>Artist.set_path_effects</code></td>
<td>set path_effects, which should be a list of instances of</td>
</tr>
<tr>
<td><code>Artist.set_rasterized</code></td>
<td>Force rasterized (bitmap) drawing in vector backend output.</td>
</tr>
<tr>
<td><code>Artist.zorder</code></td>
<td>Set the artist’s visibility.</td>
</tr>
<tr>
<td><code>Artist.set_visible</code></td>
<td>Set the zorder for the artist.</td>
</tr>
<tr>
<td><code>Artist.set_window_extent</code></td>
<td>Get the axes bounding box in display space.</td>
</tr>
<tr>
<td><code>Artist.get_path_effects</code></td>
<td>Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.</td>
</tr>
</tbody>
</table>
**matplotlib.artist.Artist.draw**

\[ \text{Artist.draw}(\text{renderer, *args, **kwargs}) \]

Derived classes drawing method

**matplotlib.artist.Artist.get_animated**

\[ \text{Artist.get_animated}() \]

Return the artist’s animated state

**matplotlib.artist.Artist.set_animated**

\[ \text{Artist.set_animated}(b) \]

Set the artist’s animation state.

ACCEPTS: [True | False]

**matplotlib.artist.Artist.get_agg_filter**

\[ \text{Artist.get_agg_filter}() \]

return filter function to be used for agg filter

**matplotlib.artist.Artist.get_alpha**

\[ \text{Artist.get_alpha}() \]

Return the alpha value used for blending - not supported on all backends

**matplotlib.artist.Artist.get_snap**

\[ \text{Artist.get_snap}() \]

Returns the snap setting which may be:

- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

**matplotlib.artist.Artist.get_visible**

\[ \text{Artist.get_visible}() \]

Return the artist’s visibility
**matplotlib.artist.Artist.get_zorder**

`Artist.get_zorder()`

Return the *Artist’s* zorder.

**matplotlib.artist.Artist.set_agg_filter**

`Artist.set_agg_filter(filter_func)`

set agg_filter function.

**matplotlib.artist.Artist.set_alpha**

`Artist.set_alpha(alpha)`

Set the alpha value used for blending - not supported on all backends.

ACCEPTS: float (0.0 transparent through 1.0 opaque)

**matplotlib.artist.Artist.set_sketch_params**

`Artist.set_sketch_params(scale=None, length=None, randomness=None)`

Sets the sketch parameters.

Parameters:

- **scale**: float, optional
  - The amplitude of the wiggle perpendicular to the source line, in pixels. If `scale` is `None`, or not provided, no sketch filter will be provided.

- **length**: float, optional
  - The length of the wiggle along the line, in pixels (default 128.0)

- **randomness**: float, optional
  - The scale factor by which the length is shrunken or expanded (default 16.0)

**matplotlib.artist.Artist.set_snap**

`Artist.set_snap(snaps)`

Sets the snap setting which may be:

- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.
**matplotlib.artist.Artist.get_rasterized**

`Artist.get_rasterized()`
return True if the artist is to be rasterized

**matplotlib.artist.Artist.get_sketch_params**

`Artist.get_sketch_params()`
Returns the sketch parameters for the artist.

**Returns sketch_params : tuple or None**
A 3-tuple with the following elements:
- `scale`: The amplitude of the wiggle perpendicular to the source line.
- `length`: The length of the wiggle along the line.
- `randomness`: The scale factor by which the length is shrunken or expanded.

May return `None` if no sketch parameters were set.

**matplotlib.artist.Artist.set_path_effects**

`Artist.set_path_effects(path_effects)`
set path_effects, which should be a list of instances of matplotlib.pathEffect._Base class or its derivatives.

**matplotlib.artist.Artist.set_rasterized**

`Artist.set_rasterized(rasterized)`
Force rasterized (bitmap) drawing in vector backend output.
Defaults to `None`, which implies the backend’s default behavior

**ACCEPTS: [True | False | None]**

**matplotlib.artist.Artist.zorder**

`Artist.zorder = 0`

**matplotlib.artist.Artist.set_visible**

`Artist.set_visible(b)`
Set the artist’s visibility.

**ACCEPTS: [True | False]**
**matplotlib.artist.Artist.set_zorder**

*Artist.set_zorder(level)*

Set the zorder for the artist. Artists with lower zorder values are drawn first.

*ACCEPTS:* any number

**matplotlib.artist.Artist.get_window_extent**

*Artist.get_window_extent(renderer)*

Get the axes bounding box in display space. Subclasses should override for inclusion in the bounding box “tight” calculation. Default is to return an empty bounding box at 0, 0.

Be careful when using this function, the results will not update if the artist window extent of the artist changes. The extent can change due to any changes in the transform stack, such as changing the axes limits, the figure size, or the canvas used (as is done when saving a figure). This can lead to unexpected behavior where interactive figures will look fine on the screen, but will save incorrectly.

**matplotlib.artist.Artist.get_path_effects**

*Artist.get_path_effects()*

**matplotlib.artist.Artist.get_transformed_clip_path_and_affine**

*Artist.get_transformed_clip_path_and_affine()*

Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

### 18.1.6 Figure and Axes

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Artist.remove</td>
<td>Remove the artist from the figure if possible.</td>
</tr>
<tr>
<td>Artist.axes</td>
<td>The <em>Axes</em> instance the artist resides in, or <em>None</em>.</td>
</tr>
<tr>
<td>Artist.get_axes</td>
<td>Return the <em>Axes</em> instance the artist resides in, or <em>None</em>.</td>
</tr>
<tr>
<td>Artist.set_axes</td>
<td>Set the <em>Axes</em> instance in which the artist resides, if any.</td>
</tr>
<tr>
<td>Artist.set_figure</td>
<td>Set the <em>Figure</em> instance the artist belongs to.</td>
</tr>
<tr>
<td>Artist.get_figure</td>
<td>Return the <em>Figure</em> instance the artist belongs to.</td>
</tr>
<tr>
<td>Artist.is_figure_set</td>
<td>Returns True if the artist is assigned to a <em>Figure</em>.</td>
</tr>
</tbody>
</table>

**matplotlib.artist.Artist.remove**

*Artist.remove()*

Remove the artist from the figure if possible. The effect will not be visible until the figure is redrawn, e.g., with `matplotlib.axes.Axes.draw_idle()`. Call `matplotlib.axes.Axes.relim()` to
update the axes limits if desired.
Note: `relim()` will not see collections even if the collection was added to axes with `autolim = True`.
Note: there is no support for removing the artist’s legend entry.

**matplotlib.artist.Artist.axes**

`Artist.axes`

The `Axes` instance the artist resides in, or `None`.

**matplotlib.artist.Artist.get_axes**

`Artist.get_axes()`

Return the `Axes` instance the artist resides in, or `None`.
This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

**matplotlib.artist.Artist.set_axes**

`Artist.set_axes(axes)`

Set the `Axes` instance in which the artist resides, if any.
This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

ACCEPTS: a `Axes` instance

**matplotlib.artist.Artist.set_figure**

`Artist.set_figure(fig)`

Set the `Figure` instance the artist belongs to.

ACCEPTS: a `matplotlib.figure.Figure` instance

**matplotlib.artist.Artist.get_figure**

`Artist.get_figure()`

Return the `Figure` instance the artist belongs to.

**matplotlib.artist.Artist.is_figure_set**

`Artist.is_figure_set()`

Returns True if the artist is assigned to a `Figure`.

### 18.1.7 Children
**Artist.get_children**

Return a list of the child `Artist`'s this :class:`Artist` contains.

**Artist.findobj**

Find artist objects.

---

**matplotlib.artist.Artist.get_children**

`Artist.get_children()`

Return a list of the child `Artist`'s this :class:`Artist` contains.

**matplotlib.artist.Artist.findobj**

`Artist.findobj(match=None, include_self=True)`

Find artist objects.

Recursively find all `Artist` instances contained in self.

`match` can be

- None: return all objects contained in artist.
- function with signature `boolean = match(artist)` used to filter matches
- class instance: e.g., `Line2D`. Only return artists of class type.

If `include_self` is True (default), include self in the list to be checked for a match.

---

### 18.1.8 Transform

**Artist.set_transform**

Set the `Transform` instance used by this artist.

**Artist.get_transform**

Return the `Transform` instance used by this artist.

**Artist.is_transform_set**

Returns `True` if `Artist` has a transform explicitly set.

---

**matplotlib.artist.Artist.set_transform**

`Artist.set_transform(t)`

Set the `Transform` instance used by this artist.

**ACCEPTS:** `Transform` instance

**matplotlib.artist.Artist.get_transform**

`Artist.get_transform()`

Return the `Transform` instance used by this artist.
matplotlib.artist.Artist.is_transform_set

Artist.is_transform_set()

Returns True if Artist has a transform explicitly set.

18.1.9 Units

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Artist.convert_xunits</td>
<td>For artists in an axes, if the xaxis has units support, convert x using xaxis unit type</td>
</tr>
<tr>
<td>Artist.convert_yunits</td>
<td>For artists in an axes, if the yaxis has units support, convert y using yaxis unit type</td>
</tr>
<tr>
<td>Artist.have_units</td>
<td>Return True if units are set on the x or y axes</td>
</tr>
</tbody>
</table>

matplotlib.artist.Artist.convert_xunits

Artist.convert_xunits(x)

For artists in an axes, if the xaxis has units support, convert x using xaxis unit type

matplotlib.artist.Artist.convert_yunits

Artist.convert_yunits(y)

For artists in an axes, if the yaxis has units support, convert y using yaxis unit type

matplotlib.artist.Artist.have_units

Artist.have_units()

Return True if units are set on the x or y axes

18.1.10 Metadata

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Artist.get_gid</td>
<td>Returns the group id</td>
</tr>
<tr>
<td>Artist.get_label</td>
<td>Get the label used for this artist in the legend.</td>
</tr>
<tr>
<td>Artist.set_gid</td>
<td>Sets the (group) id for the artist</td>
</tr>
<tr>
<td>Artist.set_label</td>
<td>Set the label to s for auto legend.</td>
</tr>
<tr>
<td>Artist.get_url</td>
<td>Returns the url</td>
</tr>
<tr>
<td>Artist.set_url</td>
<td>Sets the url for the artist</td>
</tr>
<tr>
<td>Artist.aname</td>
<td></td>
</tr>
</tbody>
</table>

matplotlib.artist.Artist.get_gid

Artist.get_gid()

Returns the group id
matplotlib.artist.Artist.get_label

Artist.get_label()
Get the label used for this artist in the legend.

matplotlib.artist.Artist.set_gid

Artist.set_gid(gid)
Sets the (group) id for the artist
    ACCEPTS: an id string

matplotlib.artist.Artist.set_label

Artist.set_label(s)
Set the label to s for auto legend.
    ACCEPTS: string or anything printable with ‘%s’ conversion.

matplotlib.artist.Artist.get_url

Artist.get_url()
Returns the url

matplotlib.artist.Artist.set_url

Artist.set_url(url)
Sets the url for the artist
    ACCEPTS: a url string

matplotlib.artist.Artist.aname

Artist.aname = ‘Artist’

18.1.11 Stale

| Artist.stale | If the artist is ‘stale’ and needs to be re-drawn for the output to match the internal state of the artist. |
matplotlib.artist.Artist.stale

**Artist.stale**

If the artist is ‘stale’ and needs to be re-drawn for the output to match the internal state of the artist.

### 18.2 Functions

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>allow_rasterization</td>
<td>Decorator for Artist.draw method.</td>
</tr>
<tr>
<td>get</td>
<td>Return the value of object’s property.</td>
</tr>
<tr>
<td>getp</td>
<td>Return the value of object’s property.</td>
</tr>
<tr>
<td>setp</td>
<td>Set a property on an artist object.</td>
</tr>
<tr>
<td>kwdoc</td>
<td>A helper class to inspect an Artist and return information about it's settable properties and their current values.</td>
</tr>
</tbody>
</table>

#### 18.2.1 matplotlib.artist.allow_rasterization

**matplotlib.artist.allow_rasterization(draw)**

Decorator for Artist.draw method. Provides routines that run before and after the draw call. The before and after functions are useful for changing artist-dependant renderer attributes or making other setup function calls, such as starting and flushing a mixed-mode renderer.

#### 18.2.2 matplotlib.artist.get

**matplotlib.artist.get(obj, property=None)**

Return the value of object’s property. property is an optional string for the property you want to return.

Example usage:

```
getp(obj)  # get all the object properties
getp(obj, 'linestyle')  # get the linestyle property
```

*obj* is a Artist instance, e.g., Line2D or an instance of a Axes or matplotlib.text.Text. If the *property* is ‘somename’, this function returns

```
obj.get_somename()
```

*getp()* can be used to query all the gettable properties with *getp(obj)*. Many properties have aliases for shorter typing, e.g. ‘lw’ is an alias for ‘linewidth’. In the output, aliases and full property names will be listed as:

```
property or alias = value
```

e.g.:

```
linewidth or lw = 2
```
18.2.3 matplotlib.artist.getp

matplotlib.artist.getp(obj, property=None)

Return the value of object’s property. property is an optional string for the property you want to return.

Example usage:

```python
getp(obj)  # get all the object properties
getp(obj, 'linestyle')  # get the linestyle property
```

obj is a Artist instance, e.g., Line2D or an instance of a Axes or matplotlib.text.Text. If the property is ‘somename’, this function returns

```python
obj.get_somename()
```

getp() can be used to query all the gettable properties with getp(obj). Many properties have aliases for shorter typing, e.g. ‘lw’ is an alias for ‘linewidth’. In the output, aliases and full property names will be listed as:

property or alias = value

e.g.:

linewidth or lw = 2

18.2.4 matplotlib.artist.setp

matplotlib.artist.setp(obj, *args, **kwargs)

Set a property on an artist object.

matplotlib supports the use of setp() (“set property”) and getp() to set and get object properties, as well as to do introspection on the object. For example, to set the linestyle of a line to be dashed, you can do:

```python
>>> line, = plot([1,2,3])
>>> setp(line, linestyle='--')
```

If you want to know the valid types of arguments, you can provide the name of the property you want to set without a value:

```python
>>> setp(line, 'linestyle')
linestyle: [ '-' | '--' | '-.' | ':' | 'steps' | 'None' ]
```

If you want to see all the properties that can be set, and their possible values, you can do:

```python
>>> setp(line)
... long output listing omitted
```

setp() operates on a single instance or a list of instances. If you are in query mode introspecting the possible values, only the first instance in the sequence is used. When actually setting values, all the
instances will be set. e.g., suppose you have a list of two lines, the following will make both lines thicker and red:

```python
>>> x = arange(0,1.0,0.01)
>>> y1 = sin(2*pi*x)
>>> y2 = sin(4*pi*x)
>>> lines = plot(x, y1, x, y2)
>>> setp(lines, linewidth=2, color='r')
```

`setp()` works with the MATLAB style string/value pairs or with python kwargs. For example, the following are equivalent:

```python
>>> setp(lines, 'linewidth', 2, 'color', 'r')  # MATLAB style
>>> setp(lines, linewidth=2, color='r')       # python style
```

### 18.2.5 matplotlib.artist.kwdoc

`matplotlib.artist.kwdoc(a)`

### 18.2.6 matplotlib.artist.ArtistInspector

**class matplotlib.artist.ArtistInspector(o)**

A helper class to inspect an `Artist` and return information about it’s settable properties and their current values.

Initialize the artist inspector with an `Artist` or sequence of `Artists`. If a sequence is used, we assume it is a homogeneous sequence (all `Artists` are of the same type) and it is your responsibility to make sure this is so.

**__init__ (o)**

Initialize the artist inspector with an `Artist` or sequence of `Artists`. If a sequence is used, we assume it is a homogeneous sequence (all `Artists` are of the same type) and it is your responsibility to make sure this is so.

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong> (o)</td>
<td>Initialize the artist inspector with an <code>Artist</code> or sequence of <code>Artists</code>.</td>
</tr>
<tr>
<td>aliased_name(s)</td>
<td>return ‘PROPNAME or alias’ if <code>s</code> has an alias, else return</td>
</tr>
<tr>
<td>aliased_name_rest(s, target)</td>
<td>return ‘PROPNAME or alias’ if <code>s</code> has an alias, else return</td>
</tr>
<tr>
<td>findobj([match])</td>
<td>Recursively find all <code>matplotlib.artist.Artist</code> instances contained in <code>self</code>.</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_aliases()</td>
<td>Get a dict mapping <strong>fullname</strong> -&gt; <strong>alias</strong> for each <strong>alias</strong> in the <code>ArtistInspector</code>.</td>
</tr>
<tr>
<td>get_setters()</td>
<td>Get the attribute strings with setters for object.</td>
</tr>
<tr>
<td>get_valid_values(attr)</td>
<td>Get the legal arguments for the setter associated with <strong>attr</strong>.</td>
</tr>
<tr>
<td>is_alias(o)</td>
<td>Return <strong>True</strong> if method object <strong>o</strong> is an alias for another function.</td>
</tr>
<tr>
<td>pprint_getters()</td>
<td>Return the getters and actual values as list of strings.</td>
</tr>
<tr>
<td>pprint_setters([prop, leadingspace])</td>
<td>If <strong>prop</strong> is <strong>None</strong>, return a list of strings of all settable properties and their valid values.</td>
</tr>
<tr>
<td>pprint_setters_rest([prop, leadingspace])</td>
<td>If <strong>prop</strong> is <strong>None</strong>, return a list of strings of all settable properties and their valid values.</td>
</tr>
<tr>
<td>properties()</td>
<td>return a dictionary mapping property name -&gt; value</td>
</tr>
</tbody>
</table>
CHAPTER
NINETEEN

AXES CLASS

```python
class matplotlib.axes.Axes(fig, rect, facecolor=None, frameon=True, sharex=None, sharey=None, label='', xscale=None, yscale=None, axisbg=None, **kwargs)
```

The `Axes` contains most of the figure elements: `Axis`, `Tick`, `Line2D`, `Text`, `Polygon`, etc., and sets the coordinate system.

The `Axes` instance supports callbacks through a callbacks attribute which is a `CallbackRegistry` instance. The events you can connect to are ‘xlim_changed’ and ‘ylim_changed’ and the callback will be called with `func(ax)` where `ax` is the `Axes` instance.

Table of Contents

- Plotting
  - Basic
  - Spans
  - Spectral
  - Statistics
  - Binned
  - Contours
  - Array
  - Unstructured Triangles
  - Text and Annotations
  - Fields
- Clearing
- Appearance
- Property cycle
- Axis / limits
  - Axis Limits and direction
19.1 Plotting

19.1.1 Basic

<table>
<thead>
<tr>
<th><strong>Axes.plot</strong></th>
<th>Plot lines and/or markers to the Axes.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Axes.errorbar</strong></td>
<td>Plot an errorbar graph.</td>
</tr>
<tr>
<td><strong>Axes.scatter</strong></td>
<td>Make a scatter plot of x vs y</td>
</tr>
<tr>
<td><strong>Axes.plot_date</strong></td>
<td>A plot with data that contains dates.</td>
</tr>
<tr>
<td><strong>Axes.step</strong></td>
<td>Make a step plot.</td>
</tr>
<tr>
<td><strong>Axes.loglog</strong></td>
<td>Make a plot with log scaling on both the x and y axis.</td>
</tr>
<tr>
<td><strong>Axes.semilogx</strong></td>
<td>Make a plot with log scaling on the x axis.</td>
</tr>
<tr>
<td><strong>Axes.semilogy</strong></td>
<td>Make a plot with log scaling on the y axis.</td>
</tr>
<tr>
<td><strong>Axes.fill_between</strong></td>
<td>Make filled polygons between two curves.</td>
</tr>
<tr>
<td><strong>Axes.fill_betweenx</strong></td>
<td>Make filled polygons between two horizontal curves.</td>
</tr>
</tbody>
</table>
Table 19.1 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Axes.bar</td>
<td>Make a bar plot.</td>
</tr>
<tr>
<td>Axes.barh</td>
<td>Make a horizontal bar plot.</td>
</tr>
<tr>
<td>Axes.stem</td>
<td>Create a stem plot.</td>
</tr>
<tr>
<td>Axes.eventplot</td>
<td>Plot identical parallel lines at specific positions.</td>
</tr>
<tr>
<td>Axes.pie</td>
<td>Plot a pie chart.</td>
</tr>
<tr>
<td>Axes.stackplot</td>
<td>Draws a stacked area plot.</td>
</tr>
<tr>
<td>Axes.broken_barh</td>
<td>Plot horizontal bars.</td>
</tr>
<tr>
<td>Axes.vlines</td>
<td>Plot vertical lines.</td>
</tr>
<tr>
<td>Axes.hlines</td>
<td>Plot horizontal lines at each y from xmin to xmax.</td>
</tr>
<tr>
<td>Axes.fill</td>
<td>Plot filled polygons.</td>
</tr>
</tbody>
</table>

matplotlib.axes.Axes.plot

Axes.plot(*args, **kwargs)

Plot lines and/or markers to the Axes. args is a variable length argument, allowing for multiple x, y pairs with an optional format string. For example, each of the following is legal:

```
plot(x, y)          # plot x and y using default line style and color
plot(x, y, 'bo')   # plot x and y using blue circle markers
plot(y)            # plot y using x as index array 0..N-1
plot(y, 'r+')       # ditto, but with red plusses
```

If x and/or y is 2-dimensional, then the corresponding columns will be plotted.

If used with labeled data, make sure that the color spec is not included as an element in data, as otherwise the last case `plot("v","r", data="v":..., "r"::...)` can be interpreted as the first case which would do `plot(v, r)` using the default line style and color.

If not used with labeled data (i.e., without a data argument), an arbitrary number of x, y, fmt groups can be specified, as in:

```
a.plot(x1, y1, 'g^', x2, y2, 'g-')
```

Return value is a list of lines that were added.

By default, each line is assigned a different style specified by a ‘style cycle’. To change this behavior, you can edit the axes.prop_cycle rcParam.

The following format string characters are accepted to control the line style or marker:
<table>
<thead>
<tr>
<th>character</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>' - '</td>
<td>solid line style</td>
</tr>
<tr>
<td>' -- '</td>
<td>dashed line style</td>
</tr>
<tr>
<td>' - . '</td>
<td>dash-dot line style</td>
</tr>
<tr>
<td>' : '</td>
<td>dotted line style</td>
</tr>
<tr>
<td>' , '</td>
<td>point marker</td>
</tr>
<tr>
<td>' o '</td>
<td>circle marker</td>
</tr>
<tr>
<td>' v '</td>
<td>triangle_down marker</td>
</tr>
<tr>
<td>' x '</td>
<td>triangle_up marker</td>
</tr>
<tr>
<td>' &lt; '</td>
<td>triangle_left marker</td>
</tr>
<tr>
<td>' &gt; '</td>
<td>triangle_right marker</td>
</tr>
<tr>
<td>' 1 '</td>
<td>tri_down marker</td>
</tr>
<tr>
<td>' 2 '</td>
<td>tri_up marker</td>
</tr>
<tr>
<td>' 3 '</td>
<td>tri_left marker</td>
</tr>
<tr>
<td>' 4 '</td>
<td>tri_right marker</td>
</tr>
<tr>
<td>' s '</td>
<td>square marker</td>
</tr>
<tr>
<td>' p '</td>
<td>pentagon marker</td>
</tr>
<tr>
<td>' * '</td>
<td>star marker</td>
</tr>
<tr>
<td>' h '</td>
<td>hexagon1 marker</td>
</tr>
<tr>
<td>' H '</td>
<td>hexagon2 marker</td>
</tr>
<tr>
<td>' + '</td>
<td>plus marker</td>
</tr>
<tr>
<td>' x '</td>
<td>x marker</td>
</tr>
<tr>
<td>' D '</td>
<td>diamond marker</td>
</tr>
<tr>
<td>' d '</td>
<td>thin_diamond marker</td>
</tr>
<tr>
<td>' l '</td>
<td>vline marker</td>
</tr>
<tr>
<td>' _ '</td>
<td>hline marker</td>
</tr>
</tbody>
</table>

The following color abbreviations are supported:

<table>
<thead>
<tr>
<th>character</th>
<th>color</th>
</tr>
</thead>
<tbody>
<tr>
<td>' b '</td>
<td>blue</td>
</tr>
<tr>
<td>' g '</td>
<td>green</td>
</tr>
<tr>
<td>' r '</td>
<td>red</td>
</tr>
<tr>
<td>' c '</td>
<td>cyan</td>
</tr>
<tr>
<td>' m '</td>
<td>magenta</td>
</tr>
<tr>
<td>' y '</td>
<td>yellow</td>
</tr>
<tr>
<td>' k '</td>
<td>black</td>
</tr>
<tr>
<td>' w '</td>
<td>white</td>
</tr>
</tbody>
</table>

In addition, you can specify colors in many weird and wonderful ways, including full names ('green'), hex strings ('#008000'), RGB or RGBA tuples ((0,1,0,1)) or grayscale intensities as a string ('0.8'). Of these, the string specifications can be used in place of a fmt group, but the tuple forms can be used only as kwargs.

Line styles and colors are combined in a single format string, as in 'bo' for blue circles.

The kwargs can be used to set line properties (any property that has a set_* method). You can use this to set a line label (for auto legends), linewidth, anitialising, marker face color, etc. Here is an
example:

```python
plot([1,2,3], [1,2,3], 'go-', label='line 1', linewidth=2)
plot([1,2,3], [1,4,9], 'rs', label='line 2')
axis([0, 4, 0, 10])
legend()
```

If you make multiple lines with one plot command, the kwargs apply to all those lines, e.g.:

```python
plot(x1, y1, x2, y2, antialiased=False)
```

Neither line will be antialiased.

You do not need to use format strings, which are just abbreviations. All of the line properties can be controlled by keyword arguments. For example, you can set the color, marker, linestyle, and markerfacecolor with:

```python
plot(x, y, color='green', linestyle='dashed', marker='o',
     markerfacecolor='blue', markersize=12).
```

See `Line2D` for details.

The kwargs are `Line2D` properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)][Path</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>drawstyle</td>
<td>['default'</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>['full'</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with '%s' conversion.</td>
</tr>
<tr>
<td>linestype or ls</td>
<td>['solid'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>marker</td>
<td>A valid marker style</td>
</tr>
<tr>
<td>markeredgecolor or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
</tbody>
</table>
Table 19.2 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>[None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt’</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter’</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

kwargsscalex and scaley, if defined, are passed on to autoscale_view() to determine whether the x and y axes are autoscaled; the default is True.

**Note:** In addition to the above described arguments, this function can take a data keyword argument. If such a data argument is given, the following arguments are replaced by data[<arg>]:

- All arguments with the following names: ‘x’, ‘y’.

```
matplotlib.axes.Axes.errorbar

Axes.errorbar(x, y, yerr=None, xerr=None, fmt='', color=None, elinewidth=None, capsize=None, barsabove=False, lolims=False, uplims=False, xlolims=False, xulims=False, errorevery=1, capthick=None, **kwargs)
```

Plot an errorbar graph.

Plot x versus y with error deltas in yerr and xerr. Vertical errorbars are plotted if yerr is not None. Horizontal errorbars are plotted if xerr is not None.

x, y, xerr, and yerr can all be scalars, which plots a single error bar at x, y.

**Parameters**

- **x**: scalar or array-like
- **y**: scalar or array-like
- **xerr/yerr**: scalar or array-like, shape(N,) or shape(2,N), optional
If a scalar number, len(N) array-like object, or a N-element array-like object, errorbars are drawn at +/-value relative to the data. Default is None.

If a sequence of shape 2xN, errorbars are drawn at -row1 and +row2 relative to the data.

fmt : plot format string, optional, default: None

The plot format symbol. If fmt is ‘none’ (case-insensitive), only the errorbars are plotted. This is used for adding errorbars to a bar plot, for example. Default is ‘’, an empty plot format string; properties are then identical to the defaults for plot().

ecolor : mpl color, optional, default: None

A matplotlib color arg which gives the color the errorbar lines; if None, use the color of the line connecting the markers.

elinewidth : scalar, optional, default: None

The linewidth of the errorbar lines. If None, use the linewidth.

capsize : scalar, optional, default: None

The length of the error bar caps in points; if None, it will take the value from errorbar.capsize rcParam.

capthick : scalar, optional, default: None

An alias kwarg to markeredgewidth (a.k.a. - mew). This setting is a more sensible name for the property that controls the thickness of the error bar cap in points. For backwards compatibility, if mew or markeredgewidth are given, then they will over-ride capthick. This may change in future releases.

barsabove : bool, optional, default: False

if True, will plot the errorbars above the plot symbols. Default is below.

lolims / uplims / xlolims / xuplims : bool, optional, default:None

These arguments can be used to indicate that a value gives only upper/lower limits. In that case a caret symbol is used to indicate this. lims-arguments may be of the same type as xerr and yerr. To use limits with inverted axes, set_xlim() or set_ylim() must be called before errorbar().

errorevery : positive integer, optional, default:1

subsamples the errorbars. e.g., if errorevery=5, errorbars for every 5-th data-point will be plotted. The data plot itself still shows all data points.

Returns plotline : Line2D instance

x, y plot markers and/or line

caplines : list of Line2D instances

error bar cap

barlinecols : list of LineCollection
horizontal and vertical error ranges.

**Other Parameters kwargs**: All other keyword arguments are passed on to the plot command for the markers. For example, this code makes big red squares with thick green edges:

```python
x, y, yerr = rand(3, 10)
errorbar(x, y, yerr, marker='s', mfc='red', mec='green', ms=20, mew=4)
```

where mfc, mec, ms and mew are aliases for the longer property names, markerfacecolor, markeredgecolor, markersize and markeredgewidth.

valid kwargs for the marker properties are

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>agg_filter</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>alpha</code></td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td><code>animated</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>antialiased</code> or aa</td>
<td>[True</td>
</tr>
<tr>
<td><code>axes</code></td>
<td>an Axes instance</td>
</tr>
<tr>
<td><code>clip_box</code></td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td><code>clip_on</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>clip_path</code></td>
<td>[ (Path, Transform)</td>
</tr>
<tr>
<td><code>color</code> or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td><code>contains</code></td>
<td>a callable function</td>
</tr>
<tr>
<td><code>dash_capstyle</code></td>
<td>['butt'</td>
</tr>
<tr>
<td><code>dash_joinstyle</code></td>
<td>['miter'</td>
</tr>
<tr>
<td><code>dashes</code></td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td><code>drawstyle</code></td>
<td>['default'</td>
</tr>
<tr>
<td><code>figure</code></td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td><code>fillstyle</code></td>
<td>['full'</td>
</tr>
<tr>
<td><code>gid</code></td>
<td>an id string</td>
</tr>
<tr>
<td><code>label</code></td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td><code>linestyle</code> or ls</td>
<td>['solid'</td>
</tr>
<tr>
<td><code>linewidth</code> or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td><code>marker</code></td>
<td>A valid marker style</td>
</tr>
<tr>
<td><code>markeredgecolor</code> or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td><code>markeredgewidth</code> or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td><code>markerfacecolor</code> or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td><code>markerfacecoloralt</code> or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td><code>markersize</code> or ms</td>
<td>float</td>
</tr>
<tr>
<td><code>markevery</code></td>
<td>[None</td>
</tr>
<tr>
<td><code>path_effects</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>picker</code></td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td><code>pickradius</code></td>
<td>float distance in points</td>
</tr>
<tr>
<td><code>rasterized</code></td>
<td>[True</td>
</tr>
</tbody>
</table>

Continued on next page
Table 19.3 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt', 'round', 'projecting']</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter', 'round', 'bevel']</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True, False]</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

**Examples**

![Graph](image)

**Note:** In addition to the above described arguments, this function can take a data keyword argument. If such a data argument is given, the following arguments are replaced by data[<arg>]:

- All arguments with the following names: ‘x’, ‘xerr’, ‘y’, ‘yerr’.
**matplotlib.axes.Axes.scatter**

*A.xes.scatter*(x, y, s=None, c=None, marker=None, cmap=None, norm=None, vmin=None, vmax=None, alpha=None, linewidths=None, verts=None, edgecolors=None, **kwargs)

Make a scatter plot of x vs y

Marker size is scaled by s and marker color is mapped to c

**Parameters** x, y : array_like, shape (n, )

Input data

s : scalar or array_like, shape (n, ), optional

size in points^2. Default is rcParams['lines.markersize'] ** 2.

c : color, sequence, or sequence of color, optional, default: ‘b’

c can be a single color format string, or a sequence of color specifications of length N, or a sequence of N numbers to be mapped to colors using the cmap and norm specified via kwargs (see below). Note that c should not be a single numeric RGB or RGBA sequence because that is indistinguishable from an array of values to be colormapped. c can be a 2-D array in which the rows are RGB or RGBA, however, including the case of a single row to specify the same color for all points.

marker : MarkerStyle, optional, default: ‘o’

See markers for more information on the different styles of markers scatter supports. marker can be either an instance of the class or the text shorthand for a particular marker.

cmap : Colormap, optional, default: None

A Colormap instance or registered name. cmap is only used if c is an array of floats. If None, defaults to rc image.cmap.

norm : Normalize, optional, default: None

A Normalize instance is used to scale luminance data to 0, 1. norm is only used if c is an array of floats. If None, use the default normalize().

vmin, vmax : scalar, optional, default: None

vmin and vmax are used in conjunction with norm to normalize luminance data. If either are None, the min and max of the color array is used. Note if you pass a norm instance, your settings for vmin and vmax will be ignored.

alpha : scalar, optional, default: None

The alpha blending value, between 0 (transparent) and 1 (opaque)

linewidths : scalar or array_like, optional, default: None

If None, defaults to (lines.linewidth,).

verts : sequence of (x, y), optional
If `marker` is None, these vertices will be used to construct the marker. The center of the marker is located at (0,0) in normalized units. The overall marker is rescaled by `s`.

`edgecolors` : color or sequence of color, optional, default: None
If None, defaults to ‘face’
If ‘face’, the edge color will always be the same as the face color.
If it is ‘none’, the patch boundary will not be drawn.
For non-filled markers, the `edgecolors` kwarg is ignored and forced to ‘face’ internally.

Returns `paths` : `PathCollection`

Other Parameters `kwargs` : `Collection` properties

See also:

`plot` to plot scatter plots when markers are identical in size and color

Notes

• The `plot` function will be faster for scatterplots where markers don’t vary in size or color.

• Any or all of `x`, `y`, `s`, and `c` may be masked arrays, in which case all masks will be combined and only unmasked points will be plotted.

Fundamentally, scatter works with 1-D arrays; `x`, `y`, `s`, and `c` may be input as 2-D arrays, but within scatter they will be flattened. The exception is `c`, which will be flattened only if its size matches the size of `x` and `y`. 
Note: In addition to the above described arguments, this function can take a data keyword argument. If such a data argument is given, the following arguments are replaced by data[<arg>]:


**matplotlib.axes.Axes.plot_date**

Axes.plot_date(x, y, fmt=’o’, tz=None, xdate=True, ydate=False, **kwargs)

A plot with data that contains dates.

Similar to the plot() command, except the x or y (or both) data is considered to be dates, and the axis is labeled accordingly.

x and/or y can be a sequence of dates represented as float days since 0001-01-01 UTC.

Note if you are using custom date tickers and formatters, it may be necessary to set the formatters/locators after the call to meth:plot_date since meth:plot_date will set the default tick locator to class:matplotlib.dates.AutoDateLocator (if the tick locator is not already set to a class:matplotlib.dates.DateLocator instance) and the default tick formatter
to class: :class:`matplotlib.dates.AutoDateFormatter` (if the tick formatter is not already set to a class: :class:`matplotlib.dates.DateFormatter` instance).

**Parameters**

- **fmt** : string
  The plot format string.

- **tz** : [None | timezone string | tzinfo instance]
  The time zone to use in labeling dates. If None, defaults to rc value.

- **xdate** : boolean
  If True, the x-axis will be labeled with dates.

- **ydate** : boolean
  If True, the y-axis will be labeled with dates.

**Returns**

- **lines**

**Other Parameters**

**kwargs** : :class:`matplotlib.lines.Line2D` properties :

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a :class:<code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[ (Path, Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>[‘butt’</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>[‘miter’</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>drawstyle</td>
<td>[‘default’</td>
</tr>
<tr>
<td>figure</td>
<td>a :class:<code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>[‘full’</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>[‘solid’</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>marker</td>
<td>A valid marker style</td>
</tr>
<tr>
<td>markeredgecolor or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
</tbody>
</table>
Table 19.4 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>[None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

.. note::

In addition to the above described arguments, this function can take a data keyword argument. If such a data argument is given, the following arguments are replaced by data[<arg>]:

• All arguments with the following names: ‘x’, ‘y’.

See also:

matplotlib.dates helper functions on dates

matplotlib.dates.date2num how to convert dates to num

matplotlib.dates.num2date how to convert num to dates

matplotlib.dates.drange how floating point dates

matplotlib.axes.Axes.step

Axes.step(x, y, *args, **kwargs)

Make a step plot.

Parameters x : array_like

1-D sequence, and it is assumed, but not checked, that it is uniformly increasing.

y : array_like
1-D sequence, and it is assumed, but not checked, that it is uniformly increasing.

Returns list

List of lines that were added.

Other Parameters where : [ ‘pre’ | ‘post’ | ‘mid’ ]

If ‘pre’ (the default), the interval from x[i] to x[i+1] has level y[i+1].
If ‘post’, that interval has level y[i].
If ‘mid’, the jumps in y occur half-way between the x-values.

Notes

Additional parameters are the same as those for plot().

Note: In addition to the above described arguments, this function can take a data keyword argument. If such a data argument is given, the following arguments are replaced by data[<arg>]:

• All arguments with the following names: ‘x’, ‘y’.

matplotlib.axes.Axes.loglog

Axes.loglog(*args, **kwargs)

Make a plot with log scaling on both the x and y axis.

loglog() supports all the keyword arguments of plot() and matplotlib.axes.Axes.set_xscale() / matplotlib.axes.Axes.set_yscale().

Notable keyword arguments:

baseX/baseY: scalar > 1 Base of the x/y logarithm

subX/subY: [ None | sequence ] The location of the minor x/y ticks; None defaults to autosubs, which depend on the number of decades in the plot; see matplotlib.axes.Axes.set_xscale() / matplotlib.axes.Axes.set_yscale() for details

nonposX/nonposY: [‘mask’ | ‘clip’ ] Non-positive values in x or y can be masked as invalid, or clipped to a very small positive number

The remaining valid kwargs are Line2D properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased</td>
<td>[True</td>
</tr>
</tbody>
</table>

569
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>drawstyle</td>
<td>['default'</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>['full'</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['solid'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>marker</td>
<td>A valid marker style</td>
</tr>
<tr>
<td>markeredgecolor or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>[None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

Example:
**Axes.semilogx**

Make a plot with log scaling on the x axis.

**Parameters**

- **baseX** : float, optional
  - Base of the x logarithm. The scalar should be larger than 1.

- **subX** : array_like, optional
  - The location of the minor xticks; None defaults to autosubs, which depend on the number of decades in the plot; see *set_xscale()* for details.

- **nonpos** : string, optional, {'mask', 'clip'}
  - Non-positive values in x can be masked as invalid, or clipped to a very small positive number.

**Returns**

- **plot**
  - Log-scaled plot on the x axis.

**Other Parameters**

- :class:`~matplotlib.lines.Line2D` properties:
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>drawstyle</td>
<td>['default'</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>['full'</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['solid'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>marker</td>
<td>A valid marker style</td>
</tr>
<tr>
<td>markeredgecolor or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>[None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

See also:
For example code and figure.

Notes

This function supports all the keyword arguments of `plot()` and `matplotlib.axes.Axes.set_xscale()`.

**matplotlib.axes.Axes.semilogy**

`Axes.semilogy(*args, **kwargs)`

Make a plot with log scaling on the y axis.

**Parameters**

- `basey` : scalar > 1
  
  Base of the y logarithm.

- `subsy` [None or iterable] The location of the minor yticks. None defaults to autosubs, which depend on the number of decades in the plot. See `set_yscale()` for details.

- `nonposy` [{'mask' | 'clip'} str] Non-positive values in y can be masked as invalid, or clipped to a very small positive number.

**Returns** `Line2D` Line instance of the plot.

**Other Parameters**


---

### Property Description

---

- `agg_filter <matplotlib.artist.Artist.set_agg_filter>`: unknown
- `alpha <matplotlib.artist.Artist.set_alpha>`: float (0.0 transparent through 1.0 opaque)
- `animated <matplotlib.artist.Artist.set_animated>`: [True | False]
- `antialiased <matplotlib.lines.Line2D.set_antialiased>`: [True | False]
```python
:meth:`clip_on <matplotlib.artist.Artist.set_clip_on>` [True | False]
:meth:`clip_path <matplotlib.artist.Artist.set_clip_path>` |
  ([:class:`~matplotlib.path.Path`, :class:`~matplotlib.transforms.Transform`) | |
  :class:`~matplotlib.patches.Patch` | None ]
:meth:`color <matplotlib.lines.Line2D.set_color>` or c any matplotlib color
:meth:`contains <matplotlib.artist.Artist.set_contains>` a callable function
:meth:`dash_capstyle <matplotlib.lines.Line2D.set_dash_capstyle>` ['butt' | |
  'round' | 'projecting']
:meth:`dash_joinstyle <matplotlib.lines.Line2D.set_dash_joinstyle>` ['miter' | |
  'round' | 'bevel']
:meth:`dashes <matplotlib.lines.Line2D.set_dashes>` a sequence of on/off ink in |
  points
:meth:`drawstyle <matplotlib.lines.Line2D.set_drawstyle>` ['default' | 'steps' | |
  'steps-pre' | 'steps-mid' | 'steps-post']
:meth:`figure <matplotlib.artist.Artist.set_figure>` a |
  :class:`~matplotlib.figure.Figure` instance
:meth:`fillstyle <matplotlib.lines.Line2D.set_fillstyle>` ['full' | 'left' | 'right' | |
  'bottom' | 'top' | 'none']
:meth:`gid <matplotlib.artist.Artist.set_gid>` an id string
:meth:`label <matplotlib.artist.Artist.set_label>` string or anything printable |
  with '%s' conversion.
:meth:`linestyle <matplotlib.lines.Line2D.set_linestyle>` or ls ['solid' | 'dashed', |
  'dashdot', 'dotted' | (offset, on-off-dash-seq) | '...-' | '.--' | '.:' | |
  ':.' | Line2D | 'None' | ' ' | '-' | '---' |
  '-----------']
:meth:`linewidth <matplotlib.lines.Line2D.set_linewidth>` or lw float value in |
  points
  style <matplotlib.markers>`
:meth:`markeredgecolor <matplotlib.lines.Line2D.set_markeredgecolor>` or |
  mec any matplotlib color
:meth:`markeredgewidth <matplotlib.lines.Line2D.set_markeredgewidth>` or |
  mew float value in points
:meth:`markerfacecolor <matplotlib.lines.Line2D.set_markerfacecolor>` or mfc |
  any matplotlib color
:meth:`markerfacecoloralt <matplotlib.lines.Line2D.set_markerfacecoloralt>` |
  or mfcalt any matplotlib color
:meth:`markersize <matplotlib.lines.Line2D.set_markersize>` or ms float
```
:method:`markevery` <matplotlib.lines.Line2D.set_markevery> ` [None | int | length-2 tuple of int | slice | list/array of int | float | length-2 tuple of float]

:method:`path_effects` <matplotlib.artist.Artist.set_path_effects>` `unknown

:method:`picker` <matplotlib.lines.Line2D.set_picker>` `float distance in points or callable pick function "fn(artist, event)"

:method:`pickradius` <matplotlib.lines.Line2D.set_pickradius>` `float distance in points

:method:`rasterized` <matplotlib.artist.Artist.set_rasterized>` `[True | False | None]

:method:`sketch_params` <matplotlib.artist.Artist.set_sketch_params>` `unknown

:method:`snap` <matplotlib.artist.Artist.set_snap>` `unknown

:method:`solid_capstyle` <matplotlib.lines.Line2D.set_solid_capstyle>` `['butt' | 'round' | 'projecting']

:method:`solid_joinstyle` <matplotlib.lines.Line2D.set_solid_joinstyle>` `['miter' | 'round' | 'bevel']


:method:`url` <matplotlib.artist.Artist.set_url>` `a url string

:method:`visible` <matplotlib.artist.Artist.set_visible>` `[True | False]

:method:`xdata` <matplotlib.lines.Line2D.set_xdata>` `1D array

:method:`ydata` <matplotlib.lines.Line2D.set_ydata>` `1D array

:method:`zorder` <matplotlib.artist.Artist.set_zorder>` `any number

===================================================================================== 
|

See also:

**loglog()** For example code and figure.

```python
matplotlib.axes.Axes.fill_between
```

**Axes.fill_between** *(x, y1, y2=0, where=None, interpolate=False, step=None, **kwargs)*

Make filled polygons between two curves.

Create a *PolyCollection* filling the regions between *y1* and *y2* where *where==True*

**Parameters**

- **x** : array
  
  An N-length array of the x data

- **y1** : array
  
  An N-length array (or scalar) of the y data
y2 : array

An N-length array (or scalar) of the y data

where : array, optional

If None, default to fill between everywhere. If not None, it is an N-length
numpy boolean array and the fill will only happen over the regions where
where==True.

interpolate : bool, optional

If True, interpolate between the two lines to find the precise point of intersec-
tion. Otherwise, the start and end points of the filled region will only occur
on explicit values in the x array.

step : {'pre', 'post', 'mid'}, optional

If not None, fill with step logic.

See also:

In addition to the above described arguments, this function can take a data keyword argument. If
such a data argument is given, the following arguments are replaced by data[<arg>]: * All
arguments with the following names: ‘where’, ‘x’, ‘y1’, ‘y2’.

Notes

Additional Keyword args passed on to the PolyCollection.

kwargs control the Polygon properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or antialiaseds</td>
<td>Boolean or sequence of booleans</td>
</tr>
<tr>
<td>array</td>
<td>unknown</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clim</td>
<td>a length 2 sequence of floats</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>cmap</td>
<td>a colormap or registered colormap name</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or edgecolors</td>
<td>matplotlib color spec or sequence of specs</td>
</tr>
<tr>
<td>facecolor or facecolors</td>
<td>matplotlib color spec or sequence of specs</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
</tr>
<tr>
<td>----------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>hatch</td>
<td>[‘/’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or dashes or linestyles</td>
<td>['solid'</td>
</tr>
<tr>
<td>linewidth or linewdths or lw</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>offset_position</td>
<td>unknown</td>
</tr>
<tr>
<td>offsets</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>pickradius</td>
<td>unknown</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>urls</td>
<td>unknown</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>
Examples

Matplotlib, Release 2.0.2

Chapter 19. Axes class
fill between where

Now regions with $y_2 > 1$ are masked
Matplotlib, Release 2.0.2

**matplotlib.axes.Axes.fill_betweenx**

Axes.fill_betweenx(y, x1, x2=0, where=None, step=None, **kwargs)
Make filled polygons between two horizontal curves.

Create a PolyCollection filling the regions between x1 and x2 where where==True

**Parameters**

- **y**: array
  An N-length array of the y data

- **x1**: array
  An N-length array (or scalar) of the x data

- **x2**: array, optional
  An N-length array (or scalar) of the x data

- **where**: array, optional
  If None, default to fill between everywhere. If not None, it is a N length
  numpy boolean array and the fill will only happen over the regions where
  where==True

- **step**: {'pre', 'post', 'mid'}, optional
If not None, fill with step logic.

**See also:**

In addition to the above described arguments, this function can take a `data` keyword argument. If such a `data` argument is given, the following arguments are replaced by `data[<arg>]`: * All arguments with the following names: ‘where’, ‘x1’, ‘x2’, ‘y’.

**Notes**

**keyword args passed on to the** *PolyCollection*

kwargs control the *Polygon* properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>agg_filter</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>alpha</code></td>
<td>float or None</td>
</tr>
<tr>
<td><code>animated</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>antialiased</code> or antialiaseds</td>
<td>Boolean or sequence of booleans</td>
</tr>
<tr>
<td><code>array</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>axes</code></td>
<td>an <em>Axes</em> instance</td>
</tr>
<tr>
<td><code>clim</code></td>
<td>a length 2 sequence of floats</td>
</tr>
<tr>
<td><code>clip_box</code></td>
<td>a <em>matplotlib.transforms.Bbox</em> instance</td>
</tr>
<tr>
<td><code>clip_on</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>clip_path</code></td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td><code>cmap</code></td>
<td>a colormap or registered colormap name</td>
</tr>
<tr>
<td><code>color</code></td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td><code>contains</code></td>
<td>a callable function</td>
</tr>
<tr>
<td><code>edgecolor</code> or edgecolors</td>
<td>matplotlib color spec or sequence of specs</td>
</tr>
<tr>
<td><code>facecolor</code> or facecolors</td>
<td>matplotlib color spec or sequence of specs</td>
</tr>
<tr>
<td><code>figure</code></td>
<td>a <em>matplotlib.figure.Figure</em> instance</td>
</tr>
<tr>
<td><code>gid</code></td>
<td>an id string</td>
</tr>
<tr>
<td><code>hatch</code></td>
<td>[ ‘/’</td>
</tr>
<tr>
<td><code>label</code></td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td><code>linestyle</code> or dashes or linestyles</td>
<td>[‘solid’</td>
</tr>
<tr>
<td><code>linewidth</code> or linewidths or lw</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td><code>norm</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>offset_position</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>offsets</code></td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td><code>path_effects</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>picker</code></td>
<td>[None</td>
</tr>
<tr>
<td><code>pickradius</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>rasterized</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>sketch_params</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>snap</code></td>
<td>unknown</td>
</tr>
</tbody>
</table>
Table 19.8 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>transform</td>
<td><em>Transform</em> instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>urls</td>
<td>unknown</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

Examples

![Graphs](image-url)
Matplotlib, Release 2.0.2

**matplotlib.axes.Axes.bar**

*Axes.bar(left, height, width=0.8, bottom=None, **kwargs)*

Make a bar plot.

Make a bar plot with rectangles bounded by:

- **left**, **left + width**, **bottom**, **bottom + height** (left, right, bottom and top edges)

**Parameters**

- **left**: sequence of scalars
  the x coordinates of the left sides of the bars

- **height**: sequence of scalars
  the heights of the bars

- **width**: scalar or array-like, optional
  the width(s) of the bars default: 0.8

- **bottom**: scalar or array-like, optional
  the y coordinate(s) of the bars default: None

- **color**: scalar or array-like, optional

---

19.1. Plotting 583
the colors of the bar faces

**edgecolor** : scalar or array-like, optional
the colors of the bar edges

**linewidth** : scalar or array-like, optional
width of bar edge(s). If None, use default linewidth; If 0, don’t draw edges.
default: None

**tick_label** : string or array-like, optional
the tick labels of the bars default: None

**xerr** : scalar or array-like, optional
if not None, will be used to generate errorbar(s) on the bar chart default: None

**yerr** : scalar or array-like, optional
if not None, will be used to generate errorbar(s) on the bar chart default: None

**ecolor** : scalar or array-like, optional
specifies the color of errorbar(s) default: None

**capsize** : scalar, optional
determines the length in points of the error bar caps default: None, which will take the value from the errorbar.capsize rcParam.

**error_kw** : dict, optional
dictionary of kwargs to be passed to errorbar method. ecolor and capsize may be specified here rather than as independent kwargs.

**align** : {'center', 'edge'}, optional
If ‘edge’, aligns bars by their left edges (for vertical bars) and by their bottom edges (for horizontal bars). If ‘center’, interpret the left argument as the coordinates of the centers of the bars. To align on the align bars on the right edge pass a negative width.

**orientation** : {'vertical', 'horizontal'}, optional
The orientation of the bars.

**log** : boolean, optional
If true, sets the axis to be log scale. default: False

**Returns** bars : matplotlib.container.BarContainer
Container with all of the bars + errorbars

See also:

**barh** Plot a horizontal bar plot.
Notes

The optional arguments color, edgecolor, linewidth, xerr, and yerr can be either scalars or sequences of length equal to the number of bars. This enables you to use bar as the basis for stacked bar charts, or candlestick plots. Detail: xerr and yerr are passed directly to errorbar(), so they can also have shape 2xN for independent specification of lower and upper errors.

Other optional kwargs:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor</td>
<td>mpl color spec, None, ‘none’, or ‘auto’</td>
</tr>
<tr>
<td>facecolor</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>['/'</td>
</tr>
<tr>
<td>joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle</td>
<td>['solid'</td>
</tr>
<tr>
<td>linewidth</td>
<td>float or None for default</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

Examples

Example: A stacked bar chart.
Note: In addition to the above described arguments, this function can take a data keyword argument. If such a data argument is given, the following arguments are replaced by data[<arg>]:


matplotlib.axes.Axes.barh

Axes.barh(bottom, width, height=0.8, left=None, **kwargs)

Make a horizontal bar plot.

Make a horizontal bar plot with rectangles bounded by:

left, left + width, bottom, bottom + height (left, right, bottom and top edges)

bottom, width, height, and left can be either scalars or sequences

Parameters:
bottom : scalar or array-like
    the y coordinate(s) of the bars

width : scalar or array-like
    the width(s) of the bars
**height** : sequence of scalars, optional, default: 0.8
the heights of the bars

**left** : sequence of scalars
the x coordinates of the left sides of the bars

Returns **matplotlib.patches.Rectangle** instances.

Other Parameters **color** : scalar or array-like, optional
the colors of the bars

**edgecolor** : scalar or array-like, optional
the colors of the bar edges

**linewidth** : scalar or array-like, optional, default: None
width of bar edge(s). If None, use default linewidth; If 0, don’t draw edges.

**tick_label** : string or array-like, optional, default: None
the tick labels of the bars

**xerr** : scalar or array-like, optional, default: None
if not None, will be used to generate errorbar(s) on the bar chart

**yerr** : scalar or array-like, optional, default: None
if not None, will be used to generate errorbar(s) on the bar chart

**ecolor** : scalar or array-like, optional, default: None
specifies the color of errorbar(s)

**capsize** : scalar, optional
determines the length in points of the error bar caps default: None, which will take the value from the errorbar.capsize rcParam.

**error_kw** :
dictionary of kwargs to be passed to errorbar method. ecolor and capsize may be specified here rather than as independent kwargs.

**align** : {'center', 'edge'}, optional
If 'edge', aligns bars by their left edges (for vertical bars) and by their bottom edges (for horizontal bars). If 'center', interpret the bottom argument as the coordinates of the centers of the bars. To align on the align bars on the top edge pass a negative 'height'.

**log** : boolean, optional, default: False
If true, sets the axis to be log scale

See also:
**bar** Plot a vertical bar plot.

**Notes**

The optional arguments `color`, `edgecolor`, `linewidth`, `xerr`, and `yerr` can be either scalars or sequences of length equal to the number of bars. This enables you to use bar as the basis for stacked bar charts, or candlestick plots. Detail: `xerr` and `yerr` are passed directly to `errorbar()`, so they can also have shape 2xN for independent specification of lower and upper errors.

Other optional kwargs:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>agg_filter</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>alpha</code></td>
<td>float or None</td>
</tr>
<tr>
<td><code>animated</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>antialiased</code> or <code>aa</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>axes</code></td>
<td>an <code>Axes</code> instance</td>
</tr>
<tr>
<td><code>capstyle</code></td>
<td>['butt'</td>
</tr>
<tr>
<td><code>clip_box</code></td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td><code>clip_on</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>clip_path</code></td>
<td>[(<code>Path</code>, <code>Transform</code>)</td>
</tr>
<tr>
<td><code>color</code></td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td><code>contains</code></td>
<td>a callable function</td>
</tr>
<tr>
<td><code>edgecolor</code> or <code>ec</code></td>
<td>mpl color spec, None, ‘none’, or ‘auto’</td>
</tr>
<tr>
<td><code>facecolor</code> or <code>fc</code></td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td><code>figure</code></td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td><code>fill</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>gid</code></td>
<td>an id string</td>
</tr>
<tr>
<td><code>hatch</code></td>
<td>[’/’</td>
</tr>
<tr>
<td><code>joinstyle</code></td>
<td>[’miter’</td>
</tr>
<tr>
<td><code>label</code></td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td><code>linestyle</code> or <code>ls</code></td>
<td>[’solid’</td>
</tr>
<tr>
<td><code>linewidth</code> or <code>lw</code></td>
<td>float or None for default</td>
</tr>
<tr>
<td><code>path_effects</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>picker</code></td>
<td>[None</td>
</tr>
<tr>
<td><code>rasterized</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>sketch_params</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>snap</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>transform</code></td>
<td><code>Transform</code> instance</td>
</tr>
<tr>
<td><code>url</code></td>
<td>a url string</td>
</tr>
<tr>
<td><code>visible</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>zorder</code></td>
<td>any number</td>
</tr>
</tbody>
</table>
Axes.stem(*args, **kwargs)
Create a stem plot.

Call signatures:

- `stem(y, linefmt='b-', markerfmt='bo', basefmt='r-')`
- `stem(x, y, linefmt='b-', markerfmt='bo', basefmt='r-')`

A stem plot plots vertical lines (using `linefmt`) at each x location from the baseline to y, and places a marker there using `markerfmt`. A horizontal line at 0 is plotted using `basefmt`.

If no x values are provided, the default is (0, 1, ..., len(y) - 1)

Return value is a tuple (`markerline`, `stemlines`, `baseline`).

See also:
This document for details.

Example:

![Stem plot example](image)

Note: In addition to the above described arguments, this function can take a `data` keyword argument. If such a `data` argument is given, the following arguments are replaced by `data[{<arg}>]:`
matplotlib.axes.Axes.eventplot

Axes.eventplot(positions, orientation='horizontal', lineoffsets=1, linelengths=1, linewidths=None, colors=None, linestyles='solid', **kwargs)

Plot identical parallel lines at specific positions.

Plot parallel lines at the given positions. positions should be a 1D or 2D array-like object, with each row corresponding to a row or column of lines.

This type of plot is commonly used in neuroscience for representing neural events, where it is commonly called a spike raster, dot raster, or raster plot.

However, it is useful in any situation where you wish to show the timing or position of multiple sets of discrete events, such as the arrival times of people to a business on each day of the month or the date of hurricanes each year of the last century.

orientation [ [ 'horizontal' | 'vertical' ] ] 'horizontal' : the lines will be vertical and arranged in rows
‘vertical’ : lines will be horizontal and arranged in columns

lineoffsets : A float or array-like containing floats.
linelengths : A float or array-like containing floats.
linewidths : A float or array-like containing floats.

colors must be a sequence of RGBA tuples (e.g., arbitrary color strings, etc, not allowed) or a list of such sequences

linestyles : [ 'solid' | 'dashed' | 'dashdot' | 'dotted' ] or an array of these values

For linelengths, linewidths, colors, and linestyles, if only a single value is given, that value is applied to all lines. If an array-like is given, it must have the same length as positions, and each value will be applied to the corresponding row or column in positions.

Returns a list of matplotlib.collections.EventCollection objects that were added.

kwargs are LineCollection properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased</td>
<td>Boolean or sequence of booleans</td>
</tr>
<tr>
<td>array</td>
<td>unknown</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clim</td>
<td>a length 2 sequence of floats</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
</tbody>
</table>

Continued on next page
Table 19.11 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cmap</code></td>
<td>a colormap or registered colormap name</td>
</tr>
<tr>
<td><code>color</code></td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td><code>contains</code></td>
<td>a callable function</td>
</tr>
<tr>
<td><code>edgecolor</code> or <code>edgecolors</code></td>
<td>matplotlib color spec or sequence of specs</td>
</tr>
<tr>
<td><code>facecolor</code> or <code>facecolors</code></td>
<td>matplotlib color spec or sequence of specs</td>
</tr>
<tr>
<td><code>figure</code></td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td><code>gid</code></td>
<td>an id string</td>
</tr>
<tr>
<td><code>hatch</code></td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td><code>label</code></td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td><code>linestyle</code> or <code>dashes</code> or <code>linestyles</code></td>
<td><code>['solid', 'dashed', 'dashdot', 'dotted']</code> or <code>(offset, on-off-dash-seq)</code></td>
</tr>
<tr>
<td><code>linewidth</code> or <code>linewdths</code> or <code>lw</code></td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td><code>norm</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>offset_position</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>offsets</code></td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td><code>path_effects</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>paths</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>picker</code></td>
<td>[None, float, boolean, callable]</td>
</tr>
<tr>
<td><code>pickradius</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>rasterized</code></td>
<td>[True, False, None]</td>
</tr>
<tr>
<td><code>segments</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>sketch_params</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>snap</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>transform</code></td>
<td><code>Transform</code> instance</td>
</tr>
<tr>
<td><code>url</code></td>
<td>a url string</td>
</tr>
<tr>
<td><code>urls</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>verts</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>visible</code></td>
<td>[True, False]</td>
</tr>
<tr>
<td><code>zorder</code></td>
<td>any number</td>
</tr>
</tbody>
</table>

Example:
Note: In addition to the above described arguments, this function can take a data keyword argument. If such a data argument is given, the following arguments are replaced by data[arg]:


```
import matplotlib.pyplot as plt

fig, ax = plt.subplots(
    figsize=(6, 6),
    subplot_kw={'aspect': 'auto'}
)

pie = ax.pie([0.25, 0.3, 0.5],
             explode=(0.1, 0.2, 0.0),
             labels=['A', 'B', 'C'],
             colors=['red', 'green', 'blue'],
             autopct='%1.1f%%',
             pctdistance=0.6
)
```

The `autopct` parameter is used to control the autopct parameter, which is the text in the interior of each wedge. The `pctdistance` parameter is used to control the distance of the text from the center of the pie chart.

Parameters:

- **x**: array-like
  - The input array used to make the pie chart.
- **explode**: array-like, optional, default: None
  - The input array used to make the pie chart.
- **labels**: array-like, optional, default: None
  - The input array used to make the pie chart.
- **colors**: array-like, optional, default: None
  - The input array used to make the pie chart.
- **autopct**: string, optional, default: None
  - The input array used to make the pie chart.
- **pctdistance**: float, optional, default: 0.6
  - The input array used to make the pie chart.
- **shadow**: bool, optional, default: False
  - The input array used to make the pie chart.
- **labeldistance**: float, optional, default: 1.1
  - The input array used to make the pie chart.
- **startangle**: float, optional, default: None
  - The input array used to make the pie chart.
- **radius**: float, optional, default: None
  - The input array used to make the pie chart.
- **counterclock**: bool, optional, default: True
  - The input array used to make the pie chart.
- **wedgeprops**: dict, optional, default: None
  - The input array used to make the pie chart.
- **textprops**: dict, optional, default: None
  - The input array used to make the pie chart.
- **center**: tuple, optional, default: (0, 0)
  - The input array used to make the pie chart.
- **frame**: bool, optional, default: False
  - The input array used to make the pie chart.

Plot a pie chart of array `x`. The fractional area of each wedge is given by `x/sum(x)`. If `sum(x) <= 1`, then the values of `x` give the fractional area directly and the array will not be normalized. The wedges are plotted counterclockwise, by default starting from the x-axis.
If not *None*, is a \(\text{len}(x)\) array which specifies the fraction of the radius with which to offset each wedge.

**labels** : list, optional, default: *None*

A sequence of strings providing the labels for each wedge

**colors** : array-like, optional, default: *None*

A sequence of matplotlib color args through which the pie chart will cycle. If *None*, will use the colors in the currently active cycle.

**autopct** : *None* (default), string, or function, optional

If not *None*, is a string or function used to label the wedges with their numeric value. The label will be placed inside the wedge. If it is a format string, the label will be \text{fmt}\%\text{pct}. If it is a function, it will be called.

**pctdistance** : float, optional, default: 0.6

The ratio between the center of each pie slice and the start of the text generated by *autopct*. Ignored if *autopct* is *None*.

**shadow** : bool, optional, default: False

Draw a shadow beneath the pie.

**labeldistance** : float, optional, default: 1.1

The radial distance at which the pie labels are drawn

**startangle** : float, optional, default: *None*

If not *None*, rotates the start of the pie chart by angle degrees counterclockwise from the x-axis.

**radius** : float, optional, default: *None*

The radius of the pie, if *radius* is *None* it will be set to 1.

**counterclock** : bool, optional, default: True

Specify fractions direction, clockwise or counterclockwise.

**wedgeprops** : dict, optional, default: *None*

Dict of arguments passed to the wedge objects making the pie. For example, you can pass in``‘wedgeprops = {‘linewidth’: 3}‘‘ to set the width of the wedge border lines equal to 3. For more details, look at the doc/arguments of the wedge object. By default clip_on=False.

**textprops** : dict, optional, default: *None*

Dict of arguments to pass to the text objects.

**center** : list of float, optional, default: (0, 0)

Center position of the chart. Takes value (0, 0) or is a sequence of 2 scalars.

**frame** : bool, optional, default: False
Plot axes frame with the chart if true.

**Returns patches** : list

A sequence of `matplotlib.patches.Wedge` instances

**texts** : list

A is a list of the label `matplotlib.text.Text` instances.

**autotexts** : list

A is a list of `Text` instances for the numeric labels. Is returned only if parameter `autopct` is not `None`.

**Notes**

The pie chart will probably look best if the figure and axes are square, or the Axes aspect is equal.

**Examples**

```
Frogs 15.0%
Hogs 30.0%
Dogs 45.0%
Logs 10.0%
```

**Note:** In addition to the above described arguments, this function can take a `data` keyword argument. If such a `data` argument is given, the following arguments are replaced by `data[<arg>]`: 

---

594 Chapter 19. Axes class
• All arguments with the following names: ‘colors’, ‘explode’, ‘labels’, ‘x’.

matplotlib.axes.Axes.stackplot

Axes.stackplot(x, *args, **kwargs)

Draws a stacked area plot.

x : 1d array of dimension N

y [2d array of dimension MxN, OR any number 1d arrays each of dimension] 1xN. The data is assumed to be unstacked. Each of the following calls is legal:

\[
\begin{align*}
\text{stackplot}(x, y) & \quad \# \text{ where } y \text{ is } MxN \\
\text{stackplot}(x, y_1, y_2, y_3, y_4) & \quad \# \text{ where } y_1, y_2, y_3, y_4, \text{ are all } 1xNm
\end{align*}
\]

Keyword arguments:

baseline [[‘zero’, ‘sym’, ‘wiggle’, ‘weighted_wiggle’]] Method used to calculate the baseline. ‘zero’ is just a simple stacked plot. ‘sym’ is symmetric around zero and is sometimes called ThemeRiver. ‘wiggle’ minimizes the sum of the squared slopes. ‘weighted_wiggle’ does the same but weights to account for size of each layer. It is also called Streamgraph-layout. More details can be found at http://leebyron.com/streamgraph/.

labels : A list or tuple of labels to assign to each data series.

colors [A list or tuple of colors. These will be cycled through and] used to colour the stacked areas.

All other keyword arguments are passed to fill_between()

Returns r : A list of PolyCollection, one for each element in the stacked area plot.

matplotlib.axes.Axes.broken_barh

Axes.broken_barh(xranges, yrange, **kwargs)

Plot horizontal bars.

A collection of horizontal bars spanning yrange with a sequence of xranges.

Required arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xranges</td>
<td>sequence of (xmin, xwidth)</td>
</tr>
<tr>
<td>yrange</td>
<td>sequence of (ymin, ywidth)</td>
</tr>
</tbody>
</table>

kwargs are matplotlib.collections.BrokenBarHCollection properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or antialiaseds</td>
<td>Boolean or sequence of booleans</td>
</tr>
</tbody>
</table>
Table 19.12 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>unknown</td>
</tr>
<tr>
<td>axes</td>
<td>an <em>Axes</em> instance</td>
</tr>
<tr>
<td>clim</td>
<td>a length 2 sequence of floats</td>
</tr>
<tr>
<td>clip_box</td>
<td>a <em>matplotlib.transforms.Bbox</em> instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>cmap</td>
<td>a colormap or registered colormap name</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or edgecolors</td>
<td>matplotlib color spec or sequence of specs</td>
</tr>
<tr>
<td>facecolor or facecolors</td>
<td>matplotlib color spec or sequence of specs</td>
</tr>
<tr>
<td>figure</td>
<td>a <em>matplotlib.figure.Figure</em> instance</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[ ‘/’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or dashes or linestyles</td>
<td>[‘solid’</td>
</tr>
<tr>
<td>linewidth or linewights or lw</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>norm</td>
<td>unknown</td>
</tr>
<tr>
<td>offset_position</td>
<td>unknown</td>
</tr>
<tr>
<td>offsets</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>pickradius</td>
<td>unknown</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td><em>Transform</em> instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>urls</td>
<td>unknown</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

these can either be a single argument, i.e.:

```python
facecolors = 'black'
```

or a sequence of arguments for the various bars, i.e.,:

```python
facecolors = ('black', 'red', 'green')
```

Example:
**Note:** In addition to the above described arguments, this function can take a `data` keyword argument. If such a `data` argument is given, the following arguments are replaced by `data[<arg>]`:

- All positional and all keyword arguments.

---

**matplotlib.axes.Axes.vlines**

```python
Axes.vlines(x, ymin, ymax, colors='k', linestyles='solid', label='', **kwargs)
```

Plot vertical lines. Plot vertical lines at each `x` from `ymin` to `ymax`.

**Parameters**

- `x` : scalar or 1D array_like
  x-indexes where to plot the lines.
- `ymin, ymax` : scalar or 1D array_like
  Respective beginning and end of each line. If scalars are provided, all lines will have same length.
- `colors` : array_like of colors, optional, default: ‘k’
- `linestyles` : [‘solid’ | ‘dashed’ | ‘dashdot’ | ‘dotted’], optional
**label**: string, optional, default: ‘’

**Returns** lines: *LineCollection*

**Other Parameters** **kwargs**: *LineCollection* properties.

**See also**:

*hlines* horizontal lines

**Examples**

```
0 1 2 3 4 5
time (s)
0.0
0.2
0.4
0.6
0.8
1.0
```

**Vertical lines demo**

```
0 1 2 3 4 5
time (s)
0.0
0.2
0.4
0.6
0.8
1.0
```

**Horizontal lines demo**

**Note:** In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, the following arguments are replaced by *data*[<arg>]:

- All arguments with the following names: ‘colors’, ‘x’, ‘ymax’, ‘ymin’.

**matplotlib.axes.Axes.hlines**

*Axes.hlines*(y, xmin, xmax, colors='k', linestyles='solid', label=' ', **kwargs)

Plot horizontal lines at each y from xmin to xmax.

**Parameters** **y**: scalar or sequence of scalar

y-indexes where to plot the lines.

**xmin, xmax**: scalar or 1D array_like

Respective beginning and end of each line. If scalars are provided, all lines will have same length.
colors: array_like of colors, optional, default: ‘k’

linestyles: ['solid' | 'dashed' | 'dashdot' | 'dotted'], optional

label: string, optional, default: ‘’

Returns lines: LineCollection

Other Parameters kwargs: LineCollection properties.

See also:

vlines vertical lines

Examples

Note: In addition to the above described arguments, this function can take a data keyword argument. If such a data argument is given, the following arguments are replaced by data[<arg>]:

- All arguments with the following names: ‘xmax’, ‘xmin’, ‘y’.

matplotlib.axes.Axes.fill

Axes.fill(*args, **kwargs)

Plot filled polygons.

Parameters args: a variable length argument

It allowing for multiple x, y pairs with an optional color format string; see plot() for details on the argument parsing. For example, each of the follow-
ing is legal:

19.1. Plotting
ax.fill(x, y)
ax.fill(x, y, "b")
ax.fill(x, y, "b", x, y, "r")

An arbitrary number of \(x, y\), color groups can be specified: ax.fill(x1, y1, ‘g’, x2, y2, ‘r’)

**Returns** a list of *Patch*

**Other Parameters** `**kwargs` : *Polygon* properties

**Notes**

The same color strings that *plot()* supports are supported by the fill format string.

If you would like to fill below a curve, e.g., shade a region between 0 and \(y\) along \(x\), use *fill_between()*

**Examples**

![Graph showing an example of fill](image)

**Note:** In addition to the above described arguments, this function can take a data keyword argument.
If such a data argument is given, the following arguments are replaced by data[<arg>]:

- All arguments with the following names: ‘x’, ‘y’.

### 19.1.2 Spans

<table>
<thead>
<tr>
<th>Axes.axhline</th>
<th>Add a horizontal line across the axis.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Axes.axhspan</td>
<td>Add a horizontal span (rectangle) across the axis.</td>
</tr>
<tr>
<td>Axes.axvline</td>
<td>Add a vertical line across the axes.</td>
</tr>
<tr>
<td>Axes.axvspan</td>
<td>Add a vertical span (rectangle) across the axes.</td>
</tr>
</tbody>
</table>

**matplotlib.axes.Axes.axhline**

Axes.axhline(y=0, xmin=0, xmax=1, **kwargs)

Add a horizontal line across the axis.

**Parameters**

- **y** : scalar, optional, default: 0
  
  y position in data coordinates of the horizontal line.

- **xmin** : scalar, optional, default: 0
  
  Should be between 0 and 1, 0 being the far left of the plot, 1 the far right of the plot.

- **xmax** : scalar, optional, default: 1
  
  Should be between 0 and 1, 0 being the far left of the plot, 1 the far right of the plot.

**Returns**

`Line2D`

**See also:**

- `axhspan` for example plot and source code

**Notes**

kwargs are passed to `Line2D` and can be used to control the line properties.

**Examples**

- draw a thick red hline at ‘y’ = 0 that spans the xrange:

  ```
  >>> axhline(linewidth=4, color='r')
  ```

- draw a default hline at ‘y’ = 1 that spans the xrange:
Matplotlib, Release 2.0.2

```python
>>> axhline(y=1)

• draw a default hline at ‘y’ = .5 that spans the middle half of the xrange:

```python
>>> axhline(y=.5, xmin=0.25, xmax=0.75)
```

Valid kwargs are Line2D properties, with the exception of ‘transform’:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>[‘butt’</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>[‘miter’</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>drawstyle</td>
<td>[‘default’</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>[‘full’</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>[‘solid’</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>marker</td>
<td>A valid marker style</td>
</tr>
<tr>
<td>markeredgecolor or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>[None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>[‘butt’</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>[‘miter’</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
</tbody>
</table>
### Table 19.14 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>url</code></td>
<td>a url string</td>
</tr>
<tr>
<td><code>visible</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>xdata</code></td>
<td>1D array</td>
</tr>
<tr>
<td><code>ydata</code></td>
<td>1D array</td>
</tr>
<tr>
<td><code>zorder</code></td>
<td>any number</td>
</tr>
</tbody>
</table>

---

**`matplotlib.axes.Axes.axhspan`**

**Axes.axhspan**(`ymin, ymax, xmin=0, xmax=1, **kwargs)``

Add a horizontal span (rectangle) across the axis.

Draw a horizontal span (rectangle) from `ymin` to `ymax`. With the default values of `xmin = 0` and `xmax = 1`, this always spans the xrange, regardless of the `xlim` settings, even if you change them, e.g., with the `set_xlim()` command. That is, the horizontal extent is in axes coords: 0 = left, 0.5 = middle, 1.0 = right but the `y` location is in data coordinates.

**Parameters**

- **`ymin`** : float
  
  Lower limit of the horizontal span in data units.

- **`ymax`** : float
  
  Upper limit of the horizontal span in data units.

- **`xmin`** : float, optional, default: 0
  
  Lower limit of the vertical span in axes (relative 0-1) units.

- **`xmax`** : float, optional, default: 1
  
  Upper limit of the vertical span in axes (relative 0-1) units.

**Returns**

- **`Polygon`**: `Polygon`

**Other Parameters**

- **`kwargs`**: `Polygon` properties.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>agg_filter</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>alpha</code></td>
<td>float or None</td>
</tr>
<tr>
<td><code>animated</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>antialiased</code> or <code>aa</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>axes</code></td>
<td>an <code>Axes</code> instance</td>
</tr>
<tr>
<td><code>capstyle</code></td>
<td>['butt'</td>
</tr>
<tr>
<td><code>clip_box</code></td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td><code>clip_on</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>clip_path</code></td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>color</strong></td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td><strong>contains</strong></td>
<td>a callable function</td>
</tr>
<tr>
<td><strong>edgecolor or ec</strong></td>
<td>mpl color spec, None, ‘none’, or ‘auto’</td>
</tr>
<tr>
<td><strong>facecolor or fc</strong></td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td><strong>figure</strong></td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td><strong>fill</strong></td>
<td>[True</td>
</tr>
<tr>
<td><strong>gid</strong></td>
<td>an id string</td>
</tr>
<tr>
<td><strong>hatch</strong></td>
<td>[‘/’</td>
</tr>
<tr>
<td><strong>joinstyle</strong></td>
<td>[‘miter’, ‘round’, ‘bevel’]</td>
</tr>
<tr>
<td><strong>label</strong></td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td><strong>linestyle or ls</strong></td>
<td>[‘solid’, ‘dashed’, ‘dashdot’, ‘dotted’</td>
</tr>
<tr>
<td><strong>linewidth or lw</strong></td>
<td>float or None for default</td>
</tr>
<tr>
<td><strong>path_effects</strong></td>
<td>unknown</td>
</tr>
<tr>
<td><strong>picker</strong></td>
<td>[None</td>
</tr>
<tr>
<td><strong>rasterized</strong></td>
<td>[True</td>
</tr>
<tr>
<td><strong>sketch_params</strong></td>
<td>unknown</td>
</tr>
<tr>
<td><strong>snap</strong></td>
<td>unknown</td>
</tr>
<tr>
<td><strong>transform</strong></td>
<td><code>Transform</code> instance</td>
</tr>
<tr>
<td><strong>url</strong></td>
<td>a url string</td>
</tr>
<tr>
<td><strong>visible</strong></td>
<td>[True</td>
</tr>
<tr>
<td><strong>zorder</strong></td>
<td>any number</td>
</tr>
</tbody>
</table>

**See also:**

`axvspan` Add a vertical span (rectangle) across the axes.
Examples

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()

# Add a vertical line across the axes.
ax.vlines(0.5, 0, 1, colors='r', linestyles='dashed')
ax.vlines(1.5, 0, 1, colors='g', linestyles='solid')
ax.vlines(0.5, 0, 0.5, colors='b')
ax.vlines(1.5, 0, 0.5, colors='b', linestyles='dotted')

plt.show()
```

**matplotlib.axes.Axes.axvline**

```
Axes.axvline(x=0, ymin=0, ymax=1, **kwargs)
```

Add a vertical line across the axes.

**Parameters**

- **x**: scalar, optional, default: 0
  
  x position in data coordinates of the vertical line.

- **ymin**: scalar, optional, default: 0

  Should be between 0 and 1, 0 being the bottom of the plot, 1 the top of the plot.

- **ymax**: scalar, optional, default: 1

  Should be between 0 and 1, 0 being the bottom of the plot, 1 the top of the plot.

**Returns**

- **Line2D**

See also:
**axhspan** for example plot and source code

**Examples**

- draw a thick red vline at \( x = 0 \) that spans the yrang:
  ```python
  >>> axvline(linewidth=4, color='r')
  ```

- draw a default vline at \( x = 1 \) that spans the yrang:
  ```python
  >>> axvline(x=1)
  ```

- draw a default vline at \( x = 0.5 \) that spans the middle half of the yrang:
  ```python
  >>> axvline(x=.5, ymin=0.25, ymax=0.75)
  ```

Valid kwargs are *Line2D* properties, with the exception of ‘transform’:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an <em>Axes</em> instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a <em>matplotlib.transforms.Bbox</em> instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any <em>matplotlib</em> color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>drawstyle</td>
<td>['default'</td>
</tr>
<tr>
<td>figure</td>
<td>a <em>matplotlib.figure.Figure</em> instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>['full'</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['solid'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>marker</td>
<td>A valid marker style</td>
</tr>
<tr>
<td>markeredgcolor or mec</td>
<td>any <em>matplotlib</em> color</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any <em>matplotlib</em> color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any <em>matplotlib</em> color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
</tbody>
</table>
Table 19.16 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>markevery</td>
<td>[None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

matplotlib.axes.Axes.axvspan

**Axes.axvspan**(xmin, xmax, ymin=0, ymax=1, **kwargs)

Add a vertical span (rectangle) across the axes.

Draw a vertical span (rectangle) from xmin to xmax. With the default values of ymin = 0 and ymax = 1. This always spans the yrange, regardless of the ylim settings, even if you change them, e.g., with the set_ylim() command. That is, the vertical extent is in axes coords: 0 = bottom, 0.5 = middle, 1.0 = top but the y location is in data coordinates.

**Parameters**

xmin : scalar

Number indicating the first X-axis coordinate of the vertical span rectangle in data units.

xmax : scalar

Number indicating the second X-axis coordinate of the vertical span rectangle in data units.

ymin : scalar, optional

Number indicating the first Y-axis coordinate of the vertical span rectangle in relative Y-axis units (0-1). Default to 0.

ymax : scalar, optional

Number indicating the second Y-axis coordinate of the vertical span rectangle in relative Y-axis units (0-1). Default to 1.

**Returns**

rectangle : matplotlib.patches.Polygon
Vertical span (rectangle) from (xmin, ymin) to (xmax, ymax).

**Other Parameters **

Optional parameters are properties of the class matplotlib.patches.Polygon.

See also:

axhspan

Examples

Draw a vertical, green, translucent rectangle from x = 1.25 to x = 1.55 that spans the yrange of the axes.

```python
>>> axvspan(1.25, 1.55, facecolor='g', alpha=0.5)
```

### 19.1.3 Spectral

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Axes.acorr</code></td>
<td>Plot the autocorrelation of x.</td>
</tr>
<tr>
<td><code>Axes.angle_spectrum</code></td>
<td>Plot the angle spectrum.</td>
</tr>
<tr>
<td><code>Axes.cohere</code></td>
<td>Plot the coherence between x and y.</td>
</tr>
<tr>
<td><code>Axes.csd</code></td>
<td>Plot the cross-spectral density.</td>
</tr>
<tr>
<td><code>Axes.magnitude_spectrum</code></td>
<td>Plot the magnitude spectrum.</td>
</tr>
<tr>
<td><code>Axes.phase_spectrum</code></td>
<td>Plot the phase spectrum.</td>
</tr>
<tr>
<td><code>Axes.psd</code></td>
<td>Plot the power spectral density.</td>
</tr>
<tr>
<td><code>Axes.specgram</code></td>
<td>Plot a spectrogram.</td>
</tr>
<tr>
<td><code>Axes.xcorr</code></td>
<td>Plot the cross correlation between x and y.</td>
</tr>
</tbody>
</table>

**matplotlib.axes.Axes.acorr**

`Axes.acorr(x, **kwargs)`  
Plot the autocorrelation of x.

**Parameters**

- `x` : sequence of scalar
- `hold` : boolean, optional, deprecated, default: True
- `detrend` : callable, optional, default: mlab.detrend_none  
  x is detrended by the detrend callable. Default is no normalization.
- `normed` : boolean, optional, default: True
  if True, input vectors are normalised to unit length.
- `usevlines` : boolean, optional, default: True
  if True, Axes.vlines is used to plot the vertical lines from the origin to the acorr. Otherwise, Axes.plot is used.
**maxlags**: integer, optional, default: 10

number of lags to show. If None, will return all 2 * \( \text{len}(x) \) - 1 lags.

**Returns** 
\((\text{lags}, c, \text{line}, b)\) where:

- **lags** are a length 2\(^{\text{maxlags}}\)+1 lag vector.
- **c** is the 2\(^{\text{maxlags}}\)+1 auto correlation vector.
- **line** is a `Line2D` instance returned by `plot`.
- **b** is the x-axis.

**Other Parameters**

**linestyle**: `Line2D` prop, optional, default: None

Only used if `usevlines` is False.

**marker**: string, optional, default: ‘o’

**Notes**

The cross correlation is performed with `numpy.correlate()` with mode = 2.

**Examples**

`xcorr` is top graph, and `acorr` is bottom graph.
Note: In addition to the above described arguments, this function can take a data keyword argument. If such a data argument is given, the following arguments are replaced by data[arg]:

- All arguments with the following names: 'x'.

```
matplotlib.axes.Axes.angle_spectrum
```

Axes.angle_spectrum(x, Fs=None, Fc=None, window=None, pad_to=None, sides=None, **kwargs)

Plot the angle spectrum.

Call signature:

```
angle_spectrum(x, Fs=2, Fc=0, window=mlab.window_hanning,
               pad_to=None, sides='default', **kwargs)
```

Compute the angle spectrum (wrapped phase spectrum) of x. Data is padded to a length of pad_to and the windowing function window is applied to the signal.

**Parameters**

- **x** : 1-D array or sequence
  
  Array or sequence containing the data

- **Fs** : scalar
The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.

**window**: callable or ndarray

A function or a vector of length NFFT. To create window vectors see `window_hanning()`, `window_none()`, `numpy.blackman()`, `numpy.hamming()`, `numpy.bartlett()`, `scipy.signal()`, `scipy.signal.get_window()`, etc. The default is `window_hanning()`. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

**sides**: [‘default’ | ‘onesided’ | ‘twosided’]

Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. ‘onesided’ forces the return of a one-sided spectrum, while ‘twosided’ forces two-sided.

**pad_to**: integer

The number of points to which the data segment is padded when performing the FFT. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the n parameter in the call to `ff`t(). The default is None, which sets pad_to equal to the length of the input signal (i.e. no padding).

**Fc**: integer

The center frequency of x (defaults to 0), which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

**kwargs**:

Keyword arguments control the `Line2D` properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>agg_filter</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>alpha</code></td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td><code>animated</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>antialiased</code> or <code>aa</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>axes</code></td>
<td>an <code>Axes</code> instance</td>
</tr>
<tr>
<td><code>clip_box</code></td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td><code>clip_on</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>clip_path</code></td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td><code>color</code> or <code>c</code></td>
<td>any <code>matplotlib</code> color</td>
</tr>
<tr>
<td><code>contains</code></td>
<td>a callable function</td>
</tr>
<tr>
<td><code>dash_capstyle</code></td>
<td>['butt'</td>
</tr>
<tr>
<td><code>dash_joinstyle</code></td>
<td>['miter'</td>
</tr>
</tbody>
</table>
Table 19.18 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>drawstyle</td>
<td>['default'</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>['full'</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with '%s' conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['solid'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>marker</td>
<td>A valid marker style</td>
</tr>
<tr>
<td>markeredgewidth or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgecolor or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgewidth or mec</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>[None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

Returns spectrum : 1-D array

The values for the angle spectrum in radians (real valued)

freqs : 1-D array

The frequencies corresponding to the elements in spectrum

line : a Line2D instance

The line created by this function

See also:

magnitude_spectrum() angle_spectrum() plots the magnitudes of the corresponding frequencies.

phase_spectrum() phase_spectrum() plots the unwrapped version of this function.
specgram() can plot the angle spectrum of segments within the signal in a colormap. In addition to the above described arguments, this function can take a data keyword argument. If such a data argument is given, the following arguments are replaced by data[arg]: * All arguments with the following names: ‘x’.

Examples

$
\text{Axes.cohere}(x, y, \text{NFFT}=256, \text{Fs}=2, \text{Fc}=0, \text{detrend}=<\text{function detrend\_none}>, \text{window}=<\text{function window\_hanning}>, \text{noverlap}=0, \text{pad\_to}=\text{None}, \text{sides}='\text{default}', \text{scale\_by\_freq}=\text{None}, **\text{kwargs})$

Plot the coherence between $x$ and $y$. Coherence is the normalized cross spectral density:

$$C_{xy} = \frac{|P_{xy}|^2}{P_{xx}P_{yy}}$$  \hspace{1cm} (19.1)

**Parameters**

Fs : scalar

The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.
window : callable or ndarray

A function or a vector of length NFFT. To create window vectors see window_hanning(), window_none(), numpy.blackman(), numpy.hanning(), numpy.bartlett(), scipy.signal(), scipy.signal.get_window(), etc. The default is window_hanning(). If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

sides : [ 'default' | 'onesided' | 'twosided' ]

Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. ‘onesided’ forces the return of a one-sided spectrum, while ‘twosided’ forces two-sided.

pad_to : integer

The number of points to which the data segment is padded when performing the FFT. This can be different from NFFT, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the n parameter in the call to fft(). The default is None, which sets pad_to equal to NFFT

NFFT : integer

The number of data points used in each block for the FFT. A power 2 is most efficient. The default value is 256. This should NOT be used to get zero padding, or the scaling of the result will be incorrect. Use pad_to for this instead.

detrend : {'default', 'constant', 'mean', 'linear', 'none'} or callable

The function applied to each segment before fft-ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the detrend parameter is a vector, in matplotlib it is a function. The pylab module defines detrend_none(), detrend_mean(), and detrend_linear(), but you can use a custom function as well. You can also use a string to choose one of the functions. ‘default’, ‘constant’, and ‘mean’ call detrend_mean(). ‘linear’ calls detrend_linear(). ‘none’ calls detrend_none().

scale_by_freq : boolean, optional

Specifies whether the resulting density values should be scaled by the scaling frequency, which gives density in units of Hz^-1. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

noverlap : integer

The number of points of overlap between blocks. The default value is 0 (no overlap).

Fc : integer
The center frequency of \( x \) (defaults to 0), which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

**kwargs:

Keyword arguments control the Line2D properties of the coherence plot:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>(Path, Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>[‘butt’</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>[‘miter’</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>drawstyle</td>
<td>[‘default’</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>[‘full’</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>[‘solid’</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>marker</td>
<td>A valid marker style</td>
</tr>
<tr>
<td>markeredgecolor or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgedwidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>[None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>[‘butt’</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>[‘miter’</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
</tbody>
</table>

19.1. Plotting
Table 19.19 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

**Returns** The return value is a tuple \((C_{xy}, f)\), where \(f\) are the frequencies of the coherence vector.

kwargs are applied to the lines.

**References**


**Examples**

![Graphs showing coherence and frequency](image)

**Note:** In addition to the above described arguments, this function can take a `data` keyword argument. If such a `data` argument is given, the following arguments are replaced by `data[<arg>]`: 

616 Chapter 19. Axes class
matplotlib.axes.Axes.csd

Axes.csd(x, y, NFFT=None, Fs=None, Fc=None, detrend=None, window=None, noverlap=None, pad_to=None, sides=None, scale_by_freq=None, return_line=None, **kwargs)

Plot the cross-spectral density.

Call signature:

csd(x, y, NFFT=256, Fs=2, Fc=0, detrend=mlab.detrend_none,
window=mlab.window_hanning, noverlap=0, pad_to=None,
sides='default', scale_by_freq=None, return_line=None, **kwargs)

The cross spectral density $P_{xy}$ by Welch’s average periodogram method. The vectors $x$ and $y$ are divided into $NFFT$ length segments. Each segment is detrended by function `detrend` and windowed by function `window`. `noverlap` gives the length of the overlap between segments. The product of the direct FFTs of $x$ and $y$ are averaged over each segment to compute $P_{xy}$, with a scaling to correct for power loss due to windowing.

If len($x$) < $NFFT$ or len($y$) < $NFFT$, they will be zero padded to $NFFT$.

**Parameters** $x$, $y$ : 1-D arrays or sequences

Arrays or sequences containing the data

**Fs** : scalar

The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.

**window** : callable or ndarray

A function or a vector of length $NFFT$. To create window vectors see `window_hanning()`, `window_none()`, `numpy.blackman()`, `numpy.hanning()`, `numpy.bartlett()`, `scipy.signal()`, `scipy.signal.get_window()`, etc. The default is `window_hanning()`. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

**sides** : [‘default’ | ‘onesided’ | ‘twosided’]

Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. ‘onesided’ forces the return of a one-sided spectrum, while ‘twosided’ forces two-sided.

**pad_to** : integer

The number of points to which the data segment is padded when performing the FFT. This can be different from $NFFT$, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in
the plot, allowing for more detail. This corresponds to the \( n \) parameter in the call to \texttt{fft()}. The default is \texttt{None}, which sets \texttt{pad_to} equal to \texttt{NFFT}

\textbf{NFFT} : integer

The number of data points used in each block for the FFT. A power 2 is most efficient. The default value is 256. This should \texttt{NOT} be used to get zero padding, or the scaling of the result will be incorrect. Use \texttt{pad_to} for this instead.

\textbf{detrend} : \{'default', 'constant', 'mean', 'linear', 'none'} or callable

The function applied to each segment before \texttt{fft-ing}, designed to remove the mean or linear trend. Unlike in MATLAB, where the \texttt{detrend} parameter is a vector, in matplotlib is it a function. The \texttt{pylab} module defines \texttt{detrend_none()}, \texttt{detrend_mean()}, and \texttt{detrend_linear()}, but you can use a custom function as well. You can also use a string to choose one of the functions. ‘default’, ‘constant’, and ‘mean’ call \texttt{detrend_mean()}. ‘linear’ calls \texttt{detrend_linear()}. ‘none’ calls \texttt{detrend_none()}. 

\textbf{scale\_by\_freq} : boolean, optional

Specifies whether the resulting density values should be scaled by the scaling frequency, which gives density in units of Hz\(^{-1}\). This allows for integration over the returned frequency values. The default is \texttt{True} for MATLAB compatibility.

\textbf{noverlap} : integer

The number of points of overlap between segments. The default value is 0 (no overlap).

\textbf{Fc} : integer

The center frequency of \( x \) (defaults to 0), which offsets the \( x \) extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsamped to baseband.

\textbf{return\_line} : bool

Whether to include the line object plotted in the returned values. Default is \texttt{False}.

\textbf{**kwargs} :

Keyword arguments control the \texttt{Line2D} properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{agg filter}</td>
<td>unknown</td>
</tr>
<tr>
<td>\texttt{alpha}</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>\texttt{animated}</td>
<td>[True</td>
</tr>
<tr>
<td>\texttt{antialiased}</td>
<td>[True</td>
</tr>
<tr>
<td>\texttt{axes}</td>
<td>an \texttt{Axes} instance</td>
</tr>
</tbody>
</table>

Continued on next page

Chapter 19. Axes class
Table 19.20 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clip_box</td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>`[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>`[(Path.Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any <code>matplotlib</code> color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>`['butt'</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>`['miter'</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>drawstyle</td>
<td>`['default'</td>
</tr>
<tr>
<td>figure</td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>`['full'</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>`['solid'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>marker</td>
<td>A valid marker style</td>
</tr>
<tr>
<td>markeredgecolor or mec</td>
<td>any <code>matplotlib</code> color</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any <code>matplotlib</code> color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any <code>matplotlib</code> color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>`[None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function ( \text{fn(artist, event)} )</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>`[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>`['butt'</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>`['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a <code>matplotlib.transforms.Transform</code> instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>`[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

**Returns**

- \( \text{Pxy} \) : 1-D array
  - The values for the cross spectrum \( P\_\{xy\} \) before scaling (complex valued)

- \( \text{freqs} \) : 1-D array
  - The frequencies corresponding to the elements in \( P\_\{xy\} \)

- \( \text{line} \) : a `Line2D` instance
The line created by this function. Only returned if `return_line` is True.

See also:

`psd() psd()` is the equivalent to setting `y=x`.

In addition to the above described arguments, this function can take a `data` keyword argument. If such a `data` argument is given, the following arguments are replaced by `data[<arg>]`: * All arguments with the following names: ‘x’, ‘y’.

Notes

For plotting, the power is plotted as $10 \log_{10}(P_{xy})$ for decibels, though $P_{xy}$ itself is returned.

References


Examples
matplotlib.axes.Axes.magnitude_spectrum

**Axes.magnitude_spectrum** *(x, Fs=None, Fc=None, window=None, pad_to=None, sides=None, scale=None, **kwargs)*

Plot the magnitude spectrum.

Call signature:

```python
magnitude_spectrum(x, Fs=2, Fc=0, window=mlab.window_hanning, pad_to=None, sides='default', **kwargs)
```

Compute the magnitude spectrum of `x`. Data is padded to a length of `pad_to` and the windowing function `window` is applied to the signal.

**Parameters**

- **x**: 1-D array or sequence
  
  Array or sequence containing the data

- **Fs**: scalar
  
  The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, `freqs`, in cycles per time unit. The default value is 2.

- **window**: callable or ndarray
  
  A function or a vector of length `NFFT`. To create window vectors see `window_hanning()`, `window_none()`, `numpy.blackman()`, `numpy.hanning()`, `numpy.bartlett()`, `scipy.signal`, `scipy.signal.get_window()`, etc. The default is `window_hanning()`. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

- **sides**: ['default' | 'onesided' | 'twosided']
  
  Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. ‘onesided’ forces the return of a one-sided spectrum, while ‘twosided’ forces two-sided.

- **pad_to**: integer
  
  The number of points to which the data segment is padded when performing the FFT. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the `n` parameter in the call to `fft()`. The default is `None`, which sets `pad_to` equal to the length of the input signal (i.e. no padding).

- **scale**: ['default' | 'linear' | 'dB']
  
  The scaling of the values in the `spec`. ‘linear’ is no scaling. ‘dB’ returns the values in dB scale. When `mode` is ‘density’, this is dB power (10 * log10). Otherwise this is dB amplitude (20 * log10). ‘default’ is ‘linear’.

- **Fc**: integer
The center frequency of \( x \) (defaults to 0), which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

**kwargs :

Keyword arguments control the Line2D properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>[‘butt’</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>[‘miter’</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>drawstyle</td>
<td>[‘default’</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>[‘full’</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>[‘solid’</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>marker</td>
<td>A valid marker style</td>
</tr>
<tr>
<td>markeredgecolor or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>[None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>[‘butt’</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>[‘miter’</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
</tbody>
</table>
Table 19.21 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

**Returns spectrum**: 1-D array

The values for the magnitude spectrum before scaling (real valued)

**freqs**: 1-D array

The frequencies corresponding to the elements in `spectrum`

**line**: a `Line2D` instance

The line created by this function

**See also**:

- `psd()`: plots the power spectral density.
- `angle_spectrum()`: plots the angles of the corresponding frequencies.
- `phase_spectrum()`: plots the phase (unwrapped angle) of the corresponding frequencies.
- `specgram()`: can plot the magnitude spectrum of segments within the signal in a colormap.

In addition to the above-described arguments, this function can take a `data` keyword argument. If such a `data` argument is given, the following arguments are replaced by `data[<arg>]`: * All arguments with the following names: ‘x’.

---

19.1. Plotting
Examples

```python
matplotlib.axes.Axes.phase_spectrum

Axes.phase_spectrum(x, Fs=None, Fc=None, window=None, pad_to=None, sides='default', **kwargs)

Plot the phase spectrum.

Call signature:

    phase_spectrum(x, Fs=2, Fc=0, window=mlab.window_hanning,
                   pad_to=None, sides='default', **kwargs)

Compute the phase spectrum (unwrapped angle spectrum) of x. Data is padded to a length of pad_to
and the windowing function window is applied to the signal.

Parameters

- **x** : 1-D array or sequence
  Array or sequence containing the data

- **Fs** : scalar
  The sampling frequency (samples per time unit). It is used to calculate the
  Fourier frequencies, freqs, in cycles per time unit. The default value is 2.
```
**window**: callable or ndarray

A function or a vector of length `NFFT`. To create window vectors see `window_hanning()`, `window_none()`, `numpy.blackman()`, `numpy.hamming()`, `numpy.bartlett()`, `scipy.signal()`, `scipy.signal.get_window()`, etc. The default is `window_hanning()`. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

**sides**: ['default' | 'onesided' | 'twosided']

Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. ‘onesided’ forces the return of a one-sided spectrum, while ‘twosided’ forces two-sided.

**pad_to**: integer

The number of points to which the data segment is padded when performing the FFT. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the `n` parameter in the call to `fft()`. The default is None, which sets `pad_to` equal to the length of the input signal (i.e. no padding).

**Fc**: integer

The center frequency of `x` (defaults to 0), which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

**kwargs**:

Keyword arguments control the `Line2D` properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>agg_filter</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>alpha</code></td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td><code>animated</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>antialiased</code> or <code>aa</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>axes</code></td>
<td>an <code>Axes</code> instance</td>
</tr>
<tr>
<td><code>clip_box</code></td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td><code>clip_on</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>clip_path</code></td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td><code>color</code> or <code>c</code></td>
<td>any matplotlib color</td>
</tr>
<tr>
<td><code>contains</code></td>
<td>a callable function</td>
</tr>
<tr>
<td><code>dash_capstyle</code></td>
<td>['butt'</td>
</tr>
<tr>
<td><code>dash_joinstyle</code></td>
<td>['miter'</td>
</tr>
<tr>
<td><code>dashes</code></td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td><code>drawstyle</code></td>
<td>['default'</td>
</tr>
<tr>
<td><code>figure</code></td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
</tbody>
</table>
Table 19.22 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fillstyle</td>
<td>['full'</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['solid'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>marker</td>
<td>A valid marker style</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>[None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

**Returns spectrum**: 1-D array

The values for the phase spectrum in radians (real valued)

**freqs**: 1-D array

The frequencies corresponding to the elements in spectrum

**line**: a Line2D instance

The line created by this function

See also:

`magnitude_spectrum()` plots the magnitudes of the corresponding frequencies.

`angle_spectrum()` plots the wrapped version of this function.

`specgram()` can plot the phase spectrum of segments within the signal in a colormap.
In addition to the above described arguments, this function can take a `data` keyword argument. If such a `data` argument is given, the following arguments are replaced by `data[<arg>]`: * All arguments with the following names: `x`.

**Examples**

![Graph showing power spectral density](image)

**matplotlib.axes.Axes.psd**

`Axes.psd(x, NFFT=None, Fs=None, Fc=None, detrend=None, window=None, noverlap=None, pad_to=None, sides='default', scale_by_freq=None, return_line=None, **kwargs)`

Plot the power spectral density.

Call signature:

```python
psd(x, NFFT=256, Fs=2, Fc=0, detrend=mlab.detrend_none,
    window=mlab.window_hanning, noverlap=0, pad_to=None,
    sides='default', scale_by_freq=None, return_line=None, **kwargs)
```

The power spectral density $P_{xx}$ by Welch’s average periodogram method. The vector $x$ is divided into $NFFT$ length segments. Each segment is detrended by function `detrend` and windowed by function `window`. `noverlap` gives the length of the overlap between segments. The $|\text{fft}(i)|^2$ of each segment $i$ are averaged to compute $P_{xx}$, with a scaling to correct for power loss due to windowing.
If \( \text{len}(x) < NFFT \), it will be zero padded to \( NFFT \).

**Parameters**

- **x**: 1-D array or sequence
  Array or sequence containing the data

- **Fs**: scalar
  The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.

- **window**: callable or ndarray
  A function or a vector of length \( NFFT \). To create window vectors see `window_hanning()`, `window_none()`, `numpy.blackman()`, `numpy.hamming()`, `numpy.bartlett()`, `scipy.signal()`, `scipy.signal.get_window()`, etc. The default is `window_hanning()`. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

- **sides**: ['default' | 'onesided' | 'twosided']
  Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. ‘onesided’ forces the return of a one-sided spectrum, while ‘twosided’ forces two-sided.

- **pad_to**: integer
  The number of points to which the data segment is padded when performing the FFT. This can be different from \( NFFT \), which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the \( n \) parameter in the call to `fft()`. The default is None, which sets pad_to equal to \( NFFT \)

- **NFFT**: integer
  The number of data points used in each block for the FFT. A power 2 is most efficient. The default value is 256. This should NOT be used to get zero padding, or the scaling of the result will be incorrect. Use `pad_to` for this instead.

- **detrend**: {'default', 'constant', 'mean', 'linear', 'none'} or callable
  The function applied to each segment before `fft-ing`, designed to remove the mean or linear trend. Unlike in MATLAB, where the detrend parameter is a vector, in matplotlib is it a function. The `pylab` module defines `detrend_none()`, `detrend_mean()`, and `detrend_linear()`, but you can use a custom function as well. You can also use a string to choose one of the functions. ‘default’, ‘constant’, and ‘mean’ call `detrend_mean()`. ‘linear’ calls `detrend_linear()`. ‘none’ calls `detrend_none()`.

- **scale_by_freq**: boolean, optional

---

**Chapter 19. Axes class**
Specifies whether the resulting density values should be scaled by the scaling frequency, which gives density in units of Hz^-1. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

nooverlap : integer

The number of points of overlap between segments. The default value is 0 (no overlap).

Fc : integer

The center frequency of x (defaults to 0), which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

return_line : bool

Whether to include the line object plotted in the returned values. Default is False.

**kwargs :

Keyword arguments control the Line2D properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[ (Path, Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>drawstyle</td>
<td>['default'</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>['full'</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['solid'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>marker</td>
<td>A valid marker style</td>
</tr>
<tr>
<td>markeredgecolor or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
</tbody>
</table>
Table 19.23 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>[None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a <code>matplotlib.transforms.Transform</code> instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

**Returns**

\(P_{xx} \): 1-D array

The values for the power spectrum \(P_{xx}\) before scaling (real valued)

\(freqs\): 1-D array

The frequencies corresponding to the elements in \(Pxx\)

\(line\): a `Line2D` instance

The line created by this function. Only returned if `return_line` is True.

**See also:**

- `specgram() specgram()` differs in the default overlap; in not returning the mean of the segment periodograms; in returning the times of the segments; and in plotting a colormap instead of a line.

- `magnitude_spectrum() magnitude_spectrum()` plots the magnitude spectrum.

- `csd() csd()` plots the spectral density between two signals.

In addition to the above described arguments, this function can take a `data` keyword argument. If such a `data` argument is given, the following arguments are replaced by `data[<arg>]`: * All arguments with the following names: ‘x’.

**Notes**

For plotting, the power is plotted as \(10 \log_{10}(P_{xx})\) for decibels, though \(Pxx\) itself is returned.
References


Examples

```python
matplotlib.axes.Axes.specgram

Axes.specgram(x, NFFT=None, Fs=None, Fc=None, detrend=None, window=None,
    noverlap=None, cmap=None, xextent=None, pad_to=None, sides=None,
    scale_by_freq=None, mode=None, scale=None, vmin=None, vmax=None,
    **kwargs)

Plot a spectrogram.

Call signature:

specgram(x, NFFT=256, Fs=2, Fc=0, detrend=mlab.detrend_none,
    window=mlab.window_hanning, noverlap=128,
    cmap=none, xextent=none, pad_to=none, sides='default',
    scale_by_freq=none, mode='default', scale='default',
    **kwargs)
```

19.1. Plotting
Compute and plot a spectrogram of data in $x$. Data are split into $NFFT$ length segments and the spectrum of each section is computed. The windowing function $window$ is applied to each segment, and the amount of overlap of each segment is specified with $noverlap$. The spectrogram is plotted as a colormap (using imshow).

**Parameters**

- **$x$**: 1-D array or sequence
  - Array or sequence containing the data.
- **$Fs$**: scalar
  - The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.
- **$window$**: callable or ndarray
  - A function or a vector of length $NFFT$. To create window vectors see `window_hanning()`, `window_none()`, `numpy.blackman()`, `numpy.hamming()`, `numpy.bartlett()`, `scipy.signal()`, `scipy.signal.get_window()`, etc. The default is `window_hanning()`. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.
- **$sides$**: [‘default’ | ‘onesided’ | ‘twosided’ ]
  - Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. ‘onesided’ forces the return of a one-sided spectrum, while ‘twosided’ forces two-sided.
- **$pad_to$**: integer
  - The number of points to which the data segment is padded when performing the FFT. This can be different from $NFFT$, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the $n$ parameter in the call to $fft()$. The default is None, which sets $pad_to$ equal to $NFFT$.
- **$NFFT$**: integer
  - The number of data points used in each block for the FFT. A power 2 is most efficient. The default value is 256. This should NOT be used to get zero padding, or the scaling of the result will be incorrect. Use $pad_to$ for this instead.
- **$detrend$**: {‘default’, ‘constant’, ‘mean’, ‘linear’, ‘none’} or callable
  - The function applied to each segment before fft-ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the $detrend$ parameter is a vector, in matplotlib is it a function. The $pylab$ module defines $detrend_none()$, $detrend_mean()$, and $detrend_linear()$, but you can use a custom function as well. You can also use a string to choose one of the functions. ‘default’, ‘constant’, and ‘mean’ call $detrend_mean()$. ‘linear’ calls $detrend_linear()$. ‘none’ calls $detrend_none()$. 

632 Chapter 19. Axes class
scale_by_freq : boolean, optional

Specifies whether the resulting density values should be scaled by the scaling frequency, which gives density in units of Hz^-1. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

mode : [ ‘default’ | ‘psd’ | ‘magnitude’ | ‘angle’ | ‘phase’ ]

What sort of spectrum to use. Default is ‘psd’, which takes the power spectral density. ‘complex’ returns the complex-valued frequency spectrum. ‘magnitude’ returns the magnitude spectrum. ‘angle’ returns the phase spectrum without unwrapping. ‘phase’ returns the phase spectrum with unwrapping.

noverlap : integer

The number of points of overlap between blocks. The default value is 128.

scale : [ ‘default’ | ‘linear’ | ‘dB’ ]

The scaling of the values in the spectrum. ‘linear’ is no scaling. ‘dB’ returns the values in dB scale. When mode is ‘psd’, this is dB power (10 * log10). Otherwise this is dB amplitude (20 * log10). ‘default’ is ‘dB’ if mode is ‘psd’ or ‘magnitude’ and ‘linear’ otherwise. This must be ‘linear’ if mode is ‘angle’ or ‘phase’.

Fc : integer

The center frequency of x (defaults to 0), which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampling to baseband.

cmap :

A matplotlib.colors.Colormap instance; if None, use default determined by rc

xextent : [None | (xmin, xmax)]

The image extent along the x-axis. The default sets xmin to the left border of the first bin (spectrum column) and xmax to the right border of the last bin. Note that for noverlap>0 the width of the bins is smaller than those of the segments.

**kwargs :

Additional kwargs are passed on to imshow which makes the specgram image

Returns spectrum : 2-D array

Columns are the periodograms of successive segments.

freqs : 1-D array

The frequencies corresponding to the rows in spectrum.

t : 1-D array
The times corresponding to midpoints of segments (i.e., the columns in `spec-
trum`).

`im` : instance of class `AxesImage`

The image created by imshow containing the spectrogram

See also:

- `psd()` differs in the default overlap; in returning the mean of the segment periodograms; in not returning times; and in generating a line plot instead of colormap.
- `magnitude_spectrum()` A single spectrum, similar to having a single segment when `mode` is ‘magnitude’. Plots a line instead of a colormap.
- `angle_spectrum()` A single spectrum, similar to having a single segment when `mode` is ‘angle’. Plots a line instead of a colormap.
- `phase_spectrum()` A single spectrum, similar to having a single segment when `mode` is ‘phase’. Plots a line instead of a colormap.

In addition to the above described arguments, this function can take a `data` keyword argument. If such a `data` argument is given, the following arguments are replaced by `data[<arg>]`: *

| All arguments with the following names: ‘x’.

Notes

- `detrend` and `scale_by_freq` only apply when `mode` is set to ‘psd’
Examples

```python
matplotlib.axes.Axes.xcorr

Axes.xcorr(x, y, normed=True, detrend=<function detrend_none>, usevlines=True, maxlags=10, **kwargs)

Plot the cross correlation between x and y.

The correlation with lag k is defined as sum_n x[n+k] * conj(y[n]).

Parameters:

- **x** : sequence of scalars of length n
- **y** : sequence of scalars of length n
- **hold** : boolean, optional, deprecated, default: True
- **detrend** : callable, optional, default: mlab.detrend_none
  x is detrended by the detrend callable. Default is no normalization.
- **normed** : boolean, optional, default: True
  if True, input vectors are normalised to unit length.
- **usevlines** : boolean, optional, default: True
```
if True, Axes.vlines is used to plot the vertical lines from the origin to the acorr. Otherwise, Axes.plot is used.

**maxlags**: integer, optional, default: 10

number of lags to show. If None, will return all 2 * len(x) - 1 lags.

**Returns** *(lags, c, line, b)*: where:

• *lags* are a length 2*maxlags+1 lag vector.
• *c* is the 2*maxlags+1 auto correlation vector
• *line* is a *Line2D* instance returned by *plot*.
• *b* is the x-axis (none, if plot is used).

**Other Parameters**

**linestyle**: *Line2D* prop, optional, default: None

Only used if usevlines is False.

**marker**: string, optional, default: ‘o’

**Notes**

The cross correlation is performed with *numpy.correlate()* with *mode = 2*.

**Note**: In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, the following arguments are replaced by *data*[<arg>]:

• All arguments with the following names: ‘x’, ‘y’.

19.1.4 Statistics

| **Axes.boxplot** | Make a box and whisker plot. |
| **Axes.violinplot** | Make a violin plot. |
| **Axes.violin** | Drawing function for violin plots. |
| **Axes.bxp** | Drawing function for box and whisker plots. |

**matplotlib.axes.Axes.boxplot**

**Axes.boxplot**(x, notch=None, sym=None, vert=None, whis=None, positions=None, widths=None, patch_artist=None, bootstrap=None, usermedians=None, conf_intervals=None, meanline=None, showmeans=None, showcaps=None, showbox=None, showfliers=None, boxprops=None, labels=None, flierprops=None, medianprops=None, meanprops=None, capprops=None, whiskerprops=None, manage_xticks=True, autorange=False, zorder=None)

Make a box and whisker plot for each column of *x* or each vector in sequence *x*. The box extends
from the lower to upper quartile values of the data, with a line at the median. The whiskers extend from the box to show the range of the data. Flier points are those past the end of the whiskers.

**Parameters**

- **x**: Array or a sequence of vectors.
  
The input data.

- **notch**: bool, optional (False)
  
  If True, will produce a notched box plot. Otherwise, a rectangular boxplot is produced. The notches represent the confidence interval (CI) around the median. See the entry for the bootstrap parameter for information regarding how the locations of the notches are computed.

  **Note:** In cases where the values of the CI are less than the lower quartile or greater than the upper quartile, the notches will extend beyond the box, giving it a distinctive “flipped” appearance. This is expected behavior and consistent with other statistical visualization packages.

- **sym**: str, optional
  
  The default symbol for flier points. Enter an empty string ("") if you don’t want to show fliers. If None, then the fliers default to ‘b+’ If you want more control use the flierprops kwarg.

- **vert**: bool, optional (True)
  
  If True (default), makes the boxes vertical. If False, everything is drawn horizontally.

- **whis**: float, sequence, or string (default = 1.5)
  
  As a float, determines the reach of the whiskers to the beyond the first and third quartiles. In other words, where IQR is the interquartile range (Q3–Q1), the upper whisker will extend to last datum less than Q3 + whis*IQR). Similarly, the lower whisker will extend to the first datum greater than Q1 - whis*IQR. Beyond the whiskers, data are considered outliers and are plotted as individual points. Set this to an unreasonably high value to force the whiskers to show the min and max values. Alternatively, set this to an ascending sequence of percentile (e.g., [5, 95]) to set the whiskers at specific percentiles of the data. Finally, whis can be the string 'range' to force the whiskers to the min and max of the data.

- **bootstrap**: int, optional
  
  Specifies whether to bootstrap the confidence intervals around the median for notched boxplots. If bootstrap is None, no bootstrapping is performed, and notches are calculated using a Gaussian-based asymptotic approximation (see McGill, R., Tukey, J.W., and Larsen, W.A., 1978, and Kendall and Stuart, 1967). Otherwise, bootstrap specifies the number of times to bootstrap the median to determine its 95% confidence intervals. Values between 1000 and 10000 are recommended.
usermedians : array-like, optional

An array or sequence whose first dimension (or length) is compatible with x. This overrides the medians computed by matplotlib for each element of usermedians that is not None. When an element of usermedians is None, the median will be computed by matplotlib as normal.

conf_intervals : array-like, optional

Array or sequence whose first dimension (or length) is compatible with x and whose second dimension is 2. When the an element of conf_intervals is not None, the notch locations computed by matplotlib are overridden (provided notch is True). When an element of conf_intervals is None, the notches are computed by the method specified by the other kwargs (e.g., bootstrap).

positions : array-like, optional

Sets the positions of the boxes. The ticks and limits are automatically set to match the positions. Defaults to range(1, N+1) where N is the number of boxes to be drawn.

widths : scalar or array-like

Sets the width of each box either with a scalar or a sequence. The default is 0.5, or 0.15*(distance between extreme positions), if that is smaller.

patch_artist : bool, optional (False)

If False produces boxes with the Line2D artist. Otherwise, boxes and drawn with Patch artists.

labels : sequence, optional

Labels for each dataset. Length must be compatible with dimensions of x.

manage_xticks : bool, optional (True)

If the function should adjust the xlim and xtick locations.

autorange : bool, optional (False)

When True and the data are distributed such that the 25th and 75th percentiles are equal, whis is set to 'range' such that the whisker ends are at the minimum and maximum of the data.

meanline : bool, optional (False)

If True (and showmeans is True), will try to render the mean as a line spanning the full width of the box according to meanprops (see below). Not recommended if shownotches is also True. Otherwise, means will be shown as points.

zorder : scalar, optional (None)

Sets the zorder of the boxplot.
**Returns result**: dict

A dictionary mapping each component of the boxplot to a list of the `matplotlib.lines.Line2D` instances created. That dictionary has the following keys (assuming vertical boxplots):

- **boxes**: the main body of the boxplot showing the quartiles and the median’s confidence intervals if enabled.
- **medians**: horizontal lines at the median of each box.
- **whiskers**: the vertical lines extending to the most extreme, non-outlier data points.
- **caps**: the horizontal lines at the ends of the whiskers.
- **fliers**: points representing data that extend beyond the whiskers (fliers).
- **means**: points or lines representing the means.

**Other Parameters**

- **showcaps**: bool, optional (True)
  Show the caps on the ends of whiskers.
- **showbox**: bool, optional (True)
  Show the central box.
- **showfliers**: bool, optional (True)
  Show the outliers beyond the caps.
- **showmeans**: bool, optional (False)
  Show the arithmetic means.
- **capprops**: dict, optional (None)
  Specifies the style of the caps.
- **boxprops**: dict, optional (None)
  Specifies the style of the box.
- **whiskerprops**: dict, optional (None)
  Specifies the style of the whiskers.
- **flierprops**: dict, optional (None)
  Specifies the style of the fliers.
- **medianprops**: dict, optional (None)
  Specifies the style of the median.
- **meanprops**: dict, optional (None)
  Specifies the style of the mean.
Examples

Default

showmeans=True

meanline=True

Tufte Style
(showbox=False,
showcaps=False)

notch=True,
bootstrap=10000

showfliers=False
I never said they'd be pretty

Custom boxprops

Custom medianprops
and flierprops

whis="range"

Custom mean
as point

Custom mean
as line

whis=[15, 85]
#percentiles

Note: In addition to the above described arguments, this function can take a data keyword argument. If such a data argument is given, the following arguments are replaced by data[<arg>]:

- All positional and all keyword arguments.

matplotlib.axes.Axes.violinplot

Axes.violinplot(dataset, positions=None, vert=True, widths=0.5, showmeans=False, showextrema=True, showmedians=False, points=100, bw_method=None)

Make a violin plot.

Make a violin plot for each column of dataset or each vector in sequence dataset. Each filled area extends to represent the entire data range, with optional lines at the mean, the median, the minimum, and the maximum.
Parameters

dataset : Array or a sequence of vectors.

   The input data.

positions : array-like, default = [1, 2, ..., n]

   Sets the positions of the violins. The ticks and limits are automatically set to
   match the positions.

vert : bool, default = True.

   If true, creates a vertical violin plot. Otherwise, creates a horizontal violin
   plot.

widths : array-like, default = 0.5

   Either a scalar or a vector that sets the maximal width of each violin. The
   default is 0.5, which uses about half of the available horizontal space.

showmeans : bool, default = False

   If True, will toggle rendering of the means.

showextrema : bool, default = True

   If True, will toggle rendering of the extrema.

showmedians : bool, default = False

   If True, will toggle rendering of the medians.

points : scalar, default = 100

   Defines the number of points to evaluate each of the gaussian kernel density
   estimations at.

bw_method : str, scalar or callable, optional

   The method used to calculate the estimator bandwidth. This can be ‘scott’,
   ‘silverman’, a scalar constant or a callable. If a scalar, this will be used di-
   rectly as kde.factor. If a callable, it should take a GaussianKDE instance
   as its only parameter and return a scalar. If None (default), ‘scott’ is used.

Returns

result : dict

   A dictionary mapping each component of the violinplot to a list of the corre-
   sponding collection instances created. The dictionary has the following keys:

   • bodies : A list of the matplotlib.collections.PolyCollection in-
   stances containing the filled area of each violin.

   • cmeans : A matplotlib.collections.LineCollection instance cre-
   ated to identify the mean values of each of the violin’s distribution.

   • cmins : A matplotlib.collections.LineCollection instance cre-
   ated to identify the bottom of each violin’s distribution.

   • cmaxes : A matplotlib.collections.LineCollection instance cre-
   ated to identify the top of each violin’s distribution.
• `cbars`: A `matplotlib.collections.LineCollection` instance created to identify the centers of each violin’s distribution.

• `cmedians`: A `matplotlib.collections.LineCollection` instance created to identify the median values of each of the violin’s distribution.

**Note:** In addition to the above described arguments, this function can take a `data` keyword argument. If such a `data` argument is given, the following arguments are replaced by `data[<arg>]`:

- All arguments with the following names: ‘dataset’.

---

**matplotlib.axes.Axes.violin**

The `Axes.violin` function is a drawing function for violin plots.

Draw a violin plot for each column of `vpstats`. Each filled area extends to represent the entire data range, with optional lines at the mean, the median, the minimum, and the maximum.

**Parameters**

- **`vpstats`**: list of dicts
  
  A list of dictionaries containing stats for each violin plot. Required keys are:

  - `coords`: A list of scalars containing the coordinates that the violin’s kernel density estimate were evaluated at.
  
  - `vals`: A list of scalars containing the values of the kernel density estimate at each of the coordinates given in `coords`.
  
  - `mean`: The mean value for this violin’s dataset.
  
  - `median`: The median value for this violin’s dataset.
  
  - `min`: The minimum value for this violin’s dataset.
  
  - `max`: The maximum value for this violin’s dataset.

- **`positions`**: array-like, default = `[1, 2, ..., n]`
  
  Sets the positions of the violins. The ticks and limits are automatically set to match the positions.

- **`vert`**: bool, default = `True`.
  
  If true, plots the violins vertically. Otherwise, plots the violins horizontally.

- **`widths`**: array-like, default = `0.5`
  
  Either a scalar or a vector that sets the maximal width of each violin. The default is `0.5`, which uses about half of the available horizontal space.

- **`showmeans`**: bool, default = `False`
If true, will toggle rendering of the means.

**showextrema**: bool, default = True

If true, will toggle rendering of the extrema.

**showmedians**: bool, default = False

If true, will toggle rendering of the medians.

**Returns result**: dict

A dictionary mapping each component of the violinplot to a list of the corresponding collection instances created. The dictionary has the following keys:

- **bodies**: A list of the `matplotlib.collections.PolyCollection` instances containing the filled area of each violin.
- **cmeans**: A `matplotlib.collections.LineCollection` instance created to identify the mean values of each of the violin’s distribution.
- **cmins**: A `matplotlib.collections.LineCollection` instance created to identify the bottom of each violin’s distribution.
- **cmaxes**: A `matplotlib.collections.LineCollection` instance created to identify the top of each violin’s distribution.
- **cbars**: A `matplotlib.collections.LineCollection` instance created to identify the centers of each violin’s distribution.
- **cmedians**: A `matplotlib.collections.LineCollection` instance created to identify the median values of each of the violin’s distribution.

**matplotlib.axes.Axes.bxp**

`Axes.bxp(bxpstats, positions=None, widths=None, vert=True, patch_artist=False, shownotches=False, showmeans=False, showcaps=True, showbox=True, showfliers=True, boxprops=None, capprops=None, meanprops=None, meanline=False, manage_xticks=True, zorder=None)`

Drawing function for box and whisker plots.

Make a box and whisker plot for each column of `x` or each vector in sequence `x`. The box extends from the lower to upper quartile values of the data, with a line at the median. The whiskers extend from the box to show the range of the data. Flier points are those past the end of the whiskers.

**Parameters bxpstats**: list of dicts

A list of dictionaries containing stats for each boxplot. Required keys are:

- **med**: The median (scalar float).
- **q1**: The first quartile (25th percentile) (scalar float).
- **q3**: The third quartile (75th percentile) (scalar float).
- **whislo**: Lower bound of the lower whisker (scalar float).
•\textit{whishi}: Upper bound of the upper whisker (scalar float).

Optional keys are:
•\textit{mean}: The mean (scalar float). Needed if showmeans=True.
•\textit{fliers}: Data beyond the whiskers (sequence of floats). Needed if showfliers=True.
•\textit{cilo} \& \textit{cihi}: Lower and upper confidence intervals about the median. Needed if shownotches=True.
•\textit{label}: Name of the dataset (string). If available, this will be used a tick label for the boxplot

\textbf{positions}: array-like, default = [1, 2, ..., n]

Sets the positions of the boxes. The ticks and limits are automatically set to match the positions.

\textbf{widths}: array-like, default = 0.5

Either a scalar or a vector and sets the width of each box. The default is 0.5, or 0.15*(distance between extreme positions) if that is smaller.

\textbf{vert}: bool, default = False

If True (default), makes the boxes vertical. If False, makes horizontal boxes.

\textbf{patch_artist}: bool, default = False

If False produces boxes with the \textit{Line2D} artist. If True produces boxes with the \textit{Patch} artist.

\textbf{shownotches}: bool, default = False

If False (default), produces a rectangular box plot. If True, will produce a notched box plot

\textbf{showmeans}: bool, default = False

If True, will toggle on the rendering of the means

\textbf{showcaps}: bool, default = True

If True, will toggle on the rendering of the caps

\textbf{showbox}: bool, default = True

If True, will toggle on the rendering of the box

\textbf{showfliers}: bool, default = True

If True, will toggle on the rendering of the fliers

\textbf{boxprops}: dict or None (default)

If provided, will set the plotting style of the boxes

\textbf{whiskerprops}: dict or None (default)
If provided, will set the plotting style of the whiskers

capprops : dict or None (default)
    If provided, will set the plotting style of the caps

flierprops : dict or None (default)
    If provided will set the plotting style of the fliers

medianprops : dict or None (default)
    If provided, will set the plotting style of the medians

meanprops : dict or None (default)
    If provided, will set the plotting style of the means

meanline : bool, default = False
    If True (and showmeans is True), will try to render the mean as a line spanning the full width of the box according to meanprops. Not recommended if shownotches is also True. Otherwise, means will be shown as points.

manage_xticks : bool, default = True
    If the function should adjust the xlim and xtick locations.

zorder : scalar, default = None
    The zorder of the resulting boxplot

Returns result : dict

A dictionary mapping each component of the boxplot to a list of the matplotlib.lines.Line2D instances created. That dictionary has the following keys (assuming vertical boxplots):

• boxes: the main body of the boxplot showing the quartiles and the median’s confidence intervals if enabled.

• medians: horizontal lines at the median of each box.

• whiskers: the vertical lines extending to the most extreme, non-outlier data points.

• caps: the horizontal lines at the ends of the whiskers.

• fliers: points representing data that extend beyond the whiskers (fliers).

• means: points or lines representing the means.
Examples

**Default**

**showmeans=True**

**showmeans=True, meanline=True**

**Tufte Style**
(showbox=False, showcaps=False)

**notch=True**

**showfliers=False**
I never said they'd be pretty

Custom boxprops

Custom medianprops and flierprops

Custom mean as point

Custom mean as line

19.1.5 Binned

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Axes.hexbin</td>
<td>Make a hexagonal binning plot.</td>
</tr>
<tr>
<td>Axes.hist</td>
<td>Plot a histogram.</td>
</tr>
<tr>
<td>Axes.hist2d</td>
<td>Make a 2D histogram plot.</td>
</tr>
</tbody>
</table>

`Axes.hexbin(x, y, C=None, gridsize=100, bins=None, xscale='linear', yscale='linear', extent=None, cmap=None, norm=None, vmin=None, vmax=None, alpha=None, linewidths=None, edgecolors='none', reduce_C_function=<function mean>, mincnt=None, marginals=False, **kwargs)`

Make a hexagonal binning plot.
Make a hexagonal binning plot of \( x \) versus \( y \), where \( x \), \( y \) are 1-D sequences of the same length, \( N \). If \( C \) is \( None \) (the default), this is a histogram of the number of occurrences of the observations at \((x[i],y[i])\).

If \( C \) is specified, it specifies values at the coordinate \((x[i],y[i])\). These values are accumulated for each hexagonal bin and then reduced according to \( \text{reduce}_C\_\text{function} \), which defaults to numpy’s mean function (\( \text{np.mean} \)). (If \( C \) is specified, it must also be a 1-D sequence of the same length as \( x \) and \( y \).)

**Parameters**

- \( x, y \) : array or masked array
- \( C \) : array or masked array, optional, default is \( None \)
- \( \text{gridsize} \) : int or (int, int), optional, default is 100
  - The number of hexagons in the \( x \)-direction, default is 100. The corresponding number of hexagons in the \( y \)-direction is chosen such that the hexagons are approximately regular. Alternatively, \( \text{gridsize} \) can be a tuple with two elements specifying the number of hexagons in the \( x \)-direction and the \( y \)-direction.
- \( \text{bins} \) : \{'log\'} or int or sequence, optional, default is \( None \)
  - If \( None \), no binning is applied; the color of each hexagon directly corresponds to its count value.
  - If \('log\'), use a logarithmic scale for the color map. Internally, \( \log_{10}(i + 1) \) is used to determine the hexagon color.
  - If an integer, divide the counts in the specified number of bins, and color the hexagons accordingly.
  - If a sequence of values, the values of the lower bound of the bins to be used.
- \( \text{xscale} \) : \{'linear', \'log\'}, optional, default is \'linear\'
  - Use a linear or log10 scale on the horizontal axis.
- \( \text{yscale} \) : \{'linear', \'log\'}, optional, default is \'linear\'
  - Use a linear or log10 scale on the vertical axis.
- \( \text{mincnt} \) : int > 0, optional, default is \( None \)
  - If not \( None \), only display cells with more than \( \text{mincnt} \) number of points in the cell.
- \( \text{marginals} \) : bool, optional, default is \( False \)
  - If \( \text{marginals} \) is \( True \), plot the marginal density as colormapped rectangles along the bottom of the \( x \)-axis and left of the \( y \)-axis.
- \( \text{extent} \) : scalar, optional, default is \( None \)
  - The limits of the bins. The default assigns the limits based on \( \text{gridsize}, x, y, \text{xscale} \) and \( \text{yscale} \).
  - If \( \text{xscale} \) or \( \text{yscale} \) is set to \'log\', the limits are expected to be the exponent for a power of 10. E.g., for \( \text{xlimits} \) of 1 and 50 in \'linear\' scale and \( \text{ylimits} \) of 10 and 1000 in \'log\' scale, enter \((1, 50, 1, 3)\).
  - Order of scalars is (left, right, bottom, top).
Returns object

a PolyCollection instance; use get_array() on this PolyCollection
to get the counts in each hexagon.

If marginals is True, horizontal bar and vertical bar (both PolyCollections)
will be attached to the return collection as attributes hbar and vbar.

Other Parameters
cmap : object, optional, default is None

a matplotlib.colors.Colormap instance. If None, defaults to rc image. cmap.

cmap.

Other Parameters
cmap : object, optional, default is None

a matplotlib.colors.Colormap instance. If None, defaults to rc image. cmap.

cmap.

Notes

The standard descriptions of all the Collection parameters:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg.filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or antialiaseds</td>
<td>Boolean or sequence of booleans</td>
</tr>
<tr>
<td>array</td>
<td>unknown</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clim</td>
<td>a length 2 sequence of floats</td>
</tr>
</tbody>
</table>
Table 19.26 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>clip_box</code></td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td><code>clip_on</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>clip_path</code></td>
<td>[ (Path, Transform)</td>
</tr>
<tr>
<td><code>cmap</code></td>
<td>a colormap or registered colormap name</td>
</tr>
<tr>
<td><code>color</code></td>
<td><code>matplotlib</code> color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td><code>contains</code></td>
<td>a callable function</td>
</tr>
<tr>
<td><code>edgecolor</code> or <code>edgecolors</code></td>
<td><code>matplotlib</code> color spec or sequence of specs</td>
</tr>
<tr>
<td><code>facecolor</code> or <code>facecolors</code></td>
<td><code>matplotlib</code> color spec or sequence of specs</td>
</tr>
<tr>
<td><code>figure</code></td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td><code>gid</code></td>
<td>an id string</td>
</tr>
<tr>
<td><code>hatch</code></td>
<td>[ ’/’</td>
</tr>
<tr>
<td><code>label</code></td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td><code>linestyle</code> or <code>dashes</code> or <code>linestyles</code></td>
<td>[‘solid’</td>
</tr>
<tr>
<td><code>linewidth</code> or <code>linewdiths</code> or <code>lw</code></td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td><code>norm</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>offsets</code></td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td><code>path_effects</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>picker</code></td>
<td>[None</td>
</tr>
<tr>
<td><code>pickradius</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>rasterized</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>sketch_params</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>snap</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>transform</code></td>
<td><code>Transform</code> instance</td>
</tr>
<tr>
<td><code>url</code></td>
<td>a url string</td>
</tr>
<tr>
<td><code>urls</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>visible</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>zorder</code></td>
<td>any number</td>
</tr>
</tbody>
</table>

**Note:**  In addition to the above described arguments, this function can take a `data` keyword argument. If such a `data` argument is given, the following arguments are replaced by `data[<arg>]`:

- All arguments with the following names: ‘x’, ‘y’.
Examples

Hexagon binning

With a log color scale

matplotlib.axes.Axes.hist

Axes.hist(x, bins=None, range=None, normed=False, weights=None, cumulative=False, bottom=None, histtype='bar', align='mid', orientation='vertical', rwidth=None, log=False, color=None, label=None, stacked=False, **kwargs)

Plot a histogram.

Compute and draw the histogram of x. The return value is a tuple (n, bins, patches) or ([n0, n1, ...], bins, [patches0, patches1,...]) if the input contains multiple data.

Multiple data can be provided via x as a list of datasets of potentially different length ([x0, x1, ...]), or as a 2-D ndarray in which each column is a dataset. Note that the ndarray form is transposed relative to the list form.

Masked arrays are not supported at present.

Parameters x : (n,) array or sequence of (n,) arrays

Input values, this takes either a single array or a sequency of arrays which are not required to be of the same length

bins : integer or array_like or ‘auto’, optional

If an integer is given, bins + 1 bin edges are returned, consistently with numpy.histogram() for numpy version >= 1.3.

Unequally spaced bins are supported if bins is a sequence.

If Numpy 1.11 is installed, may also be ‘auto’. 
Default is taken from the rcParam \texttt{hist.bins}.

**range** : tuple or None, optional

The lower and upper range of the bins. Lower and upper outliers are ignored. If not provided, \texttt{range} is \texttt{(x.min(), x.max())}. Range has no effect if \texttt{bins} is a sequence.

If \texttt{bins} is a sequence or \texttt{range} is specified, autoscaling is based on the specified bin range instead of the range of \texttt{x}.

Default is \texttt{None}

**normed** : boolean, optional

If \texttt{True}, the first element of the return tuple will be the counts normalized to form a probability density, i.e., \(n/(\text{len(x) ` dbin})\), i.e., the integral of the histogram will sum to 1. If \texttt{stacked} is also \texttt{True}, the sum of the histograms is normalized to 1.

Default is \texttt{False}

**weights** : (n,) array_like or None, optional

An array of weights, of the same shape as \texttt{x}. Each value in \texttt{x} only contributes its associated weight towards the bin count (instead of 1). If \texttt{normed} is True, the weights are normalized, so that the integral of the density over the range remains 1.

Default is \texttt{None}

**cumulative** : boolean, optional

If \texttt{True}, then a histogram is computed where each bin gives the counts in that bin plus all bins for smaller values. The last bin gives the total number of datapoints. If \texttt{normed} is also \texttt{True} then the histogram is normalized such that the last bin equals 1. If \texttt{cumulative} evaluates to less than 0 (e.g., -1), the direction of accumulation is reversed. In this case, if \texttt{normed} is also \texttt{True}, then the histogram is normalized such that the first bin equals 1.

Default is \texttt{False}

**bottom** : array_like, scalar, or None

Location of the bottom baseline of each bin. If a scalar, the base line for each bin is shifted by the same amount. If an array, each bin is shifted independently and the length of \texttt{bottom} must match the number of bins. If \texttt{None}, defaults to 0.

Default is \texttt{None}

**histtype** : {‘bar’, ‘barstacked’, ‘step’, ‘stepfilled’}, optional

The type of histogram to draw.

- ‘bar’ is a traditional bar-type histogram. If multiple data are given the bars are arranged side by side.
• ‘barstacked’ is a bar-type histogram where multiple data are stacked on top of each other.

• ‘step’ generates a lineplot that is by default unfilled.

• ‘stepfilled’ generates a lineplot that is by default filled.

Default is ‘bar’

align : {'left', 'mid', 'right'}, optional

Controls how the histogram is plotted.

• ‘left’: bars are centered on the left bin edges.

• ‘mid’: bars are centered between the bin edges.

• ‘right’: bars are centered on the right bin edges.

Default is ‘mid’

orientation : {'horizontal', 'vertical'}, optional

If ‘horizontal’, barh will be used for bar-type histograms and the bottom kwarg will be the left edges.

rwidth : scalar or None, optional

The relative width of the bars as a fraction of the bin width. If None, automatically compute the width.

Ignored if histtype is ‘step’ or ‘stepfilled’.

Default is None

log : boolean, optional

If True, the histogram axis will be set to a log scale. If log is True and x is a 1D array, empty bins will be filtered out and only the non-empty (n, bins, patches) will be returned.

Default is False

color : color or array_like of colors or None, optional

Color spec or sequence of color specs, one per dataset. Default (None) uses the standard line color sequence.

Default is None

label : string or None, optional

String, or sequence of strings to match multiple datasets. Bar charts yield multiple patches per dataset, but only the first gets the label, so that the legend command will work as expected.

default is None

stacked : boolean, optional
If True, multiple data are stacked on top of each other. If False, multiple data are arranged side by side if histtype is ‘bar’ or on top of each other if histtype is ‘step’.

Default is False

Returns n : array or list of arrays

The values of the histogram bins. See normed and weights for a description of the possible semantics. If input x is an array, then this is an array of length nbins. If input is a sequence arrays [data1, data2, ...], then this is a list of arrays with the values of the histograms for each of the arrays in the same order.

bins : array

The edges of the bins. Length nbins + 1 (nbins left edges and right edge of last bin). Always a single array even when multiple data sets are passed in.

patches : list or list of lists

Silent list of individual patches used to create the histogram or list of such list if multiple input datasets.

Other Parameters kwarg : Patch properties

See also:

hist2d 2D histograms

Notes

Until numpy release 1.5, the underlying numpy histogram function was incorrect with normed=`True` if bin sizes were unequal. MPL inherited that error. It is now corrected within MPL when using earlier numpy versions.
Examples

Histogram of IQ: \( \mu = 100, \sigma = 15 \)

Note: In addition to the above described arguments, this function can take a data keyword argument. If such a data argument is given, the following arguments are replaced by data[<arg>]:

- All arguments with the following names: ‘weights’, ‘x’.

**matplotlib.axes.Axes.hist2d**

Axes.hist2d(x, y, bins=10, range=None, normed=False, weights=None, cmin=None, cmax=None, **kwargs)

Make a 2D histogram plot.

Parameters x, y: array_like, shape (n,)

Input values

bins: [None | int | [int, int] | array_like | [array, array]]

The bin specification:

- If int, the number of bins for the two dimensions (nx=ny=bins).
- If [int, int], the number of bins in each dimension (nx, ny = bins).
• If array_like, the bin edges for the two dimensions (x_edges=y_edges=bins).

• If [array, array], the bin edges in each dimension (x_edges, y_edges = bins).

The default value is 10.

**range** : array_like shape(2, 2), optional, default: None

The leftmost and rightmost edges of the bins along each dimension (if not specified explicitly in the bins parameters): [[xmin, xmax], [ymin, ymax]]. All values outside of this range will be considered outliers and not tallied in the histogram.

**normed** : boolean, optional, default: False

Normalize histogram.

**weights** : array_like, shape (n,), optional, default: None

An array of values w_i weighing each sample (x_i, y_i).

**cmin** : scalar, optional, default: None

All bins that has count less than cmin will not be displayed and these count values in the return value count histogram will also be set to nan upon return

**cmax** : scalar, optional, default: None

All bins that has count more than cmax will not be displayed (set to none before passing to imshow) and these count values in the return value count histogram will also be set to nan upon return

**Returns** The return value is (counts, xedges, yedges, Image).

**Other Parameters**

**cmap** : {Colormap, string}, optional

A `matplotlib.colors.Colormap` instance. If not set, use rc settings.

**norm** : Normalize, optional

A `matplotlib.colors.Normalize` instance is used to scale luminance data to [0, 1]. If not set, defaults to `Normalize()`.

**vmin/vmax** : {None, scalar}, optional

Arguments passed to the Normalize instance.

**alpha** : 0 <= scalar <= 1 or None, optional

The alpha blending value.

**See also:**

**hist** 1D histogram
Notes

Rendering the histogram with a logarithmic color scale is accomplished by passing a `colors.LogNorm` instance to the `norm` keyword argument. Likewise, power-law normalization (similar in effect to gamma correction) can be accomplished with `colors.PowerNorm`.

Examples

![Histogram with logarithmic color scale](image)

**Note:** In addition to the above described arguments, this function can take a `data` keyword argument. If such a `data` argument is given, the following arguments are replaced by `data[<arg>]`:

- All arguments with the following names: ‘weights’, ‘x’, ‘y’.

19.1.6 Contours

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Axes.clabel</code></td>
<td>Label a contour plot.</td>
</tr>
<tr>
<td><code>Axes.contour</code></td>
<td>Plot contours.</td>
</tr>
<tr>
<td><code>Axes.contourf</code></td>
<td>Plot contours.</td>
</tr>
</tbody>
</table>
**Axes.clabel**

**Axes.clabel**(CS, *args, **kwargs)

Label a contour plot.

Call signature:

```python
clabel(cs, **kwargs)
```

Adds labels to line contours in cs, where cs is a ContourSet object returned by contour.

```python
clabel(cs, v, **kwargs)
```

only labels contours listed in v.

Optional keyword arguments:

- **fontsize**: size in points or relative size e.g., ‘smaller’, ‘x-large’
- **colors**:  
  - if None, the color of each label matches the color of the corresponding contour  
  - if one string color, e.g., colors = ‘r’ or colors = ‘red’, all labels will be plotted in this color  
  - if a tuple of matplotlib color args (string, float, rgb, etc), different labels will be plotted in different colors in the order specified
- **inline**: controls whether the underlying contour is removed or not. Default is True.
- **inline_spacing**: space in pixels to leave on each side of label when placing inline. Defaults to 5. This spacing will be exact for labels at locations where the contour is straight, less so for labels on curved contours.
- **fmt**: a format string for the label. Default is ‘%1.3f’ Alternatively, this can be a dictionary matching contour levels with arbitrary strings to use for each contour level (i.e., fmt[level]=string), or it can be any callable, such as a Formatter instance, that returns a string when called with a numeric contour level.
- **manual**: if True, contour labels will be placed manually using mouse clicks. Click the first button near a contour to add a label, click the second button (or potentially both mouse buttons at once) to finish adding labels. The third button can be used to remove the last label added, but only if labels are not inline. Alternatively, the keyboard can be used to select label locations (enter to end label placement, delete or backspace act like the third mouse button, and any other key will select a label location).
  
  manual can be an iterable object of x,y tuples. Contour labels will be created as if mouse is clicked at each x,y positions.
- **rightside_up**: if True (default), label rotations will always be plus or minus 90 degrees from level.
use_clabeltext: if True (default is False), ClabelText class (instead of matplotlib.Text) is used to create labels. ClabelText recalculates rotation angles of texts during the drawing time, therefore this can be used if aspect of the axes changes.
Single color - negative contours dashed
19.1. Plotting
Chapter 19. Axes class
matplotlib.axes.Axes.contour

Axes.contour(*args, **kwargs)
Plot contours.

contour() and contourf() draw contour lines and filled contours, respectively. Except as noted, function signatures and return values are the same for both versions.

contourf() differs from the MATLAB version in that it does not draw the polygon edges. To draw edges, add line contours with calls to contour().

Call signatures:

```
contour(Z)
```

make a contour plot of an array Z. The level values are chosen automatically.

```
contour(X, Y, Z)
```

X, Y specify the (x, y) coordinates of the surface

```
contour(Z, N)
contour(X, Y, Z, N)
```

contour up to N automatically-chosen levels.
contour(Z,V)
contour(X,Y,Z,V)

draw contour lines at the values specified in sequence V, which must be in increasing order.

contourf(..., V)

fill the \( \text{len}(V)-1 \) regions between the values in V, which must be in increasing order.

contour(Z, **kwargs)

Use keyword args to control colors, linewidth, origin, cmap ... see below for more details.

\( X \) and \( Y \) must both be 2-D with the same shape as \( Z \), or they must both be 1-D such that \( \text{len}(X) \) is the number of columns in \( Z \) and \( \text{len}(Y) \) is the number of rows in \( Z \).

\( C = \text{contour}(\ldots) \) returns a QuadContourSet object.

Optional keyword arguments:

- **corner_mask**: [True | False | ‘legacy’] Enable/disable corner masking, which only has an effect if \( Z \) is a masked array. If False, any quad touching a masked point is masked out. If True, only the triangular corners of quads nearest those points are always masked out, other triangular corners comprising three unmasked points are contoured as usual. If ‘legacy’, the old contouring algorithm is used, which is equivalent to False and is deprecated, only remaining whilst the new algorithm is tested fully.

  If not specified, the default is taken from rcParams[‘contour.corner_mask’], which is True unless it has been modified.

- **colors**: [None | string | (mpl_colors)] If None, the colormap specified by cmap will be used.

  If a string, like ‘r’ or ‘red’, all levels will be plotted in this color.

  If a tuple of matplotlib color args (string, float, rgb, etc), different levels will be plotted in different colors in the order specified.

- **alpha**: float The alpha blending value

- **cmap**: [None | Colormap] A colormap instance or None. If cmap is None and colors is None, a default Colormap is used.

- **norm**: [None | Normalize] A matplotlib.colors.Normalize instance for scaling data values to colors. If norm is None and colors is None, the default linear scaling is used.

- **vmin, vmax**: [None | scalar] If not None, either or both of these values will be supplied to the matplotlib.colors.Normalize instance, overriding the default color scaling based on levels.

- **levels**: [level0, level1, ..., leveln] A list of floating point numbers indicating the level curves to draw, in increasing order; e.g., to draw just the zero contour pass levels=[0]
**origin:** [ *None* | ‘upper’ | ‘lower’ | ‘image’ ] If *None*, the first value of *Z* will correspond to the lower left corner, location (0,0). If ‘image’, the rc value for `image.origin` will be used.

This keyword is not active if *X* and *Y* are specified in the call to `contour`.

**extent:** [ *None* | (*x0,x1,y0,y1*) ]

If `origin` is not *None*, then `extent` is interpreted as in `matplotlib.pyplot.imshow()`: it gives the outer pixel boundaries. In this case, the position of *Z*[0,0] is the center of the pixel, not a corner. If `origin` is *None*, then (*x0, y0*) is the position of *Z*[0,0], and (*x1, y1*) is the position of *Z*[-1,-1].

This keyword is not active if *X* and *Y* are specified in the call to `contour`.

**locator:** [ *None* | `ticker.Locator subclass` ] If `locator` is *None*, the default `MaxNLocator` is used. The locator is used to determine the contour levels if they are not given explicitly via the *V* argument.

**extend:** [ ‘neither’ | ‘both’ | ‘min’ | ‘max’ ] Unless this is ‘neither’, contour levels are automatically added to one or both ends of the range so that all data are included. These added ranges are then mapped to the special colormap values which default to the ends of the colormap range, but can be set via `matplotlib.colors.Colormap.set_under()` and `matplotlib.colors.Colormap.set_over()` methods.

**xunits, yunits:** [ *None* | registered units ] Override axis units by specifying an instance of `matplotlib.units.ConversionInterface`.

**antialiased:** [ *True* | *False* ] enable antialiasing, overriding the defaults. For filled contours, the default is *True*. For line contours, it is taken from `rcParams['lines.antialiased']`.

**nchunk:** [ 0 | integer ] If 0, no subdivision of the domain. Specify a positive integer to divide the domain into subdomains of *nchunk* by *nchunk* quads. Chunking reduces the maximum length of polygons generated by the contouring algorithm which reduces the rendering workload passed on to the backend and also requires slightly less RAM. It can however introduce rendering artifacts at chunk boundaries depending on the backend, the `antialiased` flag and value of *alpha*.

Contour-only keyword arguments:

**linewidths:** [ *None* | *number* | tuple of numbers ] If `linewidths` is *None*, the default width in `lines.linewidth` in `matplotlibrc` is used.

If a number, all levels will be plotted with this linewidth.

If a tuple, different levels will be plotted with different linewidths in the order specified.

**linestyles:** [ *None* | ‘solid’ | ‘dashed’ | ‘dashdot’ | ‘dotted’ ] If `linestyles` is *None*, the default is ‘solid’ unless the lines are monochrome. In that case, negative contours will take their linestyle from the `matplotlibrc` `contour.negative_linestyle` setting.
linestyles can also be an iterable of the above strings specifying a set of linestyles to be used. If this iterable is shorter than the number of contour levels it will be repeated as necessary.

contourf-only keyword arguments:

**hatches**: A list of cross hatch patterns to use on the filled areas. If None, no hatching will be added to the contour. Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Note: contourf fills intervals that are closed at the top; that is, for boundaries \( z_1 \) and \( z_2 \), the filled region is:

\[
z_1 < z \leq z_2
\]

There is one exception: if the lowest boundary coincides with the minimum value of the \( z \) array, then that minimum value will be included in the lowest interval.

**Examples:**

![Simplest default with labels](image-url)
labels at selected locations
Single color - negative contours dashed
Single color - negative contours solid
Crazy lines
19.1. Plotting
Chapter 19. Axes class
Listed colors (3 masked regions)
extend = neither
extend = both
extend = min
extend = max
matplotlib.axes.Axes.contourf

Axes.contourf(*args, **kwargs)
Plot contours.

contour() and contourf() draw contour lines and filled contours, respectively. Except as noted, function signatures and return values are the same for both versions.

contourf() differs from the MATLAB version in that it does not draw the polygon edges. To draw edges, add line contours with calls to contour().

Call signatures:

contour(Z)

make a contour plot of an array Z. The level values are chosen automatically.

contour(X, Y, Z)

X, Y specify the (x, y) coordinates of the surface

contour(Z, N)
contourf(X, Y, Z, N)

contour up to $N$ automatically-chosen levels.

19.1. Plotting
Contour drawing

- **`contour(Z,V)`**
  - draw contour lines at the values specified in sequence `V`, which must be in increasing order.

- **`contourf(..., V)`**
  - fill the `len(V)-1` regions between the values in `V`, which must be in increasing order.

- **`contour(Z, **kwargs)`**
  - Use keyword args to control colors, linewidth, origin, cmap ... see below for more details.

  - `X` and `Y` must both be 2-D with the same shape as `Z`, or they must both be 1-D such that `len(X)` is the number of columns in `Z` and `len(Y)` is the number of rows in `Z`.

  - `C = contour(...)` returns a `QuadContourSet` object.

Optional keyword arguments:

- **`corner_mask`** [True | False | ‘legacy’] Enable/disable corner masking, which only has an effect if `Z` is a masked array. If `False`, any quad touching a masked point is masked out. If `True`, only the triangular corners of quads nearest those points are always masked out, other triangular corners comprising three unmasked points are contoured as usual. If ‘legacy’, the old contouring algorithm is used, which is equivalent to `False` and is deprecated, only remaining whilst the new algorithm is tested fully.

  - If not specified, the default is taken from `rcParams[‘contour.corner_mask’]`, which is `True` unless it has been modified.

- **`colors`** [None | string | (mpl_colors)] If `None`, the colormap specified by `cmap` will be used.
  - If a string, like ‘r’ or ‘red’, all levels will be plotted in this color.
  - If a tuple of matplotlib color args (string, float, rgb, etc), different levels will be plotted in different colors in the order specified.

- **`alpha`** [float] The alpha blending value

- **`cmap`** [None | Colormap] A cm `Colormap` instance or `None`. If `cmap` is `None` and `colors` is `None`, a default Colormap is used.

- **`norm`** [None | Normalize] A `matplotlib.colors.Normalize` instance for scaling data values to colors. If `norm` is `None` and `colors` is `None`, the default linear scaling is used.

- **`vmin, vmax`** [None | scalar] If not `None`, either or both of these values will be supplied to the `matplotlib.colors.Normalize` instance, overriding the default color scaling based on `levels`.

- **`levels`** [level0, level1, ..., leveln] A list of floating point numbers indicating the level curves to draw, in increasing order; e.g., to draw just the zero contour pass `levels=[0]`
**origin:** [ `None` | ‘upper’ | ‘lower’ | ‘image’ ] If `None`, the first value of `Z` will correspond to the lower left corner, location (0,0). If ‘image’, the rc value for `image.origin` will be used.

This keyword is not active if `X` and `Y` are specified in the call to `contour`.

**extent:** [ `None` | `(x0,x1,y0,y1)` ]

If `origin` is not `None`, then `extent` is interpreted as in `matplotlib.pyplot.imshow()`: it gives the outer pixel boundaries. In this case, the position of `Z[0,0]` is the center of the pixel, not a corner. If `origin` is `None`, then `(x0, y0)` is the position of `Z[0,0]`, and `(x1, y1)` is the position of `Z[-1,-1]`.

This keyword is not active if `X` and `Y` are specified in the call to `contour`.

**locator:** [ `None` | `ticker.Locator subclass` ] If `locator` is `None`, the default `MaxNLocator` is used. The locator is used to determine the contour levels if they are not given explicitly via the `V` argument.

**extend:** [ ‘neither’ | ‘both’ | ‘min’ | ‘max’ ] Unless this is ‘neither’, contour levels are automatically added to one or both ends of the range so that all data are included. These added ranges are then mapped to the special colormap values which default to the ends of the colormap range, but can be set via `matplotlib.colors.Colormap.set_under()` and `matplotlib.colors.Colormap.set_over()` methods.

**xunits, yunits:** [ `None` | `registered units` ] Override axis units by specifying an instance of a `matplotlib.units.ConversionInterface`.

**antialiased:** [ `True` | `False` ] enable antialiasing, overriding the defaults. For filled contours, the default is `True`. For line contours, it is taken from `rc-Params['lines.antialiased']`.

**nchunk:** [ `0` | `integer` ] If 0, no subdivision of the domain. Specify a positive integer to divide the domain into subdomains of `nchunk` by `nchunk` quads. Chunking reduces the maximum length of polygons generated by the contouring algorithm which reduces the rendering workload passed on to the backend and also requires slightly less RAM. It can however introduce rendering artifacts at chunk boundaries depending on the backend, the `antialiased` flag and value of `alpha`.

contour-only keyword arguments:

**linewidths:** [ `None` | `number` | `tuple of numbers` ] If `linewidths` is `None`, the default width in `lines.linewidth` in `matplotlibrc` is used.

If a number, all levels will be plotted with this linewidth.

If a tuple, different levels will be plotted with different linewidths in the order specified.

**linestyles:** [ `None` | ‘solid’ | ‘dashed’ | ‘dashdot’ | ‘dotted’ ] If `linestyles` is `None`, the default is ‘solid’ unless the lines are monochrome. In that case, negative contours will take their linestyle from the `matplotlibrc contour.negative_linestyle` setting.
*linestyles* can also be an iterable of the above strings specifying a set of linestyles to be used. If this iterable is shorter than the number of contour levels it will be repeated as necessary.

contourf-only keyword arguments:

**hatches**: A list of cross hatch patterns to use on the filled areas. If None, no hatching will be added to the contour. Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Note: contourf fills intervals that are closed at the top; that is, for boundaries $z_1$ and $z_2$, the filled region is:

$$z_1 < z \leq z_2$$

There is one exception: if the lowest boundary coincides with the minimum value of the $z$ array, then that minimum value will be included in the lowest interval.

Examples:
labels at selected locations
Single color - negative contours dashed
Single color - negative contours solid
Chapter 19. Axes class
Lines with colorbar

1.0
0.5
0.0
-0.5
-1.0
-1.5
-2.0
-1.2
-0.8
-0.4
0.0
0.4
0.8
1.2

1.0
0.5
0.0
-0.5
-1.0
-1.5
-2.0
-1.2
-0.8
-0.4
0.0
0.4
0.8
1.2

19.1. Plotting
Chapter 19. Axes class
Listed colors (3 masked regions)
Matplotlib, Release 2.0.2

Chapter 19. Axes class
19.1.7 Array

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Axes.imshow</code></td>
<td>Display an image on the axes.</td>
</tr>
<tr>
<td><code>Axes.matshow</code></td>
<td>Plot a matrix or array as an image.</td>
</tr>
<tr>
<td><code>Axes.pcolor</code></td>
<td>Create a pseudocolor plot of a 2-D array.</td>
</tr>
<tr>
<td><code>Axes.pcolorfast</code></td>
<td>Pseudocolor plot of a 2-D array</td>
</tr>
<tr>
<td><code>Axes.pcolormesh</code></td>
<td>Plot a quadrilateral mesh.</td>
</tr>
<tr>
<td><code>Axes.spy</code></td>
<td>Plot the sparsity pattern on a 2-D array.</td>
</tr>
</tbody>
</table>

`matplotlib.axes.Axes.imshow`

`Axes.imshow(X, cmap=None, norm=None, aspect=None, interpolation=None, alpha=None, vmin=None, vmax=None, origin=None, extent=None, shape=None, filternorm=1, ffilterrad=4.0, imlim=None, resample=None, url=None, **kwargs)`

Display an image on the axes.

**Parameters**

- **X**: array_like, shape (n, m) or (n, m, 3) or (n, m, 4)

  Display the image in X to current axes. X may be an array or a PIL image. If X is an array, it can have the following shapes and types:
  - MxN – values to be mapped (float or int)
- MxNx3 – RGB (float or uint8)
- MxNx4 – RGBA (float or uint8)

The value for each component of MxNx3 and MxNx4 float arrays should be in the range 0.0 to 1.0. MxN arrays are mapped to colors based on the norm (mapping scalar to scalar) and the cmap (mapping the normed scalar to a color).

cmap : Colormap, optional, default: None

If None, default to rc image.cmap value. cmap is ignored if X is 3-D, directly specifying RGB(A) values.

aspect : [‘auto’ | ‘equal’ | scalar], optional, default: None

If ‘auto’, changes the image aspect ratio to match that of the axes.

If ‘equal’, and extent is None, changes the axes aspect ratio to match that of the image. If extent is not None, the axes aspect ratio is changed to match that of the extent.

If None, default to rc image.aspect value.

interpolation : string, optional, default: None


If interpolation is None, default to rc image.interpolation. See also the filternorm and filterrad parameters. If interpolation is ‘none’, then no interpolation is performed on the Agg, ps and pdf backends. Other backends will fall back to ‘nearest’.

norm : Normalize, optional, default: None

A Normalize instance is used to scale a 2-D float X input to the (0, 1) range for input to the cmap. If norm is None, use the default func:normalize. If norm is an instance of NoNorm, X must be an array of integers that index directly into the lookup table of the cmap.

vmin, vmax : scalar, optional, default: None

vmin and vmax are used in conjunction with norm to normalize luminance data. Note if you pass a norm instance, your settings for vmin and vmax will be ignored.

alpha : scalar, optional, default: None

The alpha blending value, between 0 (transparent) and 1 (opaque)

origin : [‘upper’ | ‘lower’], optional, default: None

Place the [0,0] index of the array in the upper left or lower left corner of the axes. If None, default to rc image.origin.

extent : scalars (left, right, bottom, top), optional, default: None
The location, in data-coordinates, of the lower-left and upper-right corners. If None, the image is positioned such that the pixel centers fall on zero-based (row, column) indices.

**shape**: scalars (columns, rows), optional, default: None

For raw buffer images

**filternorm**: scalar, optional, default: 1

A parameter for the antigrain image resize filter. From the antigrain documentation, if filternorm = 1, the filter normalizes integer values and corrects the rounding errors. It doesn’t do anything with the source floating point values, it corrects only integers according to the rule of 1.0 which means that any sum of pixel weights must be equal to 1.0. So, the filter function must produce a graph of the proper shape.

**filterrad**: scalar, optional, default: 4.0

The filter radius for filters that have a radius parameter, i.e. when interpolation is one of: ‘sinc’, ‘lanczos’ or ‘blackman’

**Returns image**: *AxesImage*

**Other Parameters kwargs**: *Artist* properties.

See also:

**matshow** Plot a matrix or an array as an image.

**Notes**

Unless *extent* is used, pixel centers will be located at integer coordinates. In other words: the origin will coincide with the center of pixel (0, 0).
Examples

Note: In addition to the above described arguments, this function can take a `data` keyword argument. If such a `data` argument is given, the following arguments are replaced by `data[<arg>]:`

- All positional and all keyword arguments.

**matplotlib.axes.Axes.matshow**

`Axes.matshow(Z, **kwargs)`

Plot a matrix or array as an image.

The matrix will be shown the way it would be printed, with the first row at the top. Row and column numbering is zero-based.

**Parameters**

- `Z`: array_like shape (n, m)
  
The matrix to be displayed.

**Returns**

- `image`: `AxesImage`

**Other Parameters**

- `kwargs`: `imshow` arguments
Sets origin to ‘upper’, ‘interpolation’ to ‘nearest’ and ‘aspect’ to equal.

See also:

*imshow* plot an image

**Examples**

```python
import matplotlib.pyplot as plt

import numpy as np

# Create data
X = np.arange(30)
Y = np.arange(30)
C = np.sin(np.sqrt(X**2 + Y**2))

# Create a figure
fig = plt.figure()

# Add an axes to the figure and specify the location of the first plot
ax = fig.add_subplot(1, 1, 1)

# Create a pseudocolor plot of a 2-D array
ax.pcolor(C)

fig.show()
```

---

*matplotlib.axes.Axes.pcolor*

**Axes.pcolor(*args, **kwargs)**

Create a pseudocolor plot of a 2-D array.

**Note:** *pcolor* can be very slow for large arrays; consider using the similar but much faster *pcolormesh()* instead.

Call signatures:

```python
pcolor(C, **kwargs)
pcolor(X, Y, C, **kwargs)
```

*C* is the array of color values.

*X* and *Y*, if given, specify the (*x*, *y*) coordinates of the colored quadrilaterals; the quadrilateral for *C[i,j]* has corners at:

- \((X[i, j], Y[i, j])\)
- \((X[i, j+1], Y[i, j+1])\)
- \((X[i+1, j], Y[i+1, j])\)
- \((X[i+1, j+1], Y[i+1, j+1])\).
Ideally the dimensions of $X$ and $Y$ should be one greater than those of $C$; if the dimensions are the same, then the last row and column of $C$ will be ignored.

Note that the column index corresponds to the $x$-coordinate, and the row index corresponds to $y$; for details, see the Grid Orientation section below.

If either or both of $X$ and $Y$ are 1-D arrays or column vectors, they will be expanded as needed into the appropriate 2-D arrays, making a rectangular grid.

$X$, $Y$ and $C$ may be masked arrays. If either $C[i, j]$, or one of the vertices surrounding $C[i, j]$ ($X$ or $Y$ at $[i, j], [i, j+1], [i+1, j], [i+1, j+1]$) is masked, nothing is plotted.

Keyword arguments:

- **$cmap$**: [None | Colormap] A matplotlib.colors.Colormap instance. If None, use rc settings.
- **$norm$**: [None | Normalize] An matplotlib.colors.Normalize instance is used to scale luminance data to 0,1. If None, defaults to normalize().
- **$vmin/vmax$**: [None | scalar] $vmin$ and $vmax$ are used in conjunction with $norm$ to normalize luminance data. If either is None, it is autoscaled to the respective min or max of the color array $C$. If not None, $vmin$ or $vmax$ passed in here override any pre-existing values supplied in the $norm$ instance.
- **$shading$**: ['flat' | 'faceted'] If ‘faceted’, a black grid is drawn around each rectangle; if ‘flat’, edges are not drawn. Default is ‘flat’, contrary to MATLAB.

This kwarg is deprecated; please use ‘edgecolors’ instead:

- shading='flat' – edgecolors='none'
- shading='faceted' – edgecolors='k'

- **$edgecolors$**: [None | 'none' | color | color sequence] If None, the rc setting is used by default.
  
  If 'none', edges will not be visible.

  An mpl color or sequence of colors will set the edge color

- **$alpha$**: 0 <= scalar <= 1 or None the alpha blending value

- **$snap$**: bool Whether to snap the mesh to pixel boundaries.

Return value is a matplotlib.collections.Collection instance. The grid orientation follows the MATLAB convention: an array $C$ with shape (nrows, ncolumns) is plotted with the column number as $X$ and the row number as $Y$, increasing up; hence it is plotted the way the array would be printed, except that the $Y$ axis is reversed. That is, $C$ is taken as $C^y(x, y)$.

Similarly for meshgrid():

```python
x = np.arange(5)
y = np.arange(3)
X, Y = np.meshgrid(x, y)
```

is equivalent to:

```python
```
```python
X = array([[0, 1, 2, 3, 4],
           [0, 1, 2, 3, 4],
           [0, 1, 2, 3, 4]]

Y = array([[0, 0, 0, 0, 0],
           [1, 1, 1, 1, 1],
           [2, 2, 2, 2, 2]])
```

so if you have:

```python
C = rand(len(x), len(y))
```

then you need to transpose C:

```python
pcolor(X, Y, C.T)
```

or:

```python
pcolor(C.T)
```

MATLAB `pcolor()` always discards the last row and column of `C`, but matplotlib displays the last row and column if `X` and `Y` are not specified, or if `X` and `Y` have one more row and column than `C`.

Kwargs can be used to control the `PolyCollection` properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>agg_filter</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>alpha</code></td>
<td>float or None</td>
</tr>
<tr>
<td><code>animated</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>antialiased</code></td>
<td>or <code>antialiaseds</code> Boolean or sequence of booleans</td>
</tr>
<tr>
<td><code>array</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>axes</code></td>
<td>an <code>Axes</code> instance</td>
</tr>
<tr>
<td><code>clim</code></td>
<td>a length 2 sequence of floats</td>
</tr>
<tr>
<td><code>clip_box</code></td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td><code>clip_on</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>clip_path</code></td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td><code>cmap</code></td>
<td>a colormap or registered colormap name</td>
</tr>
<tr>
<td><code>color</code></td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td><code>contains</code></td>
<td>a callable function</td>
</tr>
<tr>
<td><code>edgecolor</code></td>
<td>or <code>edgecolors</code> matplotlib color spec or sequence of specs</td>
</tr>
<tr>
<td><code>facecolor</code></td>
<td>or <code>facecolors</code> matplotlib color spec or sequence of specs</td>
</tr>
<tr>
<td><code>figure</code></td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td><code>gid</code></td>
<td>an id string</td>
</tr>
<tr>
<td><code>hatch</code></td>
<td>[‘</td>
</tr>
<tr>
<td><code>label</code></td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td><code>linestyle</code></td>
<td>or <code>dashes</code> or <code>linestyles</code> [‘solid’</td>
</tr>
<tr>
<td><code>linewidth</code></td>
<td>or <code>linewidths</code> or <code>lw</code> float or sequence of floats</td>
</tr>
<tr>
<td><code>norm</code></td>
<td>unknown</td>
</tr>
</tbody>
</table>

Continued on next page
Table 19.29 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>offset_position</td>
<td>unknown</td>
</tr>
<tr>
<td>offsets</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>pickradius</td>
<td>unknown</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>urls</td>
<td>unknown</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

**Note:** The default antialiaseds is False if the default edgecolors*="none" is used. This eliminates artificial lines at patch boundaries, and works regardless of the value of alpha. If *edgecolors is not “none”, then the default antialiaseds is taken from rcParams[‘patch.antialiased’], which defaults to True. Stroking the edges may be preferred if alpha is 1, but will cause artifacts otherwise.

See also:

*pcolormesh()* For an explanation of the differences between pcolor and pcolormesh.

**Note:** In addition to the above described arguments, this function can take a data keyword argument. If such a data argument is given, the following arguments are replaced by data[<arg>]:

- All positional and all keyword arguments.

**matplotlib.axes.Axes.pcolorfast**

Axes.pcolorfast(*args, **kwargs)

pseudocolor plot of a 2-D array

Experimental; this is a pcolor-type method that provides the fastest possible rendering with the Agg backend, and that can handle any quadrilateral grid. It supports only flat shading (no outlines), it lacks support for log scaling of the axes, and it does not have a pyplot wrapper.

Call signatures:
C is the 2D array of color values corresponding to quadrilateral cells. Let \((nr, nc)\) be its shape. \(C\) may be a masked array.

\[
\text{ax.pcolorfast}(C, **\text{kwargs}) \quad \text{is equivalent to} \quad \text{ax.pcolorfast}([0, nc], [0, nr], C, **\text{kwargs})
\]

\(x_r, y_r\) specify the ranges of \(x\) and \(y\) corresponding to the rectangular region bounding \(C\). If:

\[
x_r = [x_0, x_1]
\]

and:

\[
y_r = [y_0, y_1]
\]

then \(x\) goes from \(x_0\) to \(x_1\) as the second index of \(C\) goes from 0 to \(nc\), etc. \((x_0, y_0)\) is the outermost corner of cell \((0,0)\), and \((x_1, y_1)\) is the outermost corner of cell \((nr-1, nc-1)\). All cells are rectangles of the same size. This is the fastest version.

\(x, y\) are monotonic 1D arrays of length \(nc + 1\) and \(nr + 1\), respectively, giving the \(x\) and \(y\) boundaries of the cells. Hence the cells are rectangular but the grid may be nonuniform. The speed is intermediate. (The grid is checked, and if found to be uniform the fast version is used.)

\(X\) and \(Y\) are 2D arrays with shape \((nr + 1, nc + 1)\) that specify the \((x,y)\) coordinates of the corners of the colored quadrilaterals; the quadrilateral for \(C[i,j]\) has corners at \((X[i,j],Y[i,j]), (X[i,j+1],Y[i,j+1]), (X[i+1,j],Y[i+1,j]), (X[i+1,j+1],Y[i+1,j+1])\). The cells need not be rectangular. This is the most general, but the slowest to render. It may produce faster and more compact output using \(ps\), \(pdf\), and \(svg\) backends, however.

Note that the column index corresponds to the \(x\)-coordinate, and the row index corresponds to \(y\); for details, see \textit{Grid Orientation}.

Optional keyword arguments:

- \textit{cmap: \[None \mid Colormap\]} A \texttt{matplotlib.colors.Colormap} instance from \texttt{cm}. If \texttt{None}, use rc settings.

- \textit{norm: \[None \mid Normalize\]} A \texttt{matplotlib.colors.Normalize} instance is used to scale luminance data to 0, 1. If \texttt{None}, defaults to \texttt{normalize()}

- \textit{vmin/vmax: \[None \mid scalar\]} \(vmin\) and \(vmax\) are used in conjunction with norm to normalize luminance data. If either are \texttt{None}, the min and max of the color array \(C\) is used. If you pass a norm instance, \(vmin\) and \(vmax\) will be \texttt{None}.

- \textit{alpha: 0 <= scalar <= 1 or None} the alpha blending value

Return value is an image if a regular or rectangular grid is specified, and a \texttt{QuadMesh} collection in the general quadrilateral case.
Matplotlib, Release 2.0.2

**Note:** In addition to the above described arguments, this function can take a `data` keyword argument. If such a `data` argument is given, the following arguments are replaced by `data[<arg>]`:

- All positional and all keyword arguments.

```python
matplotlib.axes.Axes.pcolormesh

Axes.pcolormesh(*args, **kwargs)
Plot a quadrilateral mesh.
Call signatures:

```python
pcolormesh(C)
pcolormesh(X, Y, C)
pcolormesh(C, **kwargs)

Create a pseudocolor plot of a 2-D array.

pcolormesh is similar to `pcolor()`, but uses a different mechanism and returns a different object; `pcolor` returns a `PolyCollection` but `pcolormesh` returns a `QuadMesh`. It is much faster, so it is almost always preferred for large arrays.

`C` may be a masked array, but `X` and `Y` may not. Masked array support is implemented via `cmap` and `norm`; in contrast, `pcolor()` simply does not draw quadrilaterals with masked colors or vertices.

Keyword arguments:

- **`cmap`**: `[None | Colormap]` A `matplotlib.colors.Colormap` instance. If `None`, use rc settings.
- **`norm`**: `[None | Normalize]` A `matplotlib.colors.Normalize` instance is used to scale luminance data to 0,1. If `None`, defaults to `normalize()`.
- **`vmin/vmax`**: `[None | scalar]` `vmin` and `vmax` are used in conjunction with `norm` to normalize luminance data. If either is `None`, it is autoscaled to the respective min or max of the color array `C`. If not `None`, `vmin` or `vmax` passed in here override any pre-existing values supplied in the `norm` instance.
- **`shading`**: ['flat' | 'gouraud'] ‘flat’ indicates a solid color for each quad. When ‘gouraud’, each quad will be Gouraud shaded. When gouraud shading, edgecolors is ignored.
- **`edgecolors`**: [None | 'None' | 'face' | color | color sequence]
  If `None`, the rc setting is used by default.
  If 'None', edges will not be visible.
  If 'face', edges will have the same color as the faces.
  An mpl color or sequence of colors will set the edge color.
- **`alpha`**: `0 <= scalar <= 1` or `None` the alpha blending value
Return value is a `matplotlib.collections.QuadMesh` object.

**kwargs can be used to control the `matplotlib.collections.QuadMesh` properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>agg_filter</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>alpha</code></td>
<td>float or None</td>
</tr>
<tr>
<td><code>animated</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>antialiased</code></td>
<td>Boolean or sequence of booleans</td>
</tr>
<tr>
<td><code>array</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>axes</code></td>
<td>an <code>Axes</code> instance</td>
</tr>
<tr>
<td><code>clim</code></td>
<td>a length 2 sequence of floats</td>
</tr>
<tr>
<td><code>clip_box</code></td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td><code>clip_on</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>clip_path</code></td>
<td>[(Path, Transform), Patch</td>
</tr>
<tr>
<td><code>cmap</code></td>
<td>a colormap or registered colormap name</td>
</tr>
<tr>
<td><code>color</code></td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td><code>contains</code></td>
<td>a callable function</td>
</tr>
<tr>
<td><code>edgecolor</code> or edgecolors</td>
<td>matplotlib color spec or sequence of specs</td>
</tr>
<tr>
<td><code>facecolor</code> or facecolors</td>
<td>matplotlib color spec or sequence of specs</td>
</tr>
<tr>
<td><code>figure</code></td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td><code>gid</code></td>
<td>an id string</td>
</tr>
<tr>
<td><code>label</code></td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td><code>linestyle</code> or dashes or linestyles</td>
<td>[‘solid’, ‘dashed’, ‘dashdot’, ‘dotted’</td>
</tr>
<tr>
<td><code>linewidth</code> or linewidens or lw</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td><code>norm</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>offset_position</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>offsets</code></td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td><code>path_effects</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>picker</code></td>
<td>[None</td>
</tr>
<tr>
<td><code>pickradius</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>rasterized</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>sketch_params</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>snap</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>transform</code></td>
<td><code>Transform</code> instance</td>
</tr>
<tr>
<td><code>url</code></td>
<td>a url string</td>
</tr>
<tr>
<td><code>urls</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>visible</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>zorder</code></td>
<td>any number</td>
</tr>
</tbody>
</table>

See also:

**pcolor()** For an explanation of the grid orientation (*Grid Orientation*) and the expansion of 1-D X and/or Y to 2-D arrays.
Note: In addition to the above described arguments, this function can take a data keyword argument. If such a data argument is given, the following arguments are replaced by data[<arg>]:

- All positional and all keyword arguments.

matplotlib.axes.Axes.spy

Axes.spy(Z, precision=0, marker=None, markersize=None, aspect='equal', origin='upper', **kwargs)
Plot the sparsity pattern on a 2-D array.
spy(Z) plots the sparsity pattern of the 2-D array Z.

Parameters Z : sparse array (n, m)
The array to be plotted.

precision : float, optional, default: 0
If precision is 0, any non-zero value will be plotted; else, values of |Z| > precision will be plotted.
For scipy.sparse.spmatrix instances, there is a special case: if precision is ‘present’, any value present in the array will be plotted, even if it is identically zero.

origin : ["upper", “lower"], optional, default: “upper”
Place the [0,0] index of the array in the upper left or lower left corner of the axes.

aspect : [‘auto’ | ‘equal’ | scalar], optional, default: “equal”
If ‘equal’, and extent is None, changes the axes aspect ratio to match that of the image. If extent is not None, the axes aspect ratio is changed to match that of the extent.
If ‘auto’, changes the image aspect ratio to match that of the axes.
If None, default to rc image.aspect value.

Two plotting styles are available: image or marker. Both are available for full arrays, but only the marker style works for :class:`scipy.sparse.spmatrix` instances.
If *marker* and *markersize* are *None*, an image will be returned and any remaining kwargs are passed to :func:`~matplotlib.pyplot.imshow`; else, a :class:`~matplotlib.lines.Line2D` object will be returned with the value of marker determining the marker type, and any
If *marker* and *markersize* are *None*, useful kwargs include:

* **cmap**
* **alpha**

See also:

`imshow` for image options.

`plot` for plotting options

### 19.1.8 Unstructured Triangles

<table>
<thead>
<tr>
<th><strong>Axes.tripcolor</strong></th>
<th>Create a pseudocolor plot of an unstructured triangular grid.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Axes.triplot</strong></td>
<td>Draw a unstructured triangular grid as lines and/or markers.</td>
</tr>
<tr>
<td><strong>Axes.tricontour</strong></td>
<td>Draw contours on an unstructured triangular grid.</td>
</tr>
<tr>
<td><strong>Axes.tricontourf</strong></td>
<td>Draw contours on an unstructured triangular grid.</td>
</tr>
</tbody>
</table>

`matplotlib.axes.Axes.tripcolor`

**Axes.tripcolor**(*args, **kwargs*)

Create a pseudocolor plot of an unstructured triangular grid.

The triangulation can be specified in one of two ways; either:

```python
tripcolor(triangulation, ...)
```

where triangulation is a `matplotlib.tri.Triangulation` object, or

```python
tripcolor(x, y, ...)
tripcolor(x, y, triangles, ...)
tripcolor(x, y, triangles=triangles, ...)
tripcolor(x, y, mask=mask, ...)
tripcolor(x, y, triangles, mask=mask, ...)
```

in which case a Triangulation object will be created. See `Triangulation` for a explanation of these possibilities.

The next argument must be *C*, the array of color values, either one per point in the triangulation if color values are defined at points, or one per triangle in the triangulation if color values are defined at triangles. If there are the same number of points and triangles in the triangulation it is assumed that color values are defined at points; to force the use of color values at triangles use the kwarg `facecolors`=`C` instead of just *C*. 

19.1. Plotting 701
shading may be ‘flat’ (the default) or ‘gouraud’. If shading is ‘flat’ and C values are defined at points, the color values used for each triangle are from the mean C of the triangle’s three points. If shading is ‘gouraud’ then color values must be defined at points.

The remaining kwargs are the same as for pcolor().

Example:
tripcolor of Delaunay triangulation, gouraud shading
```python
matplotlib.axes.Axes.triplot

Axes.triplot(*args, **kwargs)

Draw a unstructured triangular grid as lines and/or markers.

The triangulation to plot can be specified in one of two ways; either:

```python
triplot(triangulation, ...)
```

where triangulation is a `matplotlib.tri.Triangulation` object, or

```python
triplot(x, y, ...)
triplot(x, y, triangles, ...)  
triplot(x, y, triangles=triangles, ...)  
triplot(x, y, mask=mask, ...)  
triplot(x, y, triangles, mask=mask, ...)
```

in which case a Triangulation object will be created. See `Triangulation` for a explanation of these possibilities.

The remaining args and kwargs are the same as for `plot()`.

Return a list of 2 `Line2D` containing respectively:

- the lines plotted for triangles edges
• the markers plotted for triangles nodes

Example:

triplot of Delaunay triangulation
Matplotlib, Release 2.0.2

matplotlib.axes.Axes.tricontour

Axes.tricontour(*args, **kwargs)
Draw contours on an unstructured triangular grid. tricontour() and tricontourf() draw contour lines and filled contours, respectively. Except as noted, function signatures and return values are the same for both versions.

The triangulation can be specified in one of two ways; either:

```python
tricontour(triangulation, ...)
```

where triangulation is a matplotlib.tri.Triangulation object, or

```python
tricontour(x, y, ...)                # same as tricontour(x, y, Triangulation(...))
tricontour(x, y, triangles, ...)    # pass the triangulation directly
tricontour(x, y, triangles=triangles, ...) # pass the triangulation via a parameter
tricontour(x, y, mask=mask, ...)    # mask any data values with NaN
tricontour(x, y, triangles, mask=mask, ...)
```

in which case a Triangulation object will be created. See Triangulation for a explanation of these possibilities.

The remaining arguments may be:
tricontour(..., Z)

where Z is the array of values to contour, one per point in the triangulation. The level values are chosen automatically.

tricontour(..., Z, N)

contour N automatically-chosen levels.

tricontour(..., Z, V)

draw contour lines at the values specified in sequence V, which must be in increasing order.

tricontourf(..., Z, V)

fill the (len(V)-1) regions between the values in V, which must be in increasing order.

tricontour(Z, **kwargs)

Use keyword args to control colors, linewidth, origin, cmap ... see below for more details.

C = tricontour(...) returns a TriContourSet object.

Optional keyword arguments:

colors: [ None | string | (mpl_colors) ] If None, the colormap specified by cmap will be used.

If a string, like ‘r’ or ‘red’, all levels will be plotted in this color.

If a tuple of matplotlib color args (string, float, rgb, etc), different levels will be plotted in different colors in the order specified.

alpha: float The alpha blending value

cmap: [ None | Colormap ] A cm Colormap instance or None. If cmap is None and colors is None, a default Colormap is used.

norm: [ None | Normalize ] A matplotlib.colors.Normalize instance for scaling data values to colors. If norm is None and colors is None, the default linear scaling is used.

levels [level0, level1, ..., leveln] A list of floating point numbers indicating the level curves to draw, in increasing order; e.g., to draw just the zero contour pass levels=[0]

origin: [ None | ‘upper’ | ‘lower’ | ‘image’ ] If None, the first value of Z will correspond to the lower left corner, location (0,0). If ‘image’, the rc value for image.origin will be used.

This keyword is not active if X and Y are specified in the call to contour.

extent: [ None | (x0,x1,y0,y1) ]

If origin is not None, then extent is interpreted as in matplotlib.pyplot.imshow(): it gives the outer pixel boundaries. In this case, the position of Z[0,0]
is the center of the pixel, not a corner. If \textit{origin} is \textit{None}, then \((x0, y0)\) is the position of \(Z[0,0]\), and \((x1, y1)\) is the position of \(Z[-1,-1]\).

This keyword is not active if \(X\) and \(Y\) are specified in the call to \textit{contour}.

\textit{locator}: [ \textit{None} | \texttt{ticker.Locator subclass} ] If \textit{locator} is \textit{None}, the default \texttt{MaxNLocator} is used. The locator is used to determine the contour levels if they are not given explicitly via the \textit{V} argument.

\textit{extend}: [ ‘neither’ | ‘both’ | ‘min’ | ‘max’ ] Unless this is ‘neither’, contour levels are automatically added to one or both ends of the range so that all data are included. These added ranges are then mapped to the special colormap values which default to the ends of the colormap range, but can be set via \texttt{matplotlib.colors.Colormap.set_under()} and \texttt{matplotlib.colors.Colormap.set_over()} methods.

\textit{xunits}, \textit{yunits}: [ \textit{None} | registered units ] Override axis units by specifying an instance of a \texttt{matplotlib.units.ConversionInterface}.

\text{tricontour-only keyword arguments:}

\textit{linewidths}: [ \textit{None} | number | tuple of numbers ] If \textit{linewidths} is \textit{None}, the default width in \texttt{lines.linewidth} in \texttt{matplotlibrc} is used.

If a number, all levels will be plotted with this linewidth.

If a tuple, different levels will be plotted with different linewidths in the order specified.

\textit{linestyles}: [ \textit{None} | ‘solid’ | ‘dashed’ | ‘dashdot’ | ‘dotted’ ] If \textit{linestyles} is \textit{None}, the ‘solid’ is used.

\textit{linestyles} can also be an iterable of the above strings specifying a set of linestyles to be used. If this iterable is shorter than the number of contour levels it will be repeated as necessary.

If contour is using a monochrome colormap and the contour level is less than 0, then the linestyle specified in \texttt{contour.negative_linestyle} in \texttt{matplotlibrc} will be used.

\text{tricontourf-only keyword arguments:}

\textit{antialiased}: [ \texttt{True} | \texttt{False} ] enable antialiasing

Note: \text{tricontourf} fills intervals that are closed at the top; that is, for boundaries \(z1\) and \(z2\), the filled region is:

\[
z1 < z \leq z2
\]

There is one exception: if the lowest boundary coincides with the minimum value of the \(z\) array, then that minimum value will be included in the lowest interval.

\textbf{Examples:}
Contour plot of Delaunay triangulation
**matplotlib.axes.Axes.tricontourf**

`Axes.tricontourf(*args, **kwargs)`

Draw contours on an unstructured triangular grid. `tricontour()` and `tricontourf()` draw contour lines and filled contours, respectively. Except as noted, function signatures and return values are the same for both versions.

The triangulation can be specified in one of two ways; either:

```
tricontour(triangulation, ...)
```

where triangulation is a `matplotlib.tri.Triangulation` object, or

```
tricontour(x, y, ...)
tricontour(x, y, triangles, ...)
tricontour(x, y, triangles=triangles, ...)
tricontour(x, y, mask=mask, ...)
tricontour(x, y, triangles, mask=mask, ...)
```

in which case a Triangulation object will be created. See `Triangulation` for a explanation of these possibilities.

The remaining arguments may be:
where $Z$ is the array of values to contour, one per point in the triangulation. The level values are chosen automatically.

```python
tricontour(..., Z)
```

draw contour lines at the values specified in sequence $V$, which must be in increasing order.

```python
tricontourf(..., Z, V)
```

fill the $(\text{len}(V)-1)$ regions between the values in $V$, which must be in increasing order.

```python
tricontour(Z, **kwargs)
```

Use keyword args to control colors, linewidth, origin, cmap ... see below for more details.

```
C = tricontour(...) returns a TriContourSet object.
```

Optional keyword arguments:

- `colors`: [ *None* | string | (mpl_colors) ] If *None*, the colormap specified by cmap will be used. If a string, like ‘r’ or ‘red’, all levels will be plotted in this color. If a tuple of matplotlib color args (string, float, rgb, etc), different levels will be plotted in different colors in the order specified.

- `alpha`: float The alpha blending value

- `cmap`: [ *None* | Colormap ] A cm Colormap instance or *None*. If *cmap* is *None* and *colors* is *None*, a default Colormap is used.

- `norm`: [ *None* | Normalize ] A matplotlib.colors.Normalize instance for scaling data values to colors. If *norm* is *None* and *colors* is *None*, the default linear scaling is used.

- `levels` [level0, level1, ..., leveln] A list of floating point numbers indicating the level curves to draw, in increasing order; e.g., to draw just the zero contour pass `levels=[0]`

- `origin`: [ *None* | ‘upper’ | ‘lower’ | ‘image’ ] If *None*, the first value of $Z$ will correspond to the lower left corner, location $(0,0)$. If ‘image’, the rc value for image.origin will be used.

  This keyword is not active if $X$ and $Y$ are specified in the call to contour.

- `extent`: [ *None* | (x0,x1,y0,y1) ]

  If `origin` is not *None*, then `extent` is interpreted as in matplotlib.pyplot.imshow(): it gives the outer pixel boundaries. In this case, the position of $Z[0,0]$
is the center of the pixel, not a corner. If origin is None, then (x0, y0) is the
position of Z[0,0], and (x1, y1) is the position of Z[-1,-1].

This keyword is not active if X and Y are specified in the call to contour.

locator: [ None | ticker.Locator subclass ] If locator is None, the default MaxNLocator
is used. The locator is used to determine the contour levels if they are not given
explicitly via the V argument.

extend: [ ‘neither’ | ‘both’ | ‘min’ | ‘max’ ] Unless this is ‘neither’, contour levels are au-
tomatically added to one or both ends of the range so that all data are included. These
added ranges are then mapped to the special colormap values which default to the
ends of the colormap range, but can be set via matplotlib.colors.Colormap.
set_under() and matplotlib.colors.Colormap.set_over() methods.

xunits, yunits: [ None | registered units ] Override axis units by specifying an instance of
a matplotlib.units.ConversionInterface.

tricontour-only keyword arguments:

linewdths: [ None | number | tuple of numbers ] If linewdths is None, the default width
in lines.linewidth in matplotlibrc is used.

If a number, all levels will be plotted with this linewidth.

If a tuple, different levels will be plotted with different linewidths in the order specified

linestyles: [ None | ‘solid’ | ‘dashed’ | ‘dashdot’ | ‘dotted’ ] If linestyles is None, the
‘solid’ is used.

linestyles can also be an iterable of the above strings specifying a set of linestyles to
be used. If this iterable is shorter than the number of contour levels it will be repeated
as necessary.

If contour is using a monochrome colormap and the contour level is less than 0, then
the linestyle specified in contour.negative_linestyle in matplotlibrc will be
used.

tricontourf-only keyword arguments:

antialiased: [ True | False ] enable antialiasing

Note: tricontourf fills intervals that are closed at the top; that is, for boundaries z1 and z2, the filled
region is:

z1 < z <= z2

There is one exception: if the lowest boundary coincides with the minimum value of the z array, then
that minimum value will be included in the lowest interval.

Examples:
Contour plot of Delaunay triangulation
19.1.9 Text and Annotations

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Axes.annotate</td>
<td>Annotate the point $xy$ with text $s$.</td>
</tr>
<tr>
<td>Axes.text</td>
<td>Add text to the axes.</td>
</tr>
<tr>
<td>Axes.table</td>
<td>Add a table to the current axes.</td>
</tr>
<tr>
<td>Axes.arrow</td>
<td>Add an arrow to the axes.</td>
</tr>
</tbody>
</table>

**matplotlib.axes.Axes.annotate**

```python
Axes.annotate(*args, **kwargs)
```

Annotate the point $xy$ with text $s$.

Additional kwargs are passed to Text.

**Parameters**

- $s$ : str
  
  The text of the annotation

- $xy$ : iterable
  
  Length 2 sequence specifying the $(x,y)$ point to annotate

- $xytext$ : iterable, optional
Length 2 sequence specifying the \((x,y)\) to place the text at. If None, defaults to \(xy\).

**xycoords : str, Artist, Transform, callable or tuple, optional**

The coordinate system that \(xy\) is given in.

For a **str** the allowed values are:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'figure points'</td>
<td>points from the lower left of the figure</td>
</tr>
<tr>
<td>'figure pixels'</td>
<td>pixels from the lower left of the figure</td>
</tr>
<tr>
<td>'figure fraction'</td>
<td>fraction of figure from lower left</td>
</tr>
<tr>
<td>'axes points'</td>
<td>points from lower left corner of axes</td>
</tr>
<tr>
<td>'axes pixels'</td>
<td>pixels from lower left corner of axes</td>
</tr>
<tr>
<td>'axes fraction'</td>
<td>fraction of axes from lower left</td>
</tr>
<tr>
<td>'data'</td>
<td>use the coordinate system of the object being annotated (default)</td>
</tr>
<tr>
<td>'polar'</td>
<td>((\theta, r)) if not native 'data' coordinates</td>
</tr>
</tbody>
</table>

If a **Artist** object is passed in the units are fraction if it’s bounding box.

If a **Transform** object is passed in use that to transform \(xy\) to screen coordinates.

If a callable it must take a **RendererBase** object as input and return a **Transform** or **Bbox** object.

If a tuple must be length 2 tuple of str, Artist, Transform or callable objects. The first transform is used for the \(x\) coordinate and the second for \(y\).

See **Advanced Annotation** for more details.

Defaults to 'data'

**textcoords : str, Artist, Transform, callable or tuple, optional**

The coordinate system that \(xytext\) is given, which may be different than the coordinate system used for \(xy\).

All \(xycoords\) values are valid as well as the following strings:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'offset points'</td>
<td>offset (in points) from the (xy) value</td>
</tr>
<tr>
<td>'offset pixels'</td>
<td>offset (in pixels) from the (xy) value</td>
</tr>
</tbody>
</table>

defaults to the input of \(xycoords\)

**arrowprops : dict, optional**

If not None, properties used to draw a **FancyArrowPatch** arrow between \(xy\) and \(xytext\).
If `arrowprops` does not contain the key 'arrowstyle' the allowed keys are:

<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>width</td>
<td>the width of the arrow in points</td>
</tr>
<tr>
<td>headwidth</td>
<td>the width of the base of the arrow head in points</td>
</tr>
<tr>
<td>headlength</td>
<td>the length of the arrow head in points</td>
</tr>
<tr>
<td>shrink</td>
<td>fraction of total length to ‘shrink’ from both ends</td>
</tr>
<tr>
<td>?</td>
<td>any key to <code>matplotlib.patches.FancyArrowPatch</code></td>
</tr>
</tbody>
</table>

If the `arrowprops` contains the key 'arrowstyle' the above keys are forbidden. The allowed values of 'arrowstyle' are:

<table>
<thead>
<tr>
<th>Name</th>
<th>Attrs</th>
</tr>
</thead>
<tbody>
<tr>
<td>'-'</td>
<td>None</td>
</tr>
<tr>
<td>'-&gt;'</td>
<td>head_length=0.4,head_width=0.2</td>
</tr>
<tr>
<td>'-['</td>
<td>widthB=1.0,lengthB=0.2,angleB=None</td>
</tr>
<tr>
<td>'-]'</td>
<td>widthA=1.0,widthB=1.0</td>
</tr>
<tr>
<td>'-</td>
<td>&gt;'</td>
</tr>
<tr>
<td>'&lt;-'</td>
<td>head_length=0.4,head_width=0.2</td>
</tr>
<tr>
<td>'&lt;</td>
<td>&gt;'</td>
</tr>
<tr>
<td>'&lt;</td>
<td>-</td>
</tr>
<tr>
<td>'fancy'</td>
<td>head_length=0.4,head_width=0.4, tail_width=0.5</td>
</tr>
<tr>
<td>'simple'</td>
<td>head_length=0.5,head_width=0.5, tail_width=0.2</td>
</tr>
<tr>
<td>'wedge'</td>
<td>tail_width=0.3,shrink_factor=0.5</td>
</tr>
</tbody>
</table>

Valid keys for `FancyArrowPatch` are:

<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>arrowstyle</td>
<td>the arrow style</td>
</tr>
<tr>
<td>connectionstyle</td>
<td>the connection style</td>
</tr>
<tr>
<td>relpos</td>
<td>default is (0.5, 0.5)</td>
</tr>
<tr>
<td>patchA</td>
<td>default is bounding box of the text</td>
</tr>
<tr>
<td>patchB</td>
<td>default is None</td>
</tr>
<tr>
<td>shrinkA</td>
<td>default is 2 points</td>
</tr>
<tr>
<td>shrinkB</td>
<td>default is 2 points</td>
</tr>
<tr>
<td>mutation_scale</td>
<td>default is text size (in points)</td>
</tr>
<tr>
<td>mutation_aspect</td>
<td>default is 1.</td>
</tr>
<tr>
<td>?</td>
<td>any key for <code>matplotlib.patches.PathPatch</code></td>
</tr>
</tbody>
</table>

Defaults to None

**annotation_clip** : bool, optional

Controls the visibility of the annotation when it goes outside the axes area.

If `True`, the annotation will only be drawn when the `xy` is inside the axes. If `False`, the annotation will always be drawn regardless of its position.

The default is `None`, which behave as `True` only if `xycoords` is “data”.

**Returns** `Annotation`
**Axes.text**

Add text to the axes.

Add text in string `s` to axis at location `x`, `y`, data coordinates.

**Parameters**

- **x, y**: scalars
  - data coordinates
- **s**: string
  - text
- **fontdict**: dictionary, optional, default: None
  - A dictionary to override the default text properties. If fontdict is None, the defaults are determined by your rc parameters.
- **withdash**: boolean, optional, default: False
  - Creates a `TextWithDash` instance instead of a `Text` instance.

**Other Parameters**

- **kwargs**: `Text` properties.
  - Other miscellaneous text parameters.

**Examples**

Individual keyword arguments can be used to override any given parameter:

```python
>>> text(x, y, s, fontsize=12)
```

The default transform specifies that text is in data coords, alternatively, you can specify text in axis coords (0,0 is lower-left and 1,1 is upper-right). The example below places text in the center of the axes:

```python
>>> text(0.5, 0.5, 'matplotlib', horizontalalignment='center',
...       verticalalignment='center',
...       transform=ax.transAxes)
```

You can put a rectangular box around the text instance (e.g., to set a background color) by using the keyword `bbox`. `bbox` is a dictionary of `Rectangle` properties. For example:

```python
>>> text(x, y, s, bbox=dict(facecolor='red', alpha=0.5))
```

**Axes.table**

Add a table to the current axes.

Call signature:
Returns a `matplotlib.table.Table` instance. Either `cellText` or `cellColours` must be provided. For finer grained control over tables, use the `Table` class and add it to the axes with `add_table()`.

Thanks to John Gill for providing the class and table.

**kwargs control the `Table` properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>agg_filter</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>alpha</code></td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td><code>animated</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>axes</code></td>
<td>an <code>Axes</code> instance</td>
</tr>
<tr>
<td><code>clip_box</code></td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td><code>clip_on</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>clip_path</code></td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td><code>contains</code></td>
<td>a callable function</td>
</tr>
<tr>
<td><code>figure</code></td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td><code>fontsize</code></td>
<td>a float in points</td>
</tr>
<tr>
<td><code>gid</code></td>
<td>an id string</td>
</tr>
<tr>
<td><code>label</code></td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td><code>path_effects</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>picker</code></td>
<td>[None</td>
</tr>
<tr>
<td><code>rasterized</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>sketch_params</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>snap</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>transform</code></td>
<td><code>Transform</code> instance</td>
</tr>
<tr>
<td><code>url</code></td>
<td>a url string</td>
</tr>
<tr>
<td><code>visible</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>zorder</code></td>
<td>any number</td>
</tr>
</tbody>
</table>

**`matplotlib.axes.Axes.arrow`**

`Axes.arrow(x, y, dx, dy, **kwargs)`

Add an arrow to the axes.

Draws arrow on specified axis from (x, y) to (x + dx, y + dy). Uses FancyArrow patch to construct the arrow.

**Parameters**

- **x** : float
  - X-coordinate of the arrow base
- **y** : float
Y-coordinate of the arrow base

\textbf{dx} : float

Length of arrow along x-coordinate

\textbf{dy} : float

Length of arrow along y-coordinate

Returns a : FancyArrow

patches.FancyArrow object

Other Parameters Optional kwargs (inherited from FancyArrow patch) control the arrow construction and properties:

Constructor arguments

\textit{width} : float (default: \texttt{0.001}) width of full arrow tail

\textit{length_includes_head} : [\texttt{True} \mid \texttt{False}] (default: \texttt{False}) True if head is to be counted in calculating the length.

\textit{head_width} : float or \texttt{None} (default: \texttt{3*width}) total width of the full arrow head

\textit{head_length} : float or \texttt{None} (default: \texttt{1.5 * head_width}) length of arrow head

\textit{shape} : ['full', 'left', 'right'] (default: 'full') draw the left-half, right-half, or full arrow

\textit{overhang} : float (default: \texttt{0}) fraction that the arrow is swept back (0 overhang means triangular shape). Can be negative or greater than one.

\textit{head_starts_at_zero} : [\texttt{True} \mid \texttt{False}] (default: \texttt{False}) if True, the head starts being drawn at coordinate 0 instead of ending at coordinate 0.

Other valid kwargs (inherited from :class:`Patch`) are:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{agg_filter}</td>
<td>unknown</td>
</tr>
<tr>
<td>\textit{alpha}</td>
<td>float or None</td>
</tr>
<tr>
<td>\textit{animated}</td>
<td>[\texttt{True} \mid \texttt{False}]</td>
</tr>
<tr>
<td>\textit{antialiased} or \textit{aa}</td>
<td>[\texttt{True} \mid \texttt{False}] or \texttt{None} for default</td>
</tr>
<tr>
<td>\textit{axes}</td>
<td>an \texttt{Axes} instance</td>
</tr>
<tr>
<td>\textit{capstyle}</td>
<td>['butt' \mid 'round' \mid 'projecting']</td>
</tr>
<tr>
<td>\textit{clip_box}</td>
<td>a \texttt{matplotlib.transforms.Bbox} instance</td>
</tr>
<tr>
<td>\textit{clip_on}</td>
<td>[\texttt{True} \mid \texttt{False}]</td>
</tr>
<tr>
<td>\textit{clip_path}</td>
<td>[(\texttt{Path}, \texttt{Transform})</td>
</tr>
<tr>
<td>\textit{color}</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>\textit{contains}</td>
<td>a callable function</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>edgecolor</strong> or <strong>ec</strong></td>
<td>mpl color spec, None, ‘none’, or ‘auto’</td>
</tr>
<tr>
<td><strong>facecolor</strong> or <strong>fc</strong></td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td><strong>figure</strong></td>
<td>a <em>matplotlib.figure.Figure</em> instance</td>
</tr>
<tr>
<td><strong>fill</strong></td>
<td>[True</td>
</tr>
<tr>
<td><strong>gid</strong></td>
<td>an id string</td>
</tr>
<tr>
<td><strong>hatch</strong></td>
<td>[’/’</td>
</tr>
<tr>
<td><strong>joinstyle</strong></td>
<td>[’miter’</td>
</tr>
<tr>
<td><strong>label</strong></td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td><strong>linestyle</strong> or <strong>ls</strong></td>
<td>[’solid’</td>
</tr>
<tr>
<td><strong>linewidth</strong> or <strong>lw</strong></td>
<td>float or None for default</td>
</tr>
<tr>
<td><strong>path_effects</strong></td>
<td>unknown</td>
</tr>
<tr>
<td><strong>picker</strong></td>
<td>[None</td>
</tr>
<tr>
<td><strong>rasterized</strong></td>
<td>[True</td>
</tr>
<tr>
<td><strong>sketch_params</strong></td>
<td>unknown</td>
</tr>
<tr>
<td><strong>snap</strong></td>
<td>unknown</td>
</tr>
<tr>
<td><strong>transform</strong></td>
<td><em>Transform</em> instance</td>
</tr>
<tr>
<td><strong>url</strong></td>
<td>a url string</td>
</tr>
<tr>
<td><strong>visible</strong></td>
<td>[True</td>
</tr>
<tr>
<td><strong>zorder</strong></td>
<td>any number</td>
</tr>
</tbody>
</table>

**Notes**

The resulting arrow is affected by the axes aspect ratio and limits. This may produce an arrow whose head is not square with its stem. To create an arrow whose head is square with its stem, use `annotate()` for example:

```python
ax.annotate('', xy=(0.5, 0.5), xytext=(0, 0),
            arrowprops=dict(arrowstyle="->"))
```
19.1.10 Fields

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Axes.barbs</td>
<td>Plot a 2-D field of barbs.</td>
</tr>
<tr>
<td>Axes.quiver</td>
<td>Plot a 2-D field of arrows.</td>
</tr>
<tr>
<td>Axes.quiverkey</td>
<td>Add a key to a quiver plot.</td>
</tr>
<tr>
<td>Axes.streamplot</td>
<td>Draws streamlines of a vector flow.</td>
</tr>
</tbody>
</table>

**Axes.barbs**(*args, **kw*)

Plot a 2-D field of barbs.

Call signatures:

- barb(U, V, **kw)
- barb(U, V, C, **kw)
- barb(X, Y, U, V, **kw)
- barb(X, Y, U, V, C, **kw)

Arguments:

- \( X, Y \): The x and y coordinates of the barb locations (default is head of barb; see pivot kwarg)
**U, V:** Give the x and y components of the barb shaft

**C:** An optional array used to map colors to the barbs

All arguments may be 1-D or 2-D arrays or sequences. If X and Y are absent, they will be generated as a uniform grid. If U and V are 2-D arrays but X and Y are 1-D, and if len(X) and len(Y) match the column and row dimensions of U, then X and Y will be expanded with `numpy.meshgrid()`.

U, V, C may be masked arrays, but masked X, Y are not supported at present.

Keyword arguments:

**length:** Length of the barb in points; the other parts of the barb are scaled against this. Default is 9

**pivot:** [‘tip’ | ‘middle’] The part of the arrow that is at the grid point; the arrow rotates about this point, hence the name pivot. Default is ‘tip’

**barbcolor:** [color | color sequence] Specifies the color all parts of the barb except any flags. This parameter is analogous to the `edgecolor` parameter for polygons, which can be used instead. However this parameter will override facecolor.

**flagcolor:** [color | color sequence] Specifies the color of any flags on the barb. This parameter is analogous to the `facecolor` parameter for polygons, which can be used instead. However this parameter will override facecolor. If this is not set (and C has not either) then flagcolor will be set to match barbcolor so that the barb has a uniform color. If C has been set, flagcolor has no effect.

**sizes:** A dictionary of coefficients specifying the ratio of a given feature to the length of the barb. Only those values one wishes to override need to be included. These features include:

- ‘spacing’ - space between features (flags, full/half barbs)
- ‘height’ - height (distance from shaft to top) of a flag or full barb
- ‘width’ - width of a flag, twice the width of a full barb
- ‘emptybarb’ - radius of the circle used for low magnitudes

**fill_empty:** A flag on whether the empty barbs (circles) that are drawn should be filled with the flag color. If they are not filled, they will be drawn such that no color is applied to the center. Default is False

**rounding:** A flag to indicate whether the vector magnitude should be rounded when allocating barb components. If True, the magnitude is rounded to the nearest multiple of the half-barb increment. If False, the magnitude is simply truncated to the next lowest multiple. Default is True

**barb_increments:** A dictionary of increments specifying values to associate with different parts of the barb. Only those values one wishes to override need to be included.

- ‘half’ - half barbs (Default is 5)
- ‘full’ - full barbs (Default is 10)
- ‘flag’ - flags (default is 50)
**flip_barb**: Either a single boolean flag or an array of booleans. Single boolean indicates whether the lines and flags should point opposite to normal for all barbs. An array (which should be the same size as the other data arrays) indicates whether to flip for each individual barb. Normal behavior is for the barbs and lines to point right (comes from wind barbs having these features point towards low pressure in the Northern Hemisphere.) Default is False

Barbs are traditionally used in meteorology as a way to plot the speed and direction of wind observations, but can technically be used to plot any two dimensional vector quantity. As opposed to arrows, which give vector magnitude by the length of the arrow, the barbs give more quantitative information about the vector magnitude by putting slanted lines or a triangle for various increments in magnitude, as shown schematically below:

```
| \ / |
| :  |
| :  |
| :  |
| :  |
```

The largest increment is given by a triangle (or “flag”). After those come full lines (barbs). The smallest increment is a half line. There is only, of course, ever at most 1 half line. If the magnitude is small and only needs a single half-line and no full lines or triangles, the half-line is offset from the end of the barb so that it can be easily distinguished from barbs with a single full line. The magnitude for the barb shown above would nominally be 65, using the standard increments of 50, 10, and 5.

Linewidths and edgecolors can be used to customize the barb. Additional `PolyCollection` keyword arguments:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>agg_filter</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>alpha</code></td>
<td>float or None</td>
</tr>
<tr>
<td><code>animated</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>antiалиaised</code> or <code>antiалиased</code></td>
<td>Boolean or sequence of booleans</td>
</tr>
<tr>
<td><code>array</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>axes</code></td>
<td>an <code>Axes</code> instance</td>
</tr>
<tr>
<td><code>clim</code></td>
<td>a length 2 sequence of floats</td>
</tr>
<tr>
<td><code>clip_box</code></td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td><code>clip_on</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>clip_path</code></td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td><code>cmap</code></td>
<td>a colormap or registered colormap name</td>
</tr>
<tr>
<td><code>color</code></td>
<td><code>matplotlib</code> color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td><code>contains</code></td>
<td>a callable function</td>
</tr>
<tr>
<td><code>edgecolor</code> or <code>edgecolors</code></td>
<td><code>matplotlib</code> color spec or sequence of specs</td>
</tr>
<tr>
<td><code>facecolor</code> or <code>facecolors</code></td>
<td><code>matplotlib</code> color spec or sequence of specs</td>
</tr>
<tr>
<td><code>figure</code></td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td><code>gid</code></td>
<td>an id string</td>
</tr>
<tr>
<td><code>hatch</code></td>
<td>['', '/', 'I', 'i', '+' , 'x', 'o', 'O', '.', '*']</td>
</tr>
<tr>
<td><code>label</code></td>
<td>string or anything printable with '%s' conversion.</td>
</tr>
</tbody>
</table>
Table 19.35 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>linestyle or dashes or linstyles</td>
<td>['solid'</td>
</tr>
<tr>
<td>linewidth or linwidths or lw</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>norm</td>
<td>unknown</td>
</tr>
<tr>
<td>offset_position</td>
<td>unknown</td>
</tr>
<tr>
<td>offsets</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>pickradius</td>
<td>unknown</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>urls</td>
<td>unknown</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

Example:
**Note:** In addition to the above described arguments, this function can take a `data` keyword argument. If such a `data` argument is given, the following arguments are replaced by `data[<arg>]`:

- All positional and all keyword arguments.

---

**matplotlib.axes.Axes.quiver**

`Axes.quiver(*args, **kw)`

Plot a 2-D field of arrows.

Call signatures:

```python
quiver(U, V, **kw)
quiver(U, V, C, **kw)
quiver(X, Y, U, V, **kw)
quiver(X, Y, U, V, C, **kw)
```

$U$ and $V$ are the arrow data, $X$ and $Y$ set the location of the arrows, and $C$ sets the color of the arrows. These arguments may be 1-D or 2-D arrays or sequences.

If $X$ and $Y$ are absent, they will be generated as a uniform grid. If $U$ and $V$ are 2-D arrays and $X$ and $Y$ are 1-D, and if `len(X)` and `len(Y)` match the column and row dimensions of $U$, then $X$ and $Y$ will be expanded with `numpy.meshgrid()`.
The default settings auto-scales the length of the arrows to a reasonable size. To change this behavior see the scale and scale_units kwargs.

The defaults give a slightly swept-back arrow; to make the head a triangle, make headaxislength the same as headlength. To make the arrow more pointed, reduce headwidth or increase headlength and headaxislength. To make the head smaller relative to the shaft, scale down all the head parameters. You will probably do best to leave minshaft alone.

linewdiths and edgecolors can be used to customize the arrow outlines.

**Parameters**

- **X**: 1D or 2D array, sequence, optional
  The x coordinates of the arrow locations

- **Y**: 1D or 2D array, sequence, optional
  The y coordinates of the arrow locations

- **U**: 1D or 2D array or masked array, sequence
  The x components of the arrow vectors

- **V**: 1D or 2D array or masked array, sequence
  The y components of the arrow vectors

- **C**: 1D or 2D array, sequence, optional
  The arrow colors

- **units**: [‘width’ | ‘height’ | ‘dots’ | ‘inches’ | ‘x’ | ‘y’ | ‘xy’]
  The arrow dimensions (except for length) are measured in multiples of this unit.
  - ‘width’ or ‘height’: the width or height of the axis
  - ‘dots’ or ‘inches’: pixels or inches, based on the figure dpi
  - ‘x’, ‘y’, or ‘xy’: respectively X, Y, or $\sqrt{X^2 + Y^2}$ in data units

The arrows scale differently depending on the units. For ‘x’ or ‘y’, the arrows get larger as one zooms in; for other units, the arrow size is independent of the zoom state. For ‘width’ or ‘height’, the arrow size increases with the width and height of the axes, respectively, when the window is resized; for ‘dots’ or ‘inches’, resizing does not change the arrows.

- **angles**: [‘uv’ | ‘xy’], array, optional
  Method for determining the angle of the arrows. Default is ‘uv’.
  - ‘uv’: the arrow axis aspect ratio is 1 so that if $U==*V$ the orientation of the arrow on the plot is 45 degrees counter-clockwise from the horizontal axis (positive to the right).
  - ‘xy’: arrows point from (x,y) to (x+u, y+v). Use this for plotting a gradient field, for example.
Alternatively, arbitrary angles may be specified as an array of values in degrees, counter-clockwise from the horizontal axis.

Note: inverting a data axis will correspondingly invert the arrows only with angles='xy'.

**scale**: None, float, optional

Number of data units per arrow length unit, e.g., m/s per plot width; a smaller scale parameter makes the arrow longer. Default is None.

If None, a simple autoscaling algorithm is used, based on the average vector length and the number of vectors. The arrow length unit is given by the `scale_units` parameter.

**scale_units**: ['width', 'height', 'dots', 'inches', 'x', 'y', 'xy'], None, optional

If the `scale` kwarg is None, the arrow length unit. Default is None.

e.g. `scale_units` is 'inches', scale is 2.0, and \((u, v) = (1, 0)\), then the vector will be 0.5 inches long.

If `scale_units` is 'width'/height', then the vector will be half the width/height of the axes.

If `scale_units` is 'x' then the vector will be 0.5 x-axis units. To plot vectors in the x-y plane, with \(u\) and \(v\) having the same units as \(x\) and \(y\), use angles='xy', scale_units='xy', scale=1.

**width**: scalar, optional

Shaft width in arrow units; default depends on choice of units, above, and number of vectors; a typical starting value is about 0.005 times the width of the plot.

**headwidth**: scalar, optional

Head width as multiple of shaft width, default is 3

**headlength**: scalar, optional

Head length as multiple of shaft width, default is 5

**headaxislength**: scalar, optional

Head length at shaft intersection, default is 4.5

**minshaft**: scalar, optional

Length below which arrow scales, in units of head length. Do not set this to less than 1, or small arrows will look terrible! Default is 1

**minlength**: scalar, optional

Minimum length as a multiple of shaft width; if an arrow length is less than this, plot a dot (hexagon) of this diameter instead. Default is 1.

**pivot**: [‘tail’, ‘mid’, ‘middle’, ‘tip’], optional
The part of the arrow that is at the grid point; the arrow rotates about this point, hence the name *pivot*.

**color**: [ color | color sequence ], optional

This is a synonym for the *PolyCollection* facecolor kwarg. If C has been set, *color* has no effect.

**See also:**

*quiverkey* Add a key to a quiver plot

**Notes**

Additional *PolyCollection* keyword arguments:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>agg_filter</em></td>
<td>unknown</td>
</tr>
<tr>
<td><em>alpha</em></td>
<td>float or None</td>
</tr>
<tr>
<td><em>animated</em></td>
<td>[True</td>
</tr>
<tr>
<td><em>antialiased</em></td>
<td>Boolean or sequence of booleans</td>
</tr>
<tr>
<td><em>array</em></td>
<td>unknown</td>
</tr>
<tr>
<td><em>axes</em></td>
<td>an <em>Axes</em> instance</td>
</tr>
<tr>
<td><em>clim</em></td>
<td>a length 2 sequence of floats</td>
</tr>
<tr>
<td><em>clip_box</em></td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td><em>clip_on</em></td>
<td>[True</td>
</tr>
<tr>
<td><em>clip_path</em></td>
<td>[ (Path, Transform)</td>
</tr>
<tr>
<td><em>cmap</em></td>
<td>a colormap or registered colormap name</td>
</tr>
<tr>
<td><em>color</em></td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td><em>contains</em></td>
<td>a callable function</td>
</tr>
<tr>
<td><em>edgecolor</em></td>
<td>matplotlib color spec or sequence of specs</td>
</tr>
<tr>
<td><em>facecolor</em></td>
<td>matplotlib color spec or sequence of specs</td>
</tr>
<tr>
<td><em>figure</em></td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td><em>gid</em></td>
<td>an id string</td>
</tr>
<tr>
<td><em>hatch</em></td>
<td>[ ‘/’</td>
</tr>
<tr>
<td><em>label</em></td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td><em>linestyle</em></td>
<td>[ ‘solid’</td>
</tr>
<tr>
<td><em>linewidth</em></td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td><em>norm</em></td>
<td>unknown</td>
</tr>
<tr>
<td><em>offset_position</em></td>
<td>unknown</td>
</tr>
<tr>
<td><em>offsets</em></td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td><em>path_effects</em></td>
<td>unknown</td>
</tr>
<tr>
<td><em>picker</em></td>
<td>[None</td>
</tr>
<tr>
<td><em>pickradius</em></td>
<td>unknown</td>
</tr>
<tr>
<td><em>rasterized</em></td>
<td>[True</td>
</tr>
<tr>
<td><em>sketch_params</em></td>
<td>unknown</td>
</tr>
</tbody>
</table>
Table 19.36 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>urls</td>
<td>unknown</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

Examples

→ Quiver key, length = 10

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
ax.quiverkey(*args, **kw)
```

matplotlib.axes.Axes.quiverkey

Axes.quiverkey(*args, **kw)

Add a key to a quiver plot.

Call signature:
quiverkey(Q, X, Y, U, label, **kw)

Arguments:

Q: The Quiver instance returned by a call to quiver.
X, Y: The location of the key; additional explanation follows.
U: The length of the key
label: A string with the length and units of the key

Keyword arguments:

coordinates = [‘axes’ | ‘figure’ | ‘data’ | ‘inches’] Coordinate system and units for X, Y: ‘axes’ and ‘figure’ are normalized coordinate systems with 0,0 in the lower left and 1,1 in the upper right; ‘data’ are the axes data coordinates (used for the locations of the vectors in the quiver plot itself); ‘inches’ is position in the figure in inches, with 0,0 at the lower left corner.

color: overrides face and edge colors from Q.

labelpos = [‘N’ | ‘S’ | ‘E’ | ‘W’] Position the label above, below, to the right, to the left of the arrow, respectively.

labelsep: Distance in inches between the arrow and the label. Default is 0.1

labelcolor: defaults to default Text color.

fontproperties: A dictionary with keyword arguments accepted by the FontProperties initializer: family, style, variant, size, weight

Any additional keyword arguments are used to override vector properties taken from Q.

The positioning of the key depends on X, Y, coordinates, and labelpos. If labelpos is ‘N’ or ‘S’, X, Y give the position of the middle of the key arrow. If labelpos is ‘E’, X, Y positions the head, and if labelpos is ‘W’, X, Y positions the tail; in either of these two cases, X, Y is somewhere in the middle of the arrow+label key object.

matplotlib.axes.Axes.streamplot

Axes.streamplot(x, y, u, v, density=1, linewidth=None, color=None, cmap=None, norm=None, arrowsize=1, arrowstyle='->', minlength=0.1, transform=None, zorder=None, start_points=None)

Draws streamlines of a vector flow.

x, y [1d arrays] an evenly spaced grid.

u, v [2d arrays] x and y-velocities. Number of rows should match length of y, and the number of columns should match x.

density [float or 2-tuple] Controls the closeness of streamlines. When density = 1, the domain is divided into a 30x30 grid—density linearly scales this grid. Each cell in the grid can have, at most, one traversing streamline. For different densities in each direction, use [density_x, density_y].
**linewidth** [numeric or 2d array] vary linewidth when given a 2d array with the same shape as velocities.

**color** [matplotlib color code, or 2d array] Streamline color. When given an array with the same shape as velocities, color values are converted to colors using cmap.

**cmap** [Colormap] Colormap used to plot streamlines and arrows. Only necessary when using an array input for color.

**norm** [Normalize] Normalize object used to scale luminance data to 0, 1. If None, stretch (min, max) to (0, 1). Only necessary when color is an array.

**arrowsize** [float] Factor scale arrow size.


**minlength** [float] Minimum length of streamline in axes coordinates.

**start_points**: Nx2 array Coordinates of starting points for the streamlines. In data coordinates, the same as the x and y arrays.

**zorder** [int] any number

Returns:

**stream_container** [StreamplotSet] Container object with attributes

- lines: matplotlib.collections.LineCollection of streamlines
- arrows: collection of matplotlib.patches.FancyArrowPatch objects representing arrows half-way along stream lines.

This container will probably change in the future to allow changes to the colormap, alpha, etc. for both lines and arrows, but these changes should be backward compatible.

### 19.2 Clearing

<table>
<thead>
<tr>
<th><strong>Axes.cla</strong></th>
<th>Clear the current axes.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Axes.clear</strong></td>
<td>clear the axes</td>
</tr>
</tbody>
</table>

#### 19.2.1 matplotlib.axes.Axes.cla

Axes.cla()

Clear the current axes.

#### 19.2.2 matplotlib.axes.Axes.clear

Axes.clear()

clear the axes
19.3 Appearance

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Axes.axis</td>
<td>Set axis properties.</td>
</tr>
<tr>
<td>Axes.set_axis_off</td>
<td>turn off the axis</td>
</tr>
<tr>
<td>Axes.set_axis_on</td>
<td>turn on the axis</td>
</tr>
<tr>
<td>Axes.set_frame_on</td>
<td>Set whether the axes rectangle patch is drawn</td>
</tr>
<tr>
<td>Axes.get_frame_on</td>
<td>Get whether the axes rectangle patch is drawn</td>
</tr>
<tr>
<td>Axes.set_axisbelow</td>
<td>Set whether the axis ticks and gridlines are above or below most</td>
</tr>
<tr>
<td>Axes.get_axisbelow</td>
<td>Get whether axis below is true or not</td>
</tr>
<tr>
<td>Axes.grid</td>
<td>Turn the axes grids on or off.</td>
</tr>
<tr>
<td>Axes.get_axisbgcolor</td>
<td>Deprecated since version 2.0.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Axes.get_facecolor</td>
<td></td>
</tr>
<tr>
<td>Axes.get_fc</td>
<td></td>
</tr>
<tr>
<td>Axes.set_facecolor</td>
<td></td>
</tr>
<tr>
<td>Axes.set_fc</td>
<td></td>
</tr>
<tr>
<td>Axes.set_axisbgcolor</td>
<td>Deprecated since version 2.0.</td>
</tr>
</tbody>
</table>

19.3.1 matplotlib.axes.Axes.axis

Axes.axis(*v, **kwargs)

Set axis properties.

Valid signatures:

```python
xmin, xmax, ymin, ymax = axis()
xmin, xmax, ymin, ymax = axis(list_arg)
xmin, xmax, ymin, ymax = axis(string_arg)
xmin, xmax, ymin, ymax = axis(**kwargs)
```

**Parameters**

- **v**: list of float or {'on', 'off', 'equal', 'tight', 'scaled', 'normal', 'auto', 'image', 'square'}
  
  Optional positional argument
  
  Axis data limits set from a list; or a command relating to axes:
<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘on’</td>
<td>Toggle axis lines and labels on</td>
</tr>
<tr>
<td>‘off’</td>
<td>Toggle axis lines and labels off</td>
</tr>
<tr>
<td>‘equal’</td>
<td>Equal scaling by changing limits</td>
</tr>
<tr>
<td>‘scaled’</td>
<td>Equal scaling by changing box dimensions</td>
</tr>
<tr>
<td>‘tight’</td>
<td>Limits set such that all data is shown</td>
</tr>
<tr>
<td>‘auto’</td>
<td>Automatic scaling, fill rectangle with data</td>
</tr>
<tr>
<td>‘normal’</td>
<td>Same as ‘auto’; deprecated</td>
</tr>
<tr>
<td>‘image’</td>
<td>‘scaled’ with axis limits equal to data limits</td>
</tr>
<tr>
<td>‘square’</td>
<td>Square plot; similar to ‘scaled’, but initially forcing xmax-xmin = ymax-ymin</td>
</tr>
</tbody>
</table>

**emit** : bool, optional

Passed to set_{x,y}lim functions, if observers are notified of axis limit change

**xmin, ymin, xmax, ymax** : float, optional

The axis limits to be set

**Returns xmin, xmax, ymin, ymax** : float

The axis limits

### 19.3.2 matplotlib.axes.Axes.set_axis_off

Axes.set_axis_off()

turn off the axis

### 19.3.3 matplotlib.axes.Axes.set_axis_on

Axes.set_axis_on()

turn on the axis

### 19.3.4 matplotlib.axes.Axes.set_frame_on

Axes.set_frame_on(b)

Set whether the axes rectangle patch is drawn

ACCEPTS: [True | False ]

### 19.3.5 matplotlib.axes.Axes.get_frame_on

Axes.get_frame_on()

Get whether the axes rectangle patch is drawn
19.3.6 matplotlib.axes.Axes.set_axisbelow

```
Axes.set_axisbelow(b)
```

Set whether the axis ticks and gridlines are above or below most artists

**ACCEPTS:** [ True | False | ‘line’ ]

19.3.7 matplotlib.axes.Axes.get_axisbelow

```
Axes.get_axisbelow()
```

Get whether axis below is true or not

19.3.8 matplotlib.axes.Axes.grid

```
Axes.grid(b=None, which='major', axis='both', **kwargs)
```

Turn the axes grids on or off.

- Set the axes grids on or off; `b` is a boolean. (For MATLAB compatibility, `b` may also be a string, ‘on’ or ‘off’.)
- If `b` is `None` and `len(kwargs)==0`, toggle the grid state. If `kwargs` are supplied, it is assumed that you want a grid and `b` is thus set to `True`.
- `which` can be ‘major’ (default), ‘minor’, or ‘both’ to control whether major tick grids, minor tick grids, or both are affected.
- `axis` can be ‘both’ (default), ‘x’, or ‘y’ to control which set of gridlines are drawn.
- `kwargs` are used to set the grid line properties, e.g.:

```
ax.grid(color='r', linestyle='--', linewidth=2)
```

Valid Line2D `kwargs` are

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>agg_filter</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>alpha</code></td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td><code>animated</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>antialiased</code> or <code>aa</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>axes</code></td>
<td>an <code>Axes</code> instance</td>
</tr>
<tr>
<td><code>clip_box</code></td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td><code>clip_on</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>clip_path</code></td>
<td>[ (Path, Transform)</td>
</tr>
<tr>
<td><code>color</code> or <code>c</code></td>
<td>any matplotlib color</td>
</tr>
<tr>
<td><code>contains</code></td>
<td>a callable function</td>
</tr>
<tr>
<td><code>dash_capstyle</code></td>
<td>[‘butt’</td>
</tr>
<tr>
<td><code>dash_joinstyle</code></td>
<td>[‘miter’</td>
</tr>
<tr>
<td><code>dashes</code></td>
<td>sequence of on/off ink in points</td>
</tr>
</tbody>
</table>
### Table 19.39 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>drawstyle</strong></td>
<td>[‘default’</td>
</tr>
<tr>
<td><strong>figure</strong></td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td><strong>fillstyle</strong></td>
<td>[‘full’</td>
</tr>
<tr>
<td><strong>gid</strong></td>
<td>an id string</td>
</tr>
<tr>
<td><strong>label</strong></td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td><strong>linestyle</strong> or <strong>ls</strong></td>
<td>[‘solid’</td>
</tr>
<tr>
<td><strong>linewidth</strong> or <strong>lw</strong></td>
<td>float value in points</td>
</tr>
<tr>
<td><strong>marker</strong></td>
<td>A valid marker style</td>
</tr>
<tr>
<td><strong>markeredgecolor</strong> or <strong>mec</strong></td>
<td>any matplotlib color</td>
</tr>
<tr>
<td><strong>markeredgewidth</strong> or <strong>mew</strong></td>
<td>float value in points</td>
</tr>
<tr>
<td><strong>markerfacecolor</strong> or <strong>mfc</strong></td>
<td>any matplotlib color</td>
</tr>
<tr>
<td><strong>markerfacecoloralt</strong> or <strong>mfcalt</strong></td>
<td>any matplotlib color</td>
</tr>
<tr>
<td><strong>markersize</strong> or <strong>ms</strong></td>
<td>float</td>
</tr>
<tr>
<td><strong>markevery</strong></td>
<td>[None</td>
</tr>
<tr>
<td><strong>path_effects</strong></td>
<td>unknown</td>
</tr>
<tr>
<td><strong>picker</strong></td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td><strong>pickradius</strong></td>
<td>float distance in points</td>
</tr>
<tr>
<td><strong>rasterized</strong></td>
<td>[True</td>
</tr>
<tr>
<td><strong>sketch_params</strong></td>
<td>unknown</td>
</tr>
<tr>
<td><strong>snap</strong></td>
<td>unknown</td>
</tr>
<tr>
<td><strong>solid_capstyle</strong></td>
<td>[‘butt’</td>
</tr>
<tr>
<td><strong>solid_joinstyle</strong></td>
<td>[‘miter’</td>
</tr>
<tr>
<td><strong>transform</strong></td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td><strong>url</strong></td>
<td>a url string</td>
</tr>
<tr>
<td><strong>visible</strong></td>
<td>[True</td>
</tr>
<tr>
<td><strong>xdata</strong></td>
<td>1D array</td>
</tr>
<tr>
<td><strong>ydata</strong></td>
<td>1D array</td>
</tr>
<tr>
<td><strong>zorder</strong></td>
<td>any number</td>
</tr>
</tbody>
</table>

#### 19.3.9 matplotlib.axes.Axes.get_axis_bgcolor

**Axes.get_axis_bgcolor()**

Deprecated since version 2.0: The get_axis_bgcolor function was deprecated in version 2.0. Use get_facecolor instead.

Return the axis background color

#### 19.3.10 matplotlib.axes.Axes.get_facecolor

**Axes.get_facecolor()**
19.3.11 matplotlib.axes.Axes.get_fc

Axes.get_fc()

19.3.12 matplotlib.axes.Axes.set_facecolor

Axes.set_facecolor(color)

19.3.13 matplotlib.axes.Axes.set_fc

Axes.set_fc(color)

19.3.14 matplotlib.axes.Axes.set_axisbgcolor

Axes.set_axisbgcolor(color)

Deprecated since version 2.0: The set_axisbgcolor function was deprecated in version 2.0. Use set_facecolor instead.

set the axes background color

ACCEPTS: any matplotlib color - see colors()

19.4 Property cycle

<table>
<thead>
<tr>
<th>Axes.set_prop_cycle</th>
<th>Set the property cycle for any future plot commands on this Axes.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Axes.set_color_cycle</td>
<td>Set the color cycle for any future plot commands on this Axes.</td>
</tr>
</tbody>
</table>

19.4.1 matplotlib.axes.Axes.set_prop_cycle

Axes.set_prop_cycle(*args, **kwargs)

Set the property cycle for any future plot commands on this Axes.

set_prop_cycle(arg) set_prop_cycle(label, itr) set_prop_cycle(label1=itr1[, label2=itr2[, ...]])

Form 1 simply sets given Cycler object.

Form 2 creates and sets a Cycler from a label and an iterable.

Form 3 composes and sets a Cycler as an inner product of the pairs of keyword arguments. In other words, all of the iterables are cycled simultaneously, as if through zip().
Parameters arg : Cycler

Set the given Cycler. Can also be None to reset to the cycle defined by the current style.

label : str

The property key. Must be a valid Artist property. For example, ‘color’ or ‘linestyle’. Aliases are allowed, such as ‘c’ for ‘color’ and ‘lw’ for ‘linewidth’.

itr : iterable

Finite-length iterable of the property values. These values are validated and will raise a ValueError if invalid.

19.4.2 matplotlib.axes.Axes.set_color_cycle

Axes.set_color_cycle(clist)

Set the color cycle for any future plot commands on this Axes.

clist is a list of mpl color specifiers.

Deprecated since version 1.5.

19.5 Axis / limits

Axes.get_yaxis

Return the YAxis instance

Axes.get_xaxis

Return the XAxis instance

19.5.1 matplotlib.axes.Axes.get_yaxis

Axes.get_yaxis()

Return the YAxis instance

19.5.2 matplotlib.axes.Axes.get_xaxis

Axes.get_xaxis()

Return the XAxis instance

19.5.3 Axis Limits and direction

Axes.invert_xaxis

Invert the x-axis.

Axes.invert_yaxis

Invert the y-axis.

Axes.xaxis_inverted

Returns True if the x-axis is inverted.
Table 19.42 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Axes.yaxis_inverted</code></td>
<td>Returns <code>True</code> if the y-axis is inverted.</td>
</tr>
<tr>
<td><code>Axes.set_xlim</code></td>
<td>Set the data limits for the x-axis.</td>
</tr>
<tr>
<td><code>Axes.set_ylim</code></td>
<td>Set the data limits for the y-axis.</td>
</tr>
<tr>
<td><code>Axes.get_ylim</code></td>
<td>Get the y-axis range.</td>
</tr>
<tr>
<td><code>Axes.get_xlim</code></td>
<td>Get the x-axis range.</td>
</tr>
<tr>
<td><code>Axes.update_datalim</code></td>
<td>Update the data lim bbox with seq of xy tups or equiv.</td>
</tr>
<tr>
<td><code>Axes.update_datalim_bounds</code></td>
<td>Update the datalim to include the given.</td>
</tr>
<tr>
<td><code>Axes.update_datalim_numerix</code></td>
<td>Deprecated since version 2.0.</td>
</tr>
<tr>
<td><code>Axes.set_ybound</code></td>
<td>Set the lower and upper numerical bounds of the y-axis.</td>
</tr>
<tr>
<td><code>Axes.set_xbound</code></td>
<td>Set the lower and upper numerical bounds of the x-axis.</td>
</tr>
<tr>
<td><code>Axes.get_ybound</code></td>
<td>Return y-axis numerical bounds in the form of</td>
</tr>
<tr>
<td><code>Axes.get_xbound</code></td>
<td>Returns the x-axis numerical bounds.</td>
</tr>
</tbody>
</table>

**matplotlib.axes.Axes.invert_xaxis**

```python
Axes.invert_xaxis()
```

Invert the x-axis.

**matplotlib.axes.Axes.invert_yaxis**

```python
Axes.invert_yaxis()
```

Invert the y-axis.

**matplotlib.axes.Axes.xaxis_inverted**

```python
Axes.xaxis_inverted()
```

Returns `True` if the x-axis is inverted.

**matplotlib.axes.Axes.yaxis_inverted**

```python
Axes.yaxis_inverted()
```

Returns `True` if the y-axis is inverted.

**matplotlib.axes.Axes.set_xlim**

```python
Axes.set_xlim(left=None, right=None, emit=True, auto=False, **kw)
```

Set the data limits for the x-axis.

**Parameters**

- `left`: scalar, optional
  - The left xlim (default: None, which leaves the left limit unchanged).
right : scalar, optional

The right xlim (default: None, which leaves the right limit unchanged).

emit : bool, optional

Whether to notify observers of limit change (default: True).

auto : bool or None, optional

Whether to turn on autoscaling of the x-axis. True turns on, False turns off (default action), None leaves unchanged.

xlims : tuple, optional

The left and right xlims may be passed as the tuple (left, right) as the first positional argument (or as the left keyword argument).

Returns xlims : tuple

Returns the new x-axis limits as (left, right).

Notes

The left value may be greater than the right value, in which case the x-axis values will decrease from left to right.

Examples

>>> set_xlim(left, right)
>>> set_xlim((left, right))
>>> left, right = set_xlim(left, right)

One limit may be left unchanged.

>>> set_xlim(right=right_lim)

Limits may be passed in reverse order to flip the direction of the x-axis. For example, suppose x represents the number of years before present. The x-axis limits might be set like the following so 5000 years ago is on the left of the plot and the present is on the right.

>>> set_xlim(5000, 0)

matplotlib.axes.Axes.set_ylim

Axes.set_ylim(bottom=None, top=None, emit=True, auto=False, **kw)

Set the data limits for the y-axis

Parameters bottom : scalar, optional

The bottom ylim (default: None, which leaves the bottom limit unchanged).
**top**: scalar, optional

The top ylim (default: None, which leaves the top limit unchanged).

**emit**: bool, optional

Whether to notify observers of limit change (default: True).

**auto**: bool or None, optional

Whether to turn on autoscaling of the y-axis. True turns on, False turns off (default action), None leaves unchanged.

**ylimits**: tuple, optional

The bottom and top yxlims may be passed as the tuple (bottom, top) as the first positional argument (or as the bottom keyword argument).

**Returns ylimits**: tuple

Returns the new y-axis limits as (bottom, top).

**Notes**

The bottom value may be greater than the top value, in which case the y-axis values will decrease from bottom to top.

**Examples**

```python
>>> set_ylim(bottom, top)
>>> set_ylim((bottom, top))
>>> bottom, top = set_ylim(bottom, top)
```

One limit may be left unchanged.

```python
>>> set_ylim(top=top_lim)
```

Limits may be passed in reverse order to flip the direction of the y-axis. For example, suppose y represents depth of the ocean in m. The y-axis limits might be set like the following so 5000 m depth is at the bottom of the plot and the surface, 0 m, is at the top.

```python
>>> set_ylim(5000, 0)
```
Notes

The y-axis may be inverted, in which case the bottom value will be greater than the top value.

**matplotlib.axes.Axes.get_xlim**

Axes.get_xlim()  
Get the x-axis range  

**Returns xlims** : tuple  

Returns the current x-axis limits as the tuple (left, right).

Notes

The x-axis may be inverted, in which case the left value will be greater than the right value.

**matplotlib.axes.Axes.update_datalim**

Axes.update_datalim(xys, updatex=True, updatey=True)  
Update the data lim bbox with seq of xy tups or equiv. 2-D array

**matplotlib.axes.Axes.update_datalim_bounds**

Axes.update_datalim_bounds(bounds)  
Update the datalim to include the given Bbox bounds

**matplotlib.axes.Axes.update_datalim_numerix**

Axes.update_datalim_numerix(x, y)  
Deprecated since version 2.0: The update_datalim_numerix function was deprecated in version 2.0. Use update_datalim instead.  
Update the data lim bbox with seq of xy tups

**matplotlib.axes.Axes.set_ybound**

Axes.set_ybound(lower=None, upper=None)  
Set the lower and upper numerical bounds of the y-axis. This method will honor axes inversion regardless of parameter order. It will not change the _autoscaleYon attribute.
matplotlib.axes.Axes.set_xbound

**Axes.set_xbound(lower=None, upper=None)**

Set the lower and upper numerical bounds of the x-axis. This method will honor axes inversion regardless of parameter order. It will not change the _autoscaleXon attribute.

matplotlib.axes.Axes.get_ybound

**Axes.get_ybound()**

Return y-axis numerical bounds in the form of lowerBound < upperBound

matplotlib.axes.Axes.get_xbound

**Axes.get_xbound()**

Returns the x-axis numerical bounds

This always returns:

lowerBound < upperBound

**Returns lowerBound, upperBound : float**

19.5.4 Axis Labels, title, and legend

<table>
<thead>
<tr>
<th>Axes.get_xlabel</th>
<th>Get the xlabel text string.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Axes.get_ylabel</td>
<td>Get the ylabel text string.</td>
</tr>
<tr>
<td>Axes.set_xlabel</td>
<td>Set the label for the xaxis.</td>
</tr>
<tr>
<td>Axes.set_ylabel</td>
<td>Set the label for the yaxis</td>
</tr>
<tr>
<td>Axes.set_title</td>
<td>Set a title for the axes.</td>
</tr>
<tr>
<td>Axes.get_title</td>
<td>Get an axes title.</td>
</tr>
<tr>
<td>Axes.legend</td>
<td>Places a legend on the axes.</td>
</tr>
<tr>
<td>Axes.get_legend</td>
<td>Return the legend.Legend instance, or None if no legend is defined</td>
</tr>
<tr>
<td>Axes.get_legend_handles_labels</td>
<td>Return handles and labels for legend</td>
</tr>
</tbody>
</table>

matplotlib.axes.Axes.get_xlabel

**Axes.get_xlabel()**

Get the xlabel text string.

matplotlib.axes.Axes.get_ylabel

**Axes.get_ylabel()**

Get the ylabel text string.
matplotlib.axes.Axes.set_xlabel

`Axes.set_xlabel(xlabel, fontdict=None, labelpad=None, **kwargs)`

Set the label for the xaxis.

**Parameters**
- `xlabel` : string
  - x label
- `labelpad` : scalar, optional, default: None
  - spacing in points between the label and the x-axis

**Other Parameters**
- `kwargs` : Text properties

See also:

- [text](#) for information on how override and the optional args work

matplotlib.axes.Axes.set_ylabel

`Axes.set_ylabel(ylabel, fontdict=None, labelpad=None, **kwargs)`

Set the label for the yaxis

**Parameters**
- `ylabel` : string
  - y label
- `labelpad` : scalar, optional, default: None
  - spacing in points between the label and the x-axis

**Other Parameters**
- `kwargs` : Text properties

See also:

- [text](#) for information on how override and the optional args work

matplotlib.axes.Axes.set_title

`Axes.set_title(label, fontdict=None, loc='center', **kwargs)`

Set a title for the axes.

Set one of the three available axes titles. The available titles are positioned above the axes in the center, flush with the left edge, and flush with the right edge.

**Parameters**
- `label` : str
  - Text to use for the title
- `fontdict` : dict
  - A dictionary controlling the appearance of the title text, the default `fontdict` is:
{'fontsize': rcParams['axes.titlesize'],
'fontweight': rcParams['axes.titleweight'],
'verticalalignment': 'baseline',
'horizontalalignment': 'loc'}

loc : {‘center’, ‘left’, ‘right’}, str, optional
Which title to set, defaults to ‘center’

Returns text : Text
The matplotlib text instance representing the title

Other Parameters kwarg : text properties
Other keyword arguments are text properties, see Text for a list of valid text properties.

matplotlib.axes.Axes.get_title

Axes.get_title(loc=’center’)
Get an axes title.

Get one of the three available axes titles. The available titles are positioned above the axes in the
center, flush with the left edge, and flush with the right edge.

Parameters loc : {‘center’, ‘left’, ‘right’}, str, optional
Which title to get, defaults to ‘center’

Returns title: str
The title text string.

matplotlib.axes.Axes.legend

Axes.legend(*args, **kwargs)
Places a legend on the axes.

To make a legend for lines which already exist on the axes (via plot for instance), simply call this
function with an iterable of strings, one for each legend item. For example:

```python
ax.plot([1, 2, 3])
ax.legend([A simple line])
```

However, in order to keep the “label” and the legend element instance together, it is preferable to
specify the label either at artist creation, or by calling the set_label() method on the artist:

```python
line, = ax.plot([1, 2, 3], label='Inline label')
# Overwrite the label by calling the method.
line.set_label('Label via method')
ax.legend()
```
Specific lines can be excluded from the automatic legend element selection by defining a label starting with an underscore. This is default for all artists, so calling `legend()` without any arguments and without setting the labels manually will result in no legend being drawn.

For full control of which artists have a legend entry, it is possible to pass an iterable of legend artists followed by an iterable of legend labels respectively:

```
legend((line1, line2, line3), ('label1', 'label2', 'label3'))
```

**Parameters**

`loc` : int or string or pair of floats, default: ‘upper right’

The location of the legend. Possible codes are:

<table>
<thead>
<tr>
<th>Location String</th>
<th>Location Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘best’</td>
<td>0</td>
</tr>
<tr>
<td>‘upper right’</td>
<td>1</td>
</tr>
<tr>
<td>‘upper left’</td>
<td>2</td>
</tr>
<tr>
<td>‘lower left’</td>
<td>3</td>
</tr>
<tr>
<td>‘lower right’</td>
<td>4</td>
</tr>
<tr>
<td>‘right’</td>
<td>5</td>
</tr>
<tr>
<td>‘center left’</td>
<td>6</td>
</tr>
<tr>
<td>‘center right’</td>
<td>7</td>
</tr>
<tr>
<td>‘lower center’</td>
<td>8</td>
</tr>
<tr>
<td>‘upper center’</td>
<td>9</td>
</tr>
<tr>
<td>‘center’</td>
<td>10</td>
</tr>
</tbody>
</table>

Alternatively can be a 2-tuple giving $x$, $y$ of the lower-left corner of the legend in axes coordinates (in which case `bbox_to_anchor` will be ignored).

`bbox_to_anchor` : `matplotlib.transforms.BboxBase` instance or tuple of floats

Specify any arbitrary location for the legend in `bbox_transform` coordinates (default Axes coordinates).

For example, to put the legend’s upper right hand corner in the center of the axes the following keywords can be used:

```
loc='upper right', bbox_to_anchor=(0.5, 0.5)
```

`ncol` : integer

The number of columns that the legend has. Default is 1.

`prop` : None or `matplotlib.font_manager.FontProperties` or dict

The font properties of the legend. If None (default), the current `matplotlib.rcParams` will be used.

`fontsize` : int or float or {'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large'}

Controls the font size of the legend. If the value is numeric the size will be the absolute font size in points. String values are relative to the current default font size. This argument is only used if `prop` is not specified.
numpoints : None or int

The number of marker points in the legend when creating a legend entry for a line/\texttt{matplotlib.lines.Line2D}. Default is \texttt{None} which will take the value from the \texttt{legend.numpoints rcParam}.

scatterpoints : None or int

The number of marker points in the legend when creating a legend entry for a scatter plot/\texttt{matplotlib.collections.PathCollection}. Default is \texttt{None} which will take the value from the \texttt{legend.scatterpoints rcParam}.

scatteryoffsets : iterable of floats

The vertical offset (relative to the font size) for the markers created for a scatter plot legend entry. 0.0 is at the base the legend text, and 1.0 is at the top. To draw all markers at the same height, set to [0.5]. Default [0.375, 0.5, 0.3125].

markerscale : None or int or float

The relative size of legend markers compared with the originally drawn ones. Default is \texttt{None} which will take the value from the \texttt{legend.markerscale rcParam}.

markerfirst : bool

if \texttt{True}, legend marker is placed to the left of the legend label if \texttt{False}, legend marker is placed to the right of the legend label

frameon : None or bool

Control whether the legend should be drawn on a patch (frame). Default is \texttt{None} which will take the value from the \texttt{legend.frameon rcParam}.

fancybox : None or bool

Control whether round edges should be enabled around the \texttt{FancyBoxPatch} which makes up the legend’s background. Default is \texttt{None} which will take the value from the \texttt{legend.fancybox rcParam}.

shadow : None or bool

Control whether to draw a shadow behind the legend. Default is \texttt{None} which will take the value from the \texttt{legend.shadow rcParam}.

framealpha : None or float

Control the alpha transparency of the legend’s background. Default is \texttt{None} which will take the value from the \texttt{legend.framealpha rcParam}.

facecolor : None or “inherit” or a color spec

Control the legend’s background color. Default is \texttt{None} which will take the value from the \texttt{legend.facecolor rcParam}. If "inherit", it will take the \texttt{axes.facecolor rcParam}.

edgecolor : None or “inherit” or a color spec
Control the legend’s background patch edge color. Default is None which will take the value from the legend.edgecolor $rcParam$. If "inherit", it will take the axes.edgecolor $rcParam$.

**mode**: {“expand”, None}

If mode is set to "expand" the legend will be horizontally expanded to fill the axes area (or bbox_to_anchor if defines the legend’s size).

**bbox_transform**: None or *matplotlib.transforms.Transform*

The transform for the bounding box (bbox_to_anchor). For a value of None (default) the Axes’ transAxes transform will be used.

**title**: str or None

The legend’s title. Default is no title (None).

**borderpad**: float or None

The fractional whitespace inside the legend border. Measured in font-size units. Default is None which will take the value from the legend.borderpad $rcParam$.

**labelspacing**: float or None

The vertical space between the legend entries. Measured in font-size units. Default is None which will take the value from the legend.labelspacing $rcParam$.

**handlelength**: float or None

The length of the legend handles. Measured in font-size units. Default is None which will take the value from the legend.handlelength $rcParam$.

**handletextpad**: float or None

The pad between the legend handle and text. Measured in font-size units. Default is None which will take the value from the legend.handletextpad $rcParam$.

**borderaxespad**: float or None

The pad between the axes and legend border. Measured in font-size units. Default is None which will take the value from the legend.borderaxespad $rcParam$.

**columnspacing**: float or None

The spacing between columns. Measured in font-size units. Default is None which will take the value from the legend.columnspacing $rcParam$.

**handler_map**: dict or None

The custom dictionary mapping instances or types to a legend handler. This handler_map updates the default handler map found at matplotlib.legend.Legend.get_legend_handler_map().
Notes

Not all kinds of artist are supported by the legend command. See *Legend guide* for details.

Examples

```python
matplotlib.axes.Axes.get_legend

Axes.get_legend()
Return the legend.Legend instance, or None if no legend is defined

matplotlib.axes.Axes.get_legend_handles_labels

Axes.get_legend_handles_labels(legend_handler_map=None)
Return handles and labels for legend
ax.legend() is equivalent to

h, l = ax.get_legend_handles_labels()
ax.legend(h, l)
```
19.5.5 Axis scales

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Axes.set_xscale</code></td>
<td>Set the x-axis scale</td>
</tr>
<tr>
<td><code>Axes.get_xscale</code></td>
<td>Return the x-axis scale string: linear, log, logit, symlog</td>
</tr>
<tr>
<td><code>Axes.get_yscale</code></td>
<td>Return the y-axis scale string: linear, log, logit, symlog</td>
</tr>
<tr>
<td><code>Axes.set_yscale</code></td>
<td>Set the y-axis scale</td>
</tr>
</tbody>
</table>

### `matplotlib.axes.Axes.set_xscale`

**Syntax**

`Axes.set_xscale(value, **kwargs)`

Set the x-axis scale

Set the scaling of the x-axis: ‘linear’ | ‘log’ | ‘logit’ | ‘symlog’

**Accepts:** ‘linear’ | ‘log’ | ‘logit’ | ‘symlog’

**Different kwargs are accepted, depending on the scale:** ‘linear’

- ‘log’
  - `basex/basey`: The base of the logarithm
  - `nonposx/nonposy`: [‘mask’ | ‘clip’ ] non-positive values in x or y can be masked as invalid, or clipped to a very small positive number
  - `subsx/subsy`: Where to place the subticks between each major tick. Should be a sequence of integers. For example, in a log10 scale: [2, 3, 4, 5, 6, 7, 8, 9]
    will place 8 logarithmically spaced minor ticks between each major tick.

- ‘logit’
  - `nonpos`: [‘mask’ | ‘clip’ ] values beyond ]0, 1[ can be masked as invalid, or clipped to a number very close to 0 or 1

- ‘symlog’
  - `basex/basey`: The base of the logarithm
  - `linthreshx/linthreshy`: The range (-x, x) within which the plot is linear (to avoid having the plot go to infinity around zero).
  - `subsx/subsy`: Where to place the subticks between each major tick. Should be a sequence of integers. For example, in a log10 scale: [2, 3, 4, 5, 6, 7, 8, 9]
    will place 8 logarithmically spaced minor ticks between each major tick.
  - `linscalex/linscaley`: This allows the linear range (-linthresh to linthresh) to be stretched relative to the logarithmic range. Its value is the number of decades to use for each half of the linear range. For example, when `linscale == 1.0` (the default), the space used for the positive and negative halves of the linear range will be equal to one decade in the logarithmic range.
**mpllib.axes.Axes.get_xscale**

Axes.get_xscale()

Return the xaxis scale string: linear, log, logit, symlog

**mpllib.axes.Axes.get_yscale**

Axes.get_yscale()

Return the yaxis scale string: linear, log, logit, symlog

**mpllib.axes.Axes.set_yscale**

Axes.yscale(value, **kwargs)

Set the y-axis scale

Set the scaling of the y-axis: ‘linear’ | ‘log’ | ‘logit’ | ‘symlog’

ACCEPTS: ['linear' | 'log' | 'logit' | 'symlog']

**Different kwargs are accepted, depending on the scale:**

- **‘log’**
  - *base* / *base*:
    - The base of the logarithm
  - *nonposx* / *nonposy*:
    - ['mask' | 'clip'] non-positive values in x or y can be masked as invalid, or clipped to a very small positive number
  - *subsx* / *subsy*:
    - Where to place the subticks between each major tick. Should be a sequence of integers. For example, in a log10 scale: [2, 3, 4, 5, 6, 7, 8, 9] will place 8 logarithmically spaced minor ticks between each major tick.

- **‘logit’**
  - *nonpos*:
    - ['mask' | 'clip'] values beyond ]0, 1[ can be masked as invalid, or clipped to a number very close to 0 or 1

- **‘symlog’**
  - *base* / *base*:
    - The base of the logarithm
  - *linthreshx* / *linthreshy*:
    - The range (-x, x) within which the plot is linear (to avoid having the plot go to infinity around zero).
  - *subsx* / *subsy*:
    - Where to place the subticks between each major tick. Should be a sequence of integers. For example, in a log10 scale: [2, 3, 4, 5, 6, 7, 8, 9] will place 8 logarithmically spaced minor ticks between each major tick.
  - *linscalex* / *linscaley*:
    - This allows the linear range (-*linthresh* to *linthresh*) to be stretched relative to the logarithmic range. Its value is the number of decades to use for each half of the linear range. For example, when *linscale* == 1.0 (the default), the space
used for the positive and negative halves of the linear range will be equal to one decade in the logarithmic range.

### 19.5.6 Autoscaling and margins

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Axes.use_sticky_edges</td>
<td>When autoscaling, whether to obey all Artist.sticky_edges.</td>
</tr>
<tr>
<td>Axes.margins</td>
<td>Set or retrieve autoscaling margins.</td>
</tr>
<tr>
<td>Axes.set_xmargin</td>
<td>Set padding of X data limits prior to autoscaling.</td>
</tr>
<tr>
<td>Axes.set_ymargin</td>
<td>Set padding of Y data limits prior to autoscaling.</td>
</tr>
<tr>
<td>Axes.relim</td>
<td>Recompute the data limits based on current artists.</td>
</tr>
<tr>
<td>Axes.autoscale</td>
<td>Autoscale the axis view to the data (toggle).</td>
</tr>
<tr>
<td>Axes.autoscale_view</td>
<td>Autoscale the view limits using the data limits.</td>
</tr>
<tr>
<td>Axes.get_autoscale_on</td>
<td>Get whether autoscaling is applied for both axes on plot commands</td>
</tr>
<tr>
<td>Axes.set_autoscale_on</td>
<td>Set whether autoscaling is applied on plot commands</td>
</tr>
<tr>
<td>Axes.get_autoscalex_on</td>
<td>Get whether autoscaling for the x-axis is applied on plot commands</td>
</tr>
<tr>
<td>Axes.set_autoscalex_on</td>
<td>Set whether autoscaling for the x-axis is applied on plot commands</td>
</tr>
<tr>
<td>Axes.get_autoscaley_on</td>
<td>Get whether autoscaling for the y-axis is applied on plot commands</td>
</tr>
<tr>
<td>Axes.set_autoscaley_on</td>
<td>Set whether autoscaling for the y-axis is applied on plot commands</td>
</tr>
</tbody>
</table>

**matplotlib.axes.Axes.use_sticky_edges**

**Axes.use_sticky_edges**

When autoscaling, whether to obey all Artist.sticky_edges.

Default is True.

Setting this to False ensures that the specified margins will be applied, even if the plot includes an image, for example, which would otherwise force a view limit to coincide with its data limit.

The changing this property does not change the plot until autoscale or autoscale_view is called.

**matplotlib.axes.Axes.margins**

**Axes.margins(*args, **kw)**

Set or retrieve autoscaling margins.

signatures:

```python
margins()
```

returns xmargin, ymargin
Matplotlib, Release 2.0.2

All three forms above set the xmargin and ymargin parameters. All keyword parameters are optional. A single argument specifies both xmargin and ymargin. The `tight` parameter is passed to `autoscale_view()`, which is executed after a margin is changed; the default here is `True`, on the assumption that when margins are specified, no additional padding to match tick marks is usually desired. Setting `tight` to `None` will preserve the previous setting.

Specifying any margin changes only the autoscaling; for example, if `xmargin` is not `None`, then `xmargin` times the X data interval will be added to each end of that interval before it is used in autoscaling.

```
matplotlib.axes.Axes.set_xmargin
```

Axes.set_xmargin(m)
Set padding of X data limits prior to autoscaling.

$m$ times the data interval will be added to each end of that interval before it is used in autoscaling.

accepts: float in range 0 to 1

```
matplotlib.axes.Axes.set_ymargin
```

Axes.set_ymargin(m)
Set padding of Y data limits prior to autoscaling.

$m$ times the data interval will be added to each end of that interval before it is used in autoscaling.

accepts: float in range 0 to 1

```
matplotlib.axes.Axes.relim
```

Axes.relim(visibility_only=False)
Recompute the data limits based on current artists. If you want to exclude invisible artists from the calculation, set `visibility_only=True`

At present, `Collection` instances are not supported.

```
matplotlib.axes.Axes.autoscale
```

Axes.autoscale(enable=True, axis='both', tight=None)
Autoscale the axis view to the data (toggle).
Convenience method for simple axis view autoscaling. It turns autoscaling on or off, and then, if autoscaling for either axis is on, it performs the autoscaling on the specified axis or axes.

**enable:** [True | False | None] True (default) turns autoscaling on, False turns it off. None leaves the autoscaling state unchanged.

**axis:** ['x' | 'y' | 'both'] which axis to operate on; default is ‘both’

**tight:** [True | False | None] If True, set view limits to data limits; if False, let the locator and margins expand the view limits; if None, use tight scaling if the only artist is an image, otherwise treat tight as False. The tight setting is retained for future autoscaling until it is explicitly changed.

Returns None.

```python
matplotlib.axes.Axes.autoscale_view
```

**Axes.autoscale_view(tight=None, scalex=True, scaley=True)**

Autoscale the view limits using the data limits. You can selectively autoscale only a single axis, e.g., the xaxis by setting scaley to False. The autoscaling preserves any axis direction reversal that has already been done.

If tight is False, the axis major locator will be used to expand the view limits if rc-Params['axes.autolimit_mode'] is ‘round_numbers’. Note that any margins that are in effect will be applied first, regardless of whether tight is True or False. Specifying tight as True or False saves the setting as a private attribute of the Axes; specifying it as None (the default) applies the previously saved value.

The data limits are not updated automatically when artist data are changed after the artist has been added to an Axes instance. In that case, use `matplotlib.axes.Axes.relim()` prior to calling autoscale_view.

```python
matplotlib.axes.Axes.get_autoscale_on
```

**Axes.get_autoscale_on()**

Get whether autoscaling is applied for both axes on plot commands

```python
matplotlib.axes.Axes.set_autoscale_on
```

**Axes.set_autoscale_on(b)**

Set whether autoscaling is applied on plot commands

accepts: [ True | False ]

```python
matplotlib.axes.Axes.get_autoscalex_on
```

**Axes.get_autoscalex_on()**

Get whether autoscaling for the x-axis is applied on plot commands
**Axes.set_autoscalex_on**

Set whether autoscaling for the x-axis is applied on plot commands

accepts: [ *True* | *False* ]

---

**Axes.get_autoscaley_on**

Get whether autoscaling for the y-axis is applied on plot commands

---

**Axes.set_autoscaley_on**

Set whether autoscaling for the y-axis is applied on plot commands

accepts: [ *True* | *False* ]

---

### 19.5.7 Aspect ratio

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Axes.apply_aspect</td>
<td>Use _aspect() and _adjustable() to modify the axes box or the view limits.</td>
</tr>
<tr>
<td>Axes.get_aspect</td>
<td>aspect</td>
</tr>
<tr>
<td>Axes.get_adjustable</td>
<td>ACCESS: [ ‘box’</td>
</tr>
<tr>
<td>Axes.set_adjustable</td>
<td></td>
</tr>
</tbody>
</table>

---

**Axes.apply_aspect**

Use _aspect() and _adjustable() to modify the axes box or the view limits.

---

**Axes.get_aspect**

---

**Axes.set_aspect**

**Axes.set_aspect**(aspect, adjustable=None, anchor=None)

aspect
<table>
<thead>
<tr>
<th>value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘auto’</td>
<td>automatic; fill position rectangle with data</td>
</tr>
<tr>
<td>‘normal’</td>
<td>same as ‘auto’; deprecated</td>
</tr>
<tr>
<td>‘equal’</td>
<td>same scaling from data to plot units for x and y</td>
</tr>
<tr>
<td>num</td>
<td>a circle will be stretched such that the height is num times the width. aspect=1 is the same as aspect='equal'.</td>
</tr>
</tbody>
</table>

**adjustable**

<table>
<thead>
<tr>
<th>value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘box’</td>
<td>change physical size of axes</td>
</tr>
<tr>
<td>‘datalim’</td>
<td>change xlim or ylim</td>
</tr>
<tr>
<td>‘box-forced’</td>
<td>same as ‘box’, but axes can be shared</td>
</tr>
</tbody>
</table>

‘box’ does not allow axes sharing, as this can cause unintended side effect. For cases when sharing axes is fine, use ‘box-forced’.

**anchor**

<table>
<thead>
<tr>
<th>value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘C’</td>
<td>centered</td>
</tr>
<tr>
<td>‘SW’</td>
<td>lower left corner</td>
</tr>
<tr>
<td>‘S’</td>
<td>middle of bottom edge</td>
</tr>
<tr>
<td>‘SE’</td>
<td>lower right corner</td>
</tr>
<tr>
<td>etc.</td>
<td></td>
</tr>
</tbody>
</table>

Deprecated since version 1.2: the option ‘normal’ for aspect is deprecated. Use ‘auto’ instead.

**matplotlib.axes.Axes.get_adjustable**

Axes.get_adjustable()

**matplotlib.axes.Axes.set_adjustable**

Axes.set_adjustable(adjustable)

ACCEPTS: [‘box’ | ‘datalim’ | ‘box-forced’]

19.5.8 Ticks and tick labels

<table>
<thead>
<tr>
<th>Axes.xaxis_date</th>
<th>Sets up x-axis ticks and labels that treat the x data as dates.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Axes.yaxis_date</td>
<td>Sets up y-axis ticks and labels that treat the y data as dates.</td>
</tr>
<tr>
<td>Axes.get_xmajorticklabels</td>
<td>Get the xtick labels as a list of Text instances.</td>
</tr>
</tbody>
</table>

Continued on next page
### Table 19.47 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Axes.get_xminorticklabels</code></td>
<td>Get the x minor tick labels as a list of <code>matplotlib.text.Text</code> instances.</td>
</tr>
<tr>
<td><code>Axes.get_xticklabels</code></td>
<td>Get the x tick labels as a list of <code>Text</code> instances.</td>
</tr>
<tr>
<td><code>Axes.get_xticklines</code></td>
<td>Get the xtick lines as a list of <code>Line2D</code> instances.</td>
</tr>
<tr>
<td><code>Axes.get_xticks</code></td>
<td>Return the x ticks as a list of locations.</td>
</tr>
<tr>
<td><code>Axes.get_ymajorticklabels</code></td>
<td>Get the major y tick labels as a list of <code>Text</code> instances.</td>
</tr>
<tr>
<td><code>Axes.get_yminorticklabels</code></td>
<td>Get the minor y tick labels as a list of <code>Text</code> instances.</td>
</tr>
<tr>
<td><code>Axes.get_yticklabels</code></td>
<td>Get the ytick labels as a list of <code>Text</code> instances.</td>
</tr>
<tr>
<td><code>Axes.get_yticklines</code></td>
<td>Get the ytick lines as a list of <code>Line2D</code> instances.</td>
</tr>
<tr>
<td><code>Axes.get_yticks</code></td>
<td>Return the y ticks as a list of locations.</td>
</tr>
<tr>
<td><code>Axes.minorticks_off</code></td>
<td>Remove minor ticks from the axes.</td>
</tr>
<tr>
<td><code>Axes.minorticks_on</code></td>
<td>Add autoscaling minor ticks to the axes.</td>
</tr>
<tr>
<td><code>Axes.set_xticklabels</code></td>
<td>Set the xtick labels with list of strings <code>labels</code>.</td>
</tr>
<tr>
<td><code>Axes.set_xticks</code></td>
<td>Set the x ticks with list of <code>ticks</code>.</td>
</tr>
<tr>
<td><code>Axes.set_yticklabels</code></td>
<td>Set the y tick labels with list of strings <code>labels</code>.</td>
</tr>
<tr>
<td><code>Axes.set_yticks</code></td>
<td>Set the y ticks with list of <code>ticks</code>.</td>
</tr>
<tr>
<td><code>Axes.get_xgridlines</code></td>
<td>Get the x grid lines as a list of <code>Line2D</code> instances.</td>
</tr>
<tr>
<td><code>Axes.get_ygridlines</code></td>
<td>Get the y grid lines as a list of <code>Line2D</code> instances.</td>
</tr>
<tr>
<td><code>Axes.ticklabel_format</code></td>
<td>Change the <code>ScalarFormatter</code> used by default for linear axes.</td>
</tr>
<tr>
<td><code>Axes.tick_params</code></td>
<td>Change the appearance of ticks and tick labels.</td>
</tr>
<tr>
<td><code>Axes.locator_params</code></td>
<td>Control behavior of tick locators.</td>
</tr>
</tbody>
</table>

#### matplotlib.axes.Axes.xaxis_date

**Axes.xaxis_date(**\(tz=None\))**

Sets up x-axis ticks and labels that treat the x data as dates.

\(tz\) is a timezone string or `tzinfo` instance. Defaults to rc value.

#### matplotlib.axes.Axes.yaxis_date

**Axes.yaxis_date(**\(tz=None\))**

Sets up y-axis ticks and labels that treat the y data as dates.

\(tz\) is a timezone string or `tzinfo` instance. Defaults to rc value.

#### matplotlib.axes.Axes.get_xmajorticklabels

**Axes.get_xmajorticklabels()**

Get the xtick labels as a list of `Text` instances.
matplotlib.axes.Axes.get_xminorticklabels

Axes.get_xminorticklabels()
Get the x minor tick labels as a list of matplotlib.text.Text instances.

matplotlib.axes.Axes.get_xticklabels

Axes.get_xticklabels(minor=False, which=None)
Get the x tick labels as a list of Text instances.

Parameters

- **minor**: bool
  - If True return the minor ticklabels, else return the major ticklabels
- **which**: None, ('minor', 'major', 'both')
  - Overrides minor.
  - Selects which ticklabels to return

Returns

- **ret**: list
  - List of Text instances.

matplotlib.axes.Axes.get_xticklines

Axes.get_xticklines()
Get the xtick lines as a list of Line2D instances

matplotlib.axes.Axes.get_xticks

Axes.get_xticks(minor=False)
Return the x ticks as a list of locations

matplotlib.axes.Axes.get_ymajorticklabels

Axes.get_ymajorticklabels()
Get the major y tick labels as a list of Text instances.

matplotlib.axes.Axes.get_yminorticklabels

Axes.get_yminorticklabels()
Get the minor y tick labels as a list of Text instances.
**Axes.get_yticklabels**

Get the x tick labels as a list of `Text` instances.

**Parameters**

- `minor` : bool
  
  If True return the minor ticklabels, else return the major ticklabels.

- `which` : None, ('minor', 'major', 'both')
  
  Selects which ticklabels to return.

**Returns**

- `ret` : list
  
  List of `Text` instances.

**Axes.get_yticklines**

Get the ytick lines as a list of `Line2D` instances.

**Axes.get_yticks**

Return the y ticks as a list of locations.

**Axes.minorticks_off**

Remove minor ticks from the axes.

**Axes.minorticks_on**

Add autoscaling minor ticks to the axes.

**Axes.set_xticklabels**

Set the xtick labels with list of strings `labels`

Return a list of axis text instances.

*kwargs* set the `Text` properties. Valid properties are
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>backgroundcolor</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>bbox</td>
<td>FancyBboxPatch prop dict</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>family or fontfamily or fontname or name</td>
<td>[FONTNAME</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fontproperties or font_properties</td>
<td>a matplotlib.font_manager.FontProperties instance</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>horizontalalignment or ha</td>
<td>[ 'center'</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linespacing</td>
<td>float (multiple of font size)</td>
</tr>
<tr>
<td>multialignment</td>
<td>['left'</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>position</td>
<td>(x,y)</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>rotation</td>
<td>[ angle in degrees</td>
</tr>
<tr>
<td>rotation_mode</td>
<td>unknown</td>
</tr>
<tr>
<td>size or fontsize</td>
<td>[size in points</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>stretch or fontstretch</td>
<td>[a numeric value in range 0-1000</td>
</tr>
<tr>
<td>style or fontstyle</td>
<td>[ 'normal'</td>
</tr>
<tr>
<td>text</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>usetex</td>
<td>unknown</td>
</tr>
<tr>
<td>variant or fontvariant</td>
<td>[ 'normal'</td>
</tr>
<tr>
<td>verticalalignment or ma or va</td>
<td>[ 'center'</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>weight or fontweight</td>
<td>[a numeric value in range 0-1000</td>
</tr>
<tr>
<td>wrap</td>
<td>unknown</td>
</tr>
<tr>
<td>x</td>
<td>float</td>
</tr>
<tr>
<td>y</td>
<td>float</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>
ACCEPPTS: sequence of strings

**matplotlib.axes.Axes.set_xticks**

```python
Axes.set_xticks(ticks, minor=False)
```
Set the x ticks with list of `ticks`
ACCEPPTS: sequence of floats

**matplotlib.axes.Axes.set_yticklabels**

```python
Axes.set_yticklabels(labels, fontdict=None, minor=False, **kwargs)
```
Set the y tick labels with list of strings `labels`
Return a list of `Text` instances.

`kwargs` set `Text` properties for the labels. Valid properties are

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>agg_filter</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>alpha</code></td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td><code>animated</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>axes</code></td>
<td>an <code>Axes</code> instance</td>
</tr>
<tr>
<td><code>backgroundcolor</code></td>
<td>any matplotlib color</td>
</tr>
<tr>
<td><code>bbox</code></td>
<td>FancyBboxPatch prop dict</td>
</tr>
<tr>
<td><code>clip_box</code></td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td><code>clip_on</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>clip_path</code></td>
<td>[(Path, Transform)]</td>
</tr>
<tr>
<td><code>color</code></td>
<td>any matplotlib color</td>
</tr>
<tr>
<td><code>contains</code></td>
<td>a callable function</td>
</tr>
<tr>
<td><code>family</code> or <code>fontfamily</code> or <code>fontname</code> or <code>name</code></td>
<td>[FONTPNAME</td>
</tr>
<tr>
<td><code>figure</code></td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td><code>fontproperties</code> or <code>font_properties</code></td>
<td>a <code>matplotlib.font_manager.FontProperties</code> instance</td>
</tr>
<tr>
<td><code>gid</code></td>
<td>an id string</td>
</tr>
<tr>
<td><code>horizontalalignment</code> or <code>ha</code></td>
<td>[‘center’</td>
</tr>
<tr>
<td><code>label</code></td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td><code>linespacing</code></td>
<td>float (multiple of font size)</td>
</tr>
<tr>
<td><code>multialignment</code></td>
<td>[‘left’</td>
</tr>
<tr>
<td><code>path_effects</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>picker</code></td>
<td>[None</td>
</tr>
<tr>
<td><code>position</code></td>
<td>(x,y)</td>
</tr>
<tr>
<td><code>rasterized</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>rotation</code></td>
<td>[ angle in degrees</td>
</tr>
<tr>
<td><code>rotation_mode</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>size</code> or <code>fontsize</code></td>
<td>[size in points</td>
</tr>
<tr>
<td><code>sketch_params</code></td>
<td>unknown</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>snap</strong></td>
<td>unknown</td>
</tr>
<tr>
<td><strong>stretch</strong> or fontstretch</td>
<td>[a numeric value in range 0-1000</td>
</tr>
<tr>
<td><strong>style</strong> or fontstyle</td>
<td>[ ‘normal’</td>
</tr>
<tr>
<td><strong>text</strong></td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td><strong>transform</strong></td>
<td>Transform instance</td>
</tr>
<tr>
<td><strong>url</strong></td>
<td>a url string</td>
</tr>
<tr>
<td><strong>usetex</strong></td>
<td>unknown</td>
</tr>
<tr>
<td><strong>variant</strong> or fontvariant</td>
<td>[ ‘normal’</td>
</tr>
<tr>
<td><strong>verticalalignment</strong> or ma or va</td>
<td>[ ‘center’</td>
</tr>
<tr>
<td><strong>visible</strong></td>
<td>[True</td>
</tr>
<tr>
<td><strong>wrap</strong></td>
<td>unknown</td>
</tr>
<tr>
<td><strong>x</strong></td>
<td>float</td>
</tr>
<tr>
<td><strong>y</strong></td>
<td>float</td>
</tr>
<tr>
<td><strong>zorder</strong></td>
<td>any number</td>
</tr>
</tbody>
</table>

**ACCEPTS**: sequence of strings

**matplotlib.axes.Axes.set_yticks**

```python
Axes.set_yticks(ticks, minor=False)
```

Set the y ticks with list of ticks

**ACCEPTS**: sequence of floats

Keyword arguments:

- **minor**: [False | True] Sets the minor ticks if True

**matplotlib.axes.Axes.get_xgridlines**

```python
Axes.get_xgridlines()
```

Get the x grid lines as a list of Line2D instances

**matplotlib.axes.Axes.get_ygridlines**

```python
Axes.get_ygridlines()
```

Get the y grid lines as a list of Line2D instances
Axes.ticklabel_format(**kwargs)

Change the ScalarFormatter used by default for linear axes.

Optional keyword arguments:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>style</td>
<td>['sci' (or 'scientific')</td>
</tr>
<tr>
<td>sci limits</td>
<td>(m, n), pair of integers; if style is ‘sci’, scientific notation will be used for numbers outside the range $10^m$:sup: to $10^n$:sup:. Use (0,0) to include all numbers.</td>
</tr>
<tr>
<td>use Offset</td>
<td>[True</td>
</tr>
<tr>
<td>axis</td>
<td>['x'</td>
</tr>
<tr>
<td>use Locale</td>
<td>If True, format the number according to the current locale. This affects things such as the character used for the decimal separator. If False, use C-style (English) formatting. The default setting is controlled by the axes.formatter.use_locale rcparam.</td>
</tr>
</tbody>
</table>

Only the major ticks are affected. If the method is called when the ScalarFormatter is not the Formatter being used, an AttributeError will be raised.

Axes.tick_params(axis=’both’, **kwargs)

Change the appearance of ticks and tick labels.

Parameters axis : {'x', 'y', 'both'}, optional

Which axis to apply the parameters to.

Other Parameters axis : {'x', 'y', 'both'}

Axis on which to operate; default is ‘both’.

reset : bool

If True, set all parameters to defaults before processing other keyword arguments. Default is False.

which : {'major', 'minor', 'both'}

Default is ‘major’; apply arguments to which ticks.

direction : {'in', 'out', 'inout'}

Puts ticks inside the axes, outside the axes, or both.

length : float

Tick length in points.
**width** : float

Tick width in points.

**color** : color

Tick color; accepts any mpl color spec.

**pad** : float

Distance in points between tick and label.

**labels** : float or str

Tick label font size in points or as a string (e.g., ‘large’).

**labelcolor** : color

Tick label color; mpl color spec.

**colors** : color

Changes the tick color and the label color to the same value: mpl color spec.

**zorder** : float

Tick and label zorder.

**bottom, top, left, right** : bool or {'on', 'off'}

controls whether to draw the respective ticks.

**labelbottom, labeltop, labelleft, labelright** : bool or {'on', 'off'}

controls whether to draw the respective tick labels.

### Examples

Usage

```python
ax.tick_params(direction='out', length=6, width=2, colors='r')
```

This will make all major ticks be red, pointing out of the box, and with dimensions 6 points by 2 points. Tick labels will also be red.

### matplotlib.axes.Axes.locator_params

**Axes.locator_params**(axis='both', tight=None, **kwargs)

Control behavior of tick locators.

Keyword arguments:

**axis** ['x' | 'y' | 'both'] Axis on which to operate; default is ‘both’.

**tight** [True | False | None] Parameter passed to `autoscale_view()`. Default is None, for no change.
Remaining keyword arguments are passed to directly to the \texttt{set_params()} method.

Typically one might want to reduce the maximum number of ticks and use tight bounds when plotting small subplots, for example:

\begin{verbatim}
ax.locator_params(tight=True, nbins=4)
\end{verbatim}

Because the locator is involved in autoscaling, \texttt{autoscale_view()} is called automatically after the parameters are changed.

This presently works only for the \texttt{MaxNLocator} used by default on linear axes, but it may be generalized.

\section*{19.6 Units}

\begin{Verbatim}
\begin{tabular}{l|l}
\texttt{Axes.convert_xunits} & For artists in an axes, if the xaxis has units support, \\
\texttt{Axes.convert_yunits} & For artists in an axes, if the yaxis has units support, \\
\texttt{Axes.have_units} & Return \texttt{True} if units are set on the \texttt{x} or \texttt{y} axes
\end{tabular}
\end{Verbatim}

\subsection*{19.6.1 \texttt{matplotlib.axes.Axes.convert_xunits}}

\begin{verbatim}
Axes.convert_xunits(x)
\end{verbatim}

For artists in an axes, if the xaxis has units support, convert \texttt{x} using xaxis unit type

\subsection*{19.6.2 \texttt{matplotlib.axes.Axes.convert_yunits}}

\begin{verbatim}
Axes.convert_yunits(y)
\end{verbatim}

For artists in an axes, if the yaxis has units support, convert \texttt{y} using yaxis unit type

\subsection*{19.6.3 \texttt{matplotlib.axes.Axes.have_units}}

\begin{verbatim}
Axes.have_units()
\end{verbatim}

Return \texttt{True} if units are set on the \texttt{x} or \texttt{y} axes

\section*{19.7 Adding Artists}

\begin{Verbatim}
\begin{tabular}{l|l}
\texttt{Axes.add_artist} & Add any \texttt{Artist} to the axes. \\
\texttt{Axes.add_collection} & Add a \texttt{Collection} instance to the axes. \\
\texttt{Axes.add_container} & Add a \texttt{Container} instance to the axes. \\
\texttt{Axes.add_image} & Add a \texttt{AxesImage} to the axes. \\
\texttt{Axes.add_line} & Add a \texttt{Line2D} to the list of plot
\end{tabular}
\end{Verbatim}
Table 19.51 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Axes.add_patch</code></td>
<td>Add a <code>Patch</code> ( p ) to the list of axes patches; the clipbox will be set to the Axes clipping box.</td>
</tr>
<tr>
<td><code>Axes.add_table</code></td>
<td>Add a Table instance to the</td>
</tr>
</tbody>
</table>

19.7.1 matplotlib.axes.Axes.add_artist

`Axes.add_artist(a)`

Add any `Artist` to the axes.

Use `add_artist` only for artists for which there is no dedicated “add” method; and if necessary, use a method such as `update_datalim` or `update_datalim_numerix` to manually update the dataLim if the artist is to be included in autoscaling.

Returns the artist.

19.7.2 matplotlib.axes.Axes.add_collection

`Axes.add_collection(collection, autolim=True)`

Add a `Collection` instance to the axes.

Returns the collection.

19.7.3 matplotlib.axes.Axes.add_container

`Axes.add_container(container)`

Add a `Container` instance to the axes.

Returns the collection.

19.7.4 matplotlib.axes.Axes.add_image

`Axes.add_image(image)`

Add a `AxesImage` to the axes.

Returns the image.

19.7.5 matplotlib.axes.Axes.add_line

`Axes.add_line(line)`

Add a `Line2D` to the list of plot lines

Returns the line.
19.7.6 matplotlib.axes.Axes.add_patch

`Axes.add_patch(p)`

Add a `Patch` \( p \) to the list of axes patches; the clipbox will be set to the Axes clipping box. If the transform is not set, it will be set to `transData`.

Returns the patch.

19.7.7 matplotlib.axes.Axes.add_table

`Axes.add_table(tab)`

Add a `Table` instance to the list of axes tables

Returns the table.

19.8 Twinning

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Axes.twinx()</code></td>
<td>Create a twin Axes sharing the xaxis</td>
</tr>
<tr>
<td><code>Axes.twiny()</code></td>
<td>Create a twin Axes sharing the yaxis</td>
</tr>
<tr>
<td><code>Axes.get_shared_x_axes</code></td>
<td>Return a copy of the shared axes Grouper object for x axes</td>
</tr>
<tr>
<td><code>Axes.get_shared_y_axes</code></td>
<td>Return a copy of the shared axes Grouper object for y axes</td>
</tr>
</tbody>
</table>

19.8.1 matplotlib.axes.Axes.twinx

`Axes.twinx()`

Create a twin Axes sharing the xaxis

create a twin of Axes for generating a plot with a sharex x-axis but independent y axis. The y-axis of self will have ticks on left and the returned axes will have ticks on the right. To ensure tick marks of both axis align, see `LinearLocator`

**Note:** For those who are ‘picking’ artists while using twinx, pick events are only called for the artists in the top-most axes.

19.8.2 matplotlib.axes.Axes.twiny

`Axes.twiny()`

Create a twin Axes sharing the yaxis

create a twin of Axes for generating a plot with a shared y-axis but independent x axis. The x-axis of self will have ticks on bottom and the returned axes will have ticks on the top.
Note: For those who are ‘picking’ artists while using twiny, pick events are only called for the artists in the top-most axes.

19.8.3 matplotlib.axes.Axes.get_shared_x_axes

Axes.get_shared_x_axes()
Return a copy of the shared axes Grouper object for x axes

19.8.4 matplotlib.axes.Axes.get_shared_y_axes

Axes.get_shared_y_axes()
Return a copy of the shared axes Grouper object for y axes

19.9 Axes Position

<table>
<thead>
<tr>
<th>Axes.get_anchor</th>
<th>anchor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Axes.set_anchor</td>
<td>anchor</td>
</tr>
<tr>
<td>Axes.get_axes_locator</td>
<td>return axes_locator</td>
</tr>
<tr>
<td>Axes.set_axes_locator</td>
<td>set axes_locator</td>
</tr>
<tr>
<td>Axes.reset_position</td>
<td>Make the original position the active position</td>
</tr>
<tr>
<td>Axes.get_position</td>
<td>Return the a copy of the axes rectangle as a Bbox</td>
</tr>
<tr>
<td>Axes.set_position</td>
<td>Set the axes position</td>
</tr>
</tbody>
</table>

19.9.1 matplotlib.axes.Axes.get_anchor

Axes.get_anchor()

19.9.2 matplotlib.axes.Axes.set_anchor

Axes.set_anchor(\texttt{anchor})

\texttt{anchor}
<table>
<thead>
<tr>
<th>value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘C’</td>
<td>Center</td>
</tr>
<tr>
<td>‘SW’</td>
<td>bottom left</td>
</tr>
<tr>
<td>‘S’</td>
<td>bottom</td>
</tr>
<tr>
<td>‘SE’</td>
<td>bottom right</td>
</tr>
<tr>
<td>‘E’</td>
<td>right</td>
</tr>
<tr>
<td>‘NE’</td>
<td>top right</td>
</tr>
<tr>
<td>‘N’</td>
<td>top</td>
</tr>
<tr>
<td>‘NW’</td>
<td>top left</td>
</tr>
<tr>
<td>‘W’</td>
<td>left</td>
</tr>
</tbody>
</table>

### 19.9.3 matplotlib.axes.Axes.get_axes_locator

Axes.get_axes_locator()

    return axes_locator

### 19.9.4 matplotlib.axes.Axes.set_axes_locator

Axes.set_axes_locator(locator)

    set axes_locator

    ACCEPT: a callable object which takes an axes instance and renderer and returns a bbox.

### 19.9.5 matplotlib.axes.Axes.reset_position

Axes.reset_position()

    Make the original position the active position

### 19.9.6 matplotlib.axes.Axes.get_position

Axes.get_position(original=False)

    Return the a copy of the axes rectangle as a Bbox

### 19.9.7 matplotlib.axes.Axes.set_position

Axes.set_position(pos, which='both')

    Set the axes position

    The expected shape of pos is:

    pos = [left, bottom, width, height]

    in relative 0,1 coords, or pos can be a Bbox

    There are two position variables: one which is ultimately used, but which may be modified by
    apply_aspect(), and a second which is the starting point for apply_aspect().
Optional keyword arguments: `which`

<table>
<thead>
<tr>
<th>value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘active’</td>
<td>to change the first</td>
</tr>
<tr>
<td>‘original’</td>
<td>to change the second</td>
</tr>
<tr>
<td>‘both’</td>
<td>to change both</td>
</tr>
</tbody>
</table>

19.10 Async/Event based

**Axes.stale**
If the artist is ‘stale’ and needs to be re-drawn for the output to match the internal state of the artist.

**Axes.pchanged**
Fire an event when property changed, calling all of the registered callbacks.

**Axes.add_callback**
Adds a callback function that will be called whenever one of the Artist’s properties changes.

**Axes.remove_callback**
Remove a callback based on its `id`.

19.10.1 matplotlib.axes.Axes.stale

**Axes.stale**
If the artist is ‘stale’ and needs to be re-drawn for the output to match the internal state of the artist.

19.10.2 matplotlib.axes.Axes.pchanged

**Axes.pchanged()**
Fire an event when property changed, calling all of the registered callbacks.

19.10.3 matplotlib.axes.Axes.add_callback

**Axes.add_callback(func)**
Adds a callback function that will be called whenever one of the Artist’s properties changes.

Returns an `id` that is useful for removing the callback with `remove_callback()` later.

19.10.4 matplotlib.axes.Axes.remove_callback

**Axes.remove_callback(oid)**
Remove a callback based on its `id`.

See also:

`add_callback()` For adding callbacks
## 19.11 Interactive

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Axes.can_pan</code></td>
<td>Return <code>True</code> if this axes supports any pan/zoom button functionality.</td>
</tr>
<tr>
<td><code>Axes.can_zoom</code></td>
<td>Return <code>True</code> if this axes supports the zoom box button functionality.</td>
</tr>
<tr>
<td><code>Axes.get_navigate</code></td>
<td>Get whether the axes responds to navigation commands.</td>
</tr>
<tr>
<td><code>Axes.set_navigate</code></td>
<td>Set whether the axes responds to navigation toolbar commands.</td>
</tr>
<tr>
<td><code>Axes.get_navigate_mode</code></td>
<td>Get the navigation toolbar button status: ‘PAN’, ‘ZOOM’, or None.</td>
</tr>
<tr>
<td><code>Axes.set_navigate_mode</code></td>
<td>Set the navigation toolbar button status.</td>
</tr>
<tr>
<td><code>Axes.start_pan</code></td>
<td>Called when a pan operation has started.</td>
</tr>
<tr>
<td><code>Axes.drag_pan</code></td>
<td>Called when the mouse moves during a pan operation.</td>
</tr>
<tr>
<td><code>Axes.end_pan</code></td>
<td>Called when a pan operation completes (when the mouse button is released).</td>
</tr>
<tr>
<td><code>Axes.format_coord</code></td>
<td>Return a format string formatting the <code>x</code>, <code>y</code> coordinate.</td>
</tr>
<tr>
<td><code>Axes.format_cursor_data</code></td>
<td>Return cursor data string formatted.</td>
</tr>
<tr>
<td><code>Axes.format_xdata</code></td>
<td>Return <code>x</code> string formatted.</td>
</tr>
<tr>
<td><code>Axes.format_ydata</code></td>
<td>Return <code>y</code> string formatted.</td>
</tr>
<tr>
<td><code>Axes.hitlist</code></td>
<td>List the children of the artist which contain the mouse event.</td>
</tr>
<tr>
<td><code>Axes.mouseover</code></td>
<td>Return <code>True</code> if the given mouse event (in display coordinates) is over the axes.</td>
</tr>
<tr>
<td><code>Axes.in_axes</code></td>
<td>Trigger pick event.</td>
</tr>
<tr>
<td><code>Axes.pickable</code></td>
<td>Return <code>True</code> if Artist is pickable.</td>
</tr>
<tr>
<td><code>Axes.get_picker</code></td>
<td>Return the picker object used by this artist.</td>
</tr>
<tr>
<td><code>Axes.set_picker</code></td>
<td>Set the epsilon for picking used by this artist.</td>
</tr>
<tr>
<td><code>Axes.set_contains</code></td>
<td>Replace the contains test used by this artist.</td>
</tr>
<tr>
<td><code>Axes.get_contains</code></td>
<td>Return the _contains test used by the artist, or <code>None</code> for default.</td>
</tr>
<tr>
<td><code>Axes.contains</code></td>
<td>Test whether the mouse event occurred in the axes.</td>
</tr>
<tr>
<td><code>Axes.contains_point</code></td>
<td>Returns <code>True</code> if the point (tuple of <code>x</code>, <code>y</code>) is inside the axes (the area defined by its patch).</td>
</tr>
<tr>
<td><code>Axes.get_cursor_data</code></td>
<td>Get the cursor data for a given event.</td>
</tr>
<tr>
<td><code>Axes.get_cursor_props</code></td>
<td>Return the cursor properties as a <code>linewidth</code>, <code>color</code> tuple.</td>
</tr>
<tr>
<td><code>Axes.set_cursor_props</code></td>
<td>Set the cursor property as.</td>
</tr>
</tbody>
</table>

### 19.11.1 matplotlib.axes.Axes.can_pan

The `can_pan` method checks if this axes supports any pan/zoom button functionality. It returns `True` if supported.
19.11.2 matplotlib.axes.Axes.can_zoom

Axes.can_zoom()

Return True if this axes supports the zoom box button functionality.

19.11.3 matplotlib.axes.Axes.get_navigate

Axes.get_navigate()

Get whether the axes responds to navigation commands

19.11.4 matplotlib.axes.Axes.set_navigate

Axes.set_navigate(b)

Set whether the axes responds to navigation toolbar commands

ACCEPETS: [True | False]

19.11.5 matplotlib.axes.Axes.get_navigate_mode

Axes.get_navigate_mode()

Get the navigation toolbar button status: ‘PAN’, ‘ZOOM’, or None

19.11.6 matplotlib.axes.Axes.set_navigate_mode

Axes.set_navigate_mode(b)

Set the navigation toolbar button status;

Warning: this is not a user-API function.

19.11.7 matplotlib.axes.Axes.start_pan

Axes.start_pan(x, y, button)

Called when a pan operation has started.

x, y are the mouse coordinates in display coords. button is the mouse button number:

• 1: LEFT
• 2: MIDDLE
• 3: RIGHT

Note: Intended to be overridden by new projection types.
19.11.8 matplotlib.axes.Axes.drag_pan

Axes.drag_pan(button, key, x, y)
Called when the mouse moves during a pan operation.

button is the mouse button number:
• 1: LEFT
• 2: MIDDLE
• 3: RIGHT

key is a “shift” key

x, y are the mouse coordinates in display coords.

Note: Intended to be overridden by new projection types.

19.11.9 matplotlib.axes.Axes.end_pan

Axes.end_pan()
Called when a pan operation completes (when the mouse button is up.)

Note: Intended to be overridden by new projection types.

19.11.10 matplotlib.axes.Axes.format_coord

Axes.format_coord(x, y)
Return a format string formatting the x, y coord

19.11.11 matplotlib.axes.Axes.format_cursor_data

Axes.format_cursor_data(data)
Return cursor data string formatted.

19.11.12 matplotlib.axes.Axes.format_xdata

Axes.format_xdata(x)
Return x string formatted. This function will use the attribute self.fmt_xdata if it is callable, else will fall back on the xaxis major formatter
19.11.13 matplotlib.axes.Axes.format_ydata

Axes.format_ydata(y)
Return y string formatted. This function will use the fmt_ydata attribute if it is callable, else will fall back on the yaxis major formatter.

19.11.14 matplotlib.axes.Axes.hitlist

Axes.hitlist(event)
List the children of the artist which contain the mouse event event.

19.11.15 matplotlib.axes.Axes.mouseover

Axes.mouseover

19.11.16 matplotlib.axes.Axes.in_axes

Axes.in_axes(mouseevent)
Return True if the given mouseevent (in display coords) is in the Axes

19.11.17 matplotlib.axes.Axes.pick

Axes.pick(*args)
Trigger pick event
Call signature:

pick(mouseevent)

each child artist will fire a pick event if mouseevent is over the artist and the artist has picker set

19.11.18 matplotlib.axes.Axes.pickable

Axes.pickable()
Return True if Artist is pickable.

19.11.19 matplotlib.axes.Axes.get_picker

Axes.get_picker()
Return the picker object used by this artist
19.11.20 matplotlib.axes.Axes.set_picker

Axes.set_picker(picker)

Set the epsilon for picking used by this artist

picker can be one of the following:

- None: picking is disabled for this artist (default)
- A boolean: if True then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist
- A float: if picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if it’s data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event
- A function: if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```python
hit, props = picker(artist, mouseevent)
```

to determine the hit test. if the mouse event is over the artist, return hit=True and props is a dictionary of properties you want added to the PickEvent attributes.

ACCEPTS: [None|float|boolean|callable]

19.11.21 matplotlib.axes.Axes.set_contains

Axes.set_contains(picker)

Replace the contains test used by this artist. The new picker should be a callable function which determines whether the artist is hit by the mouse event:

```python
hit, props = picker(artist, mouseevent)
```

If the mouse event is over the artist, return hit = True and props is a dictionary of properties you want returned with the contains test.

ACCEPTS: a callable function

19.11.22 matplotlib.axes.Axes.get_contains

Axes.get_contains()

Return the _contains test used by the artist, or None for default.

19.11.23 matplotlib.axes.Axes.contains

Axes.contains(mouseevent)

Test whether the mouse event occurred in the axes.
Returns True / False, {} 

19.11.24 matplotlib.axes.Axes.contains_point

Axes.contains_point(point)
Returns True if the point (tuple of x,y) is inside the axes (the area defined by the its patch). A pixel coordinate is required.

19.11.25 matplotlib.axes.Axes.get_cursor_data

Axes.get_cursor_data(event)
Get the cursor data for a given event.

19.11.26 matplotlib.axes.Axes.get_cursor_props

Axes.get_cursor_props()
Return the cursor properties as a (linewidth, color) tuple, where linewidth is a float and color is an RGBA tuple

19.11.27 matplotlib.axes.Axes.set_cursor_props

Axes.set_cursor_props(*args)
Set the cursor property as
Call signature
ax.set_cursor_props(linewidth, color)

or:
ax.set_cursor_props((linewidth, color))

ACCEPTS: a (float, color) tuple

19.12 Children

<table>
<thead>
<tr>
<th>Axes.get_children</th>
<th>return a list of child artists</th>
</tr>
</thead>
<tbody>
<tr>
<td>Axes.get_images</td>
<td>return a list of Axes images contained by the Axes</td>
</tr>
<tr>
<td>Axes.get_lines</td>
<td>Return a list of lines contained by the Axes</td>
</tr>
<tr>
<td>Axes.findobj</td>
<td>Find artist objects.</td>
</tr>
</tbody>
</table>

19.12. Children
19.12.1 matplotlib.axes.Axes.get_children

Axes.get_children()
return a list of child artists

19.12.2 matplotlib.axes.Axes.get_images

Axes.get_images()
return a list of Axes images contained by the Axes

19.12.3 matplotlib.axes.Axes.get_lines

Axes.get_lines()
Return a list of lines contained by the Axes

19.12.4 matplotlib.axes.Axes.findobj

Axes.findobj(match=None, include_self=True)
Find artist objects.
Recursively find all Artist instances contained in self.
match can be
• None: return all objects contained in artist.
• function with signature boolean = match(artist) used to filter matches
• class instance: e.g., Line2D. Only return artists of class type.

If include_self is True (default), include self in the list to be checked for a match.

19.13 Drawing

<table>
<thead>
<tr>
<th>Axes.draw</th>
<th>Draw everything (plot lines, axes, labels)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Axes.draw_artist</td>
<td>This method can only be used after an initial draw which caches the renderer.</td>
</tr>
<tr>
<td>Axes.redraw_in_frame</td>
<td>This method can only be used after an initial draw which caches the renderer.</td>
</tr>
<tr>
<td>Axes.get_renderer_cache</td>
<td>Get zorder value below which artists will be rasterized</td>
</tr>
<tr>
<td>Axes.get_rasterization_zorder</td>
<td>Set zorder value below which artists will be rasterized.</td>
</tr>
<tr>
<td>Axes.get_window_extent</td>
<td>get the axes bounding box in display space; args and</td>
</tr>
<tr>
<td>Axes.get_tightbbox</td>
<td>Return the tight bounding box of the axes.</td>
</tr>
</tbody>
</table>
19.13.1 matplotlib.axes.Axes.draw

Axes.draw(artist, renderer, *args, **kwargs)
Draw everything (plot lines, axes, labels)

19.13.2 matplotlib.axes.Axes.draw_artist

Axes.draw_artist(a)
This method can only be used after an initial draw which caches the renderer. It is used to efficiently update Axes data (axis ticks, labels, etc are not updated)

19.13.3 matplotlib.axes.Axes.redraw_in_frame

Axes.redraw_in_frame()
This method can only be used after an initial draw which caches the renderer. It is used to efficiently update Axes data (axis ticks, labels, etc are not updated)

19.13.4 matplotlib.axes.Axes.get_renderer_cache

Axes.get_renderer_cache()

19.13.5 matplotlib.axes.Axes.get_rasterization_zorder

Axes.get_rasterization_zorder()
Get zorder value below which artists will be rasterized

19.13.6 matplotlib.axes.Axes.set_rasterization_zorder

Axes.set_rasterization_zorder(z)
Set zorder value below which artists will be rasterized. Set to None to disable rasterizing of artists below a particular zorder.

19.13.7 matplotlib.axes.Axes.get_window_extent

Axes.get_window_extent(*args, **kwargs)
get the axes bounding box in display space; args and kwargs are empty

19.13.8 matplotlib.axes.Axes.get_tightbbox

Axes.get_tightbbox(renderer, call_axes_locator=True)
Return the tight bounding box of the axes. The dimension of the Bbox in canvas coordinate.
If `call_axes_locator` is `False`, it does not call the `_axes_locator` attribute, which is necessary to get the correct bounding box. `call_axes_locator==False` can be used if the caller is only interested in the relative size of the tightbbox compared to the axes bbox.

### 19.14 Bulk property manipulation

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Axes.set</code></td>
<td>A property batch setter.</td>
</tr>
<tr>
<td><code>Axes.update</code></td>
<td>Update the properties of this <em>Artist</em> from the dictionary <em>prop</em>.</td>
</tr>
<tr>
<td><code>Axes.properties</code></td>
<td>return a dictionary mapping property name -&gt; value for all Artist props.</td>
</tr>
<tr>
<td><code>Axes.update_from</code></td>
<td>Copy properties from <em>other</em> to <em>self</em>.</td>
</tr>
</tbody>
</table>


```python
Axes.set(**kwargs)
```

A property batch setter. Pass `kwargs` to set properties.

#### 19.14.2 matplotlib.axes.Axes.update

```python
Axes.update(props)
```

Update the properties of this *Artist* from the dictionary *prop*.

#### 19.14.3 matplotlib.axes.Axes.properties

```python
Axes.properties()
```

return a dictionary mapping property name -> value for all Artist props

#### 19.14.4 matplotlib.axes.Axes.update_from

```python
Axes.update_from(other)
```

Copy properties from *other* to *self*.

### 19.15 General Artist Properties

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Axes.set_alpha</code></td>
<td>Set the alpha value used for blending - not supported on all backends.</td>
</tr>
<tr>
<td><code>Axes.set_animated</code></td>
<td>Set the artist’s animation state.</td>
</tr>
<tr>
<td><code>Axes.set_clip_box</code></td>
<td>Set the artist’s clip <em>Bbox</em>.</td>
</tr>
<tr>
<td><code>Axes.set_clip_on</code></td>
<td>Set whether artist uses clipping.</td>
</tr>
</tbody>
</table>

Continued on next page
### Table 19.59 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Axes.set_clip_path</code></td>
<td>Set the artist’s clip path, which may be:</td>
</tr>
<tr>
<td><code>Axes.set_gid</code></td>
<td>Sets the (group) id for the artist</td>
</tr>
<tr>
<td><code>Axes.set_label</code></td>
<td>Set the label to <code>s</code> for auto legend.</td>
</tr>
<tr>
<td><code>Axes.set_url</code></td>
<td>Sets the url for the artist</td>
</tr>
<tr>
<td><code>Axes.set_visible</code></td>
<td>Set the artist’s visibility.</td>
</tr>
<tr>
<td><code>Axes.set_zorder</code></td>
<td>Set the zorder for the artist</td>
</tr>
<tr>
<td><code>Axes.set_rasterized</code></td>
<td>Force rasterized (bitmap) drawing in vector backend output.</td>
</tr>
<tr>
<td><code>Axes.set_sketch_params</code></td>
<td>Sets the sketch parameters.</td>
</tr>
<tr>
<td><code>Axes.set_agg_filter</code></td>
<td>Set agg_filter function.</td>
</tr>
<tr>
<td><code>Axes.set_snap</code></td>
<td>Sets the snap setting which may be:</td>
</tr>
<tr>
<td><code>Axes.set_transform</code></td>
<td>Set the <code>Transform</code> instance used by this artist.</td>
</tr>
<tr>
<td><code>Axes.set_path_effects</code></td>
<td>set path_effects, which should be a list of instances of</td>
</tr>
<tr>
<td><code>Axes.get_agg_filter</code></td>
<td>return filter function to be used for agg filter</td>
</tr>
<tr>
<td><code>Axes.get_sketch_params</code></td>
<td>Returns the sketch parameters for the artist.</td>
</tr>
<tr>
<td><code>Axes.get_alpha</code></td>
<td>Return the alpha value used for blending - not supported on all</td>
</tr>
<tr>
<td><code>Axes.get_animated</code></td>
<td>Return the artist’s animated state</td>
</tr>
<tr>
<td><code>Axes.get_clip_box</code></td>
<td>Return artist clipbox</td>
</tr>
<tr>
<td><code>Axes.get_clip_on</code></td>
<td>Return whether artist uses clipping</td>
</tr>
<tr>
<td><code>Axes.get_clip_path</code></td>
<td>Return artist clip path</td>
</tr>
<tr>
<td><code>Axes.get_gid</code></td>
<td>Returns the group id</td>
</tr>
<tr>
<td><code>Axes.get_label</code></td>
<td>Get the label used for this artist in the legend.</td>
</tr>
<tr>
<td><code>Axes.get_url</code></td>
<td>Returns the url</td>
</tr>
<tr>
<td><code>Axes.get_visible</code></td>
<td>Return the artist’s visibility</td>
</tr>
<tr>
<td><code>Axes.get_zorder</code></td>
<td>Return the Artist’s zorder.</td>
</tr>
<tr>
<td><code>Axes.get_rasterized</code></td>
<td>return True if the artist is to be rasterized</td>
</tr>
<tr>
<td><code>Axes.get_transform</code></td>
<td>Return the <code>Transform</code> instance used by this artist.</td>
</tr>
<tr>
<td><code>Axes.get_snap</code></td>
<td>Returns the snap setting which may be:</td>
</tr>
<tr>
<td><code>Axes.get_path_effects</code></td>
<td>The <code>Axes</code> instance the artist resides in, or <code>None</code>.</td>
</tr>
<tr>
<td><code>Axes.axes</code></td>
<td>Return the <code>Axes</code> instance the artist resides in, or <code>None</code>.</td>
</tr>
<tr>
<td><code>Axes.get_axes</code></td>
<td>Set the <code>Axes</code> instance in which the artist resides, if any.</td>
</tr>
<tr>
<td><code>Axes.set_figure</code></td>
<td>Set the class:<code>Axes</code> figure</td>
</tr>
<tr>
<td><code>Axes.get_figure</code></td>
<td>Return the <code>Figure</code> instance the artist belongs to.</td>
</tr>
</tbody>
</table>

#### 19.15.1 matplotlib.axes.Axes.set_alpha

- **Axes.set_alpha(alpha)**
  
  Set the alpha value used for blending - not supported on all backends.
  
  ACCEPTS: float (0.0 transparent through 1.0 opaque)
19.15.2 matplotlib.axes.Axes.set_animated

Axes.set_animated(b)
Set the artist’s animation state.
ACCEPTS: [True | False]

19.15.3 matplotlib.axes.Axes.set_clip_box

Axes.set_clip_box(clipbox)
Set the artist’s clip Bbox.
ACCEPTS: a matplotlib.transforms.Bbox instance

19.15.4 matplotlib.axes.Axes.set_clip_on

Axes.set_clip_on(b)
Set whether artist uses clipping.
When False artists will be visible out side of the axes which can lead to unexpected results.
ACCEPTS: [True | False]

19.15.5 matplotlib.axes.Axes.set_clip_path

Axes.set_clip_path(path, transform=None)
Set the artist’s clip path, which may be:

• a Patch (or subclass) instance

• a Path instance, in which case an optional Transform instance may be provided, which will be applied to the path before using it for clipping.

• None, to remove the clipping path

For efficiency, if the path happens to be an axis-aligned rectangle, this method will set the clipping box to the corresponding rectangle and set the clipping path to None.
ACCEPTS: [(Path, Transform) | Patch | None]

19.15.6 matplotlib.axes.Axes.set_gid

Axes.set_gid(gid)
Sets the (group) id for the artist
ACCEPTS: an id string
19.15.7 matplotlib.axes.Axes.set_label

Axes.set_label(s)
Set the label to s for auto legend.
ACCEPTS: string or anything printable with ‘%s’ conversion.

19.15.8 matplotlib.axes.Axes.set_url

Axes.set_url(url)
Sets the url for the artist
ACCEPTS: a url string

19.15.9 matplotlib.axes.Axes.set_visible

Axes.set_visible(b)
Set the artist’s visibility.
ACCEPTS: [True | False]

19.15.10 matplotlib.axes.Axes.set_zorder

Axes.set_zorder(level)
Set the zorder for the artist. Artists with lower zorder values are drawn first.
ACCEPTS: any number

19.15.11 matplotlib.axes.Axes.set_rasterized

Axes.set_rasterized(rasterized)
Force rasterized (bitmap) drawing in vector backend output.
Defaults to None, which implies the backend’s default behavior
ACCEPTS: [True | False | None]

19.15.12 matplotlib.axes.Axes.set_sketch_params

Axes.set_sketch_params(scale=None, length=None, randomness=None)
Sets the sketch parameters.

Parameters scale : float, optional
The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is None, or not provided, no sketch filter will be provided.
length : float, optional
The length of the wiggle along the line, in pixels (default 128.0)

randomness : float, optional

The scale factor by which the length is shrunken or expanded (default 16.0)

19.15.13 matplotlib.axes.Axes.set_agg_filter

Axes.set_agg_filter(filter_func)
set agg_filter fuction.

19.15.14 matplotlib.axes.Axes.set_snap

Axes.set_snap(snap)
Sets the snap setting which may be:

- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

19.15.15 matplotlib.axes.Axes.set_transform

Axes.set_transform(t)
Set the Transform instance used by this artist.

ACCEPTS: Transform instance

19.15.16 matplotlib.axes.Axes.set_path_effects

Axes.set_path_effects(path_effects)
set path_effects, which should be a list of instances of matplotlib.path._Base class or its derivatives.

19.15.17 matplotlib.axes.Axes.get_agg_filter

Axes.get_agg_filter()
return filter function to be used for agg filter

19.15.18 matplotlib.axes.Axes.get_sketch_params

Axes.get_sketch_params()
Returns the sketch parameters for the artist.
Returns `sketch_params` : tuple or `None`

A 3-tuple with the following elements:

- `scale`: The amplitude of the wiggle perpendicular to the source line.
- `length`: The length of the wiggle along the line.
- `randomness`: The scale factor by which the length is shrunken or expanded.

May return `None` if no sketch parameters were set.

19.15.19 `matplotlib.axes.Axes.get_alpha`

`Axes.get_alpha()`  
Return the alpha value used for blending - not supported on all backends

19.15.20 `matplotlib.axes.Axes.get_animated`

`Axes.get_animated()`  
Return the artist’s animated state

19.15.21 `matplotlib.axes.Axes.get_clip_box`

`Axes.get_clip_box()`  
Return artist clipbox

19.15.22 `matplotlib.axes.Axes.get_clip_on`

`Axes.get_clip_on()`  
Return whether artist uses clipping

19.15.23 `matplotlib.axes.Axes.get_clip_path`

`Axes.get_clip_path()`  
Return artist clip path

19.15.24 `matplotlib.axes.Axes.get_gid`

`Axes.get_gid()`  
Returns the group id

19.15.25 `matplotlib.axes.Axes.get_label`

`Axes.get_label()`  
Get the label used for this artist in the legend.
19.15.26 matplotlib.axes.Axes.get_url

Axes.get_url()

Returns the url

19.15.27 matplotlib.axes.Axes.get_visible

Axes.get_visible()

Return the artist's visibility

19.15.28 matplotlib.axes.Axes.get_zorder

Axes.get_zorder()

Return the Artist's zorder.

19.15.29 matplotlib.axes.Axes.get_rasterized

Axes.get_rasterized()

return True if the artist is to be rasterized

19.15.30 matplotlib.axes.Axes.get_transform

Axes.get_transform()

Return the Transform instance used by this artist.

19.15.31 matplotlib.axes.Axes.get_snap

Axes.get_snap()

Returns the snap setting which may be:

- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

19.15.32 matplotlib.axes.Axes.get_path_effects

Axes.get_path_effects()
19.15.33 matplotlib.axes.Axes.axes

Axes.axes

The `Axes` instance the artist resides in, or `None`.

19.15.34 matplotlib.axes.Axes.get_axes

Axes.get_axes()

Return the `Axes` instance the artist resides in, or `None`.

This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

19.15.35 matplotlib.axes.Axes.set_axes

Axes.set_axes(axes)

Set the `Axes` instance in which the artist resides, if any.

This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

ACCEPTS: an `Axes` instance

19.15.36 matplotlib.axes.Axes.set_figure

Axes.set_figure(fig)

Set the class:`Axes` figure

accepts a class:`Figure` instance

19.15.37 matplotlib.axes.Axes.get_figure

Axes.get_figure()

Return the `Figure` instance the artist belongs to.

19.16 Artist Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Axes.is_figure_set</td>
<td>Returns True if the artist is assigned to a <code>Figure</code>.</td>
</tr>
<tr>
<td>Axes.remove</td>
<td>Remove the artist from the figure if possible.</td>
</tr>
<tr>
<td>Axes.is_transform_set</td>
<td>Returns <code>True</code> if <code>Artist</code> has a transform explicitly set.</td>
</tr>
</tbody>
</table>

19.16.1 matplotlib.axes.Axes.is_figure_set

Axes.is_figure_set()

Returns True if the artist is assigned to a `Figure`.
19.16.2 matplotlib.axes.Axes.remove

Axes.remove()

Remove the artist from the figure if possible. The effect will not be visible until the figure is redrawn, e.g., with matplotlib.axes.Axes.draw_idle(). Call matplotlib.axes.Axes.relim() to update the axes limits if desired.

Note: relim() will not see collections even if the collection was added to axes with autolim = True.

Note: there is no support for removing the artist’s legend entry.

19.16.3 matplotlib.axes.Axes.is_transform_set

Axes.is_transform_set()

Returns True if Artist has a transform explicitly set.

19.17 Projection

Methods used by Axis that must be overridden for non-rectilinear Axes.

<table>
<thead>
<tr>
<th>Axes.name</th>
<th>Get the transformation used for drawing x-axis labels, ticks and gridlines.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Axes.get_xaxis_transform</td>
<td>Get the transformation used for drawing x-axis labels, ticks and gridlines.</td>
</tr>
<tr>
<td>Axes.get_yaxis_transform</td>
<td>Get the transformation used for drawing y-axis labels, ticks and gridlines.</td>
</tr>
<tr>
<td>Axes.get_data_ratio</td>
<td>Returns the aspect ratio of the raw data.</td>
</tr>
<tr>
<td>Axes.get_data_ratio_log</td>
<td>Returns the aspect ratio of the raw data in log scale.</td>
</tr>
<tr>
<td>Axes.get_xaxis_text1_transform</td>
<td>Get the transformation used for drawing x-axis labels, which will add the given amount of padding (in points) between the axes and the label.</td>
</tr>
<tr>
<td>Axes.get_xaxis_text2_transform</td>
<td>Get the transformation used for drawing the secondary x-axis labels, which will add the given amount of padding (in points) between the axes and the label.</td>
</tr>
<tr>
<td>Axes.get_yaxis_text1_transform</td>
<td>Get the transformation used for drawing y-axis labels, which will add the given amount of padding (in points) between the axes and the label.</td>
</tr>
<tr>
<td>Axes.get_yaxis_text2_transform</td>
<td>Get the transformation used for drawing the secondary y-axis labels, which will add the given amount of padding (in points) between the axes and the label.</td>
</tr>
</tbody>
</table>

19.17.1 matplotlib.axes.Axes.name

Axes.name = ‘rectilinear’
19.17.2 matplotlib.axes.Axes.get_xaxis_transform

Axes.get_xaxis_transform(which='grid')

Get the transformation used for drawing x-axis labels, ticks and gridlines. The x-direction is in data coordinates and the y-direction is in axis coordinates.

**Note:** This transformation is primarily used by the *Axis* class, and is meant to be overridden by new kinds of projections that may need to place axis elements in different locations.

19.17.3 matplotlib.axes.Axes.get_yaxis_transform

Axes.get_yaxis_transform(which='grid')

Get the transformation used for drawing y-axis labels, ticks and gridlines. The x-direction is in axis coordinates and the y-direction is in data coordinates.

**Note:** This transformation is primarily used by the *Axis* class, and is meant to be overridden by new kinds of projections that may need to place axis elements in different locations.

19.17.4 matplotlib.axes.Axes.get_data_ratio

Axes.get_data_ratio()

Returns the aspect ratio of the raw data.

This method is intended to be overridden by new projection types.

19.17.5 matplotlib.axes.Axes.get_data_ratio_log

Axes.get_data_ratio_log()

Returns the aspect ratio of the raw data in log scale. Will be used when both axis scales are in log.

19.17.6 matplotlib.axes.Axes.get_xaxis_text1_transform

Axes.get_xaxis_text1_transform(pad_points)

Get the transformation used for drawing x-axis labels, which will add the given amount of padding (in points) between the axes and the label. The x-direction is in data coordinates and the y-direction is in axis coordinates. Returns a 3-tuple of the form:

```
(transform, valign, halign)
```

where *valign* and *halign* are requested alignments for the text.
**Note:** This transformation is primarily used by the *Axis* class, and is meant to be overridden by new kinds of projections that may need to place axis elements in different locations.

### 19.17.7 matplotlib.axes.Axes.get_xaxis_text2_transform

**Axes.get_xaxis_text2_transform** (*pad_points*)

Get the transformation used for drawing the secondary x-axis labels, which will add the given amount of padding (in points) between the axes and the label. The x-direction is in data coordinates and the y-direction is in axis coordinates. Returns a 3-tuple of the form:

```
(transform, valign, halign)
```

where `valign` and `halign` are requested alignments for the text.

**Note:** This transformation is primarily used by the *Axis* class, and is meant to be overridden by new kinds of projections that may need to place axis elements in different locations.

### 19.17.8 matplotlib.axes.Axes.get_yaxis_text1_transform

**Axes.get_yaxis_text1_transform** (*pad_points*)

Get the transformation used for drawing y-axis labels, which will add the given amount of padding (in points) between the axes and the label. The x-direction is in axis coordinates and the y-direction is in data coordinates. Returns a 3-tuple of the form:

```
(transform, valign, halign)
```

where `valign` and `halign` are requested alignments for the text.

**Note:** This transformation is primarily used by the *Axis* class, and is meant to be overridden by new kinds of projections that may need to place axis elements in different locations.

### 19.17.9 matplotlib.axes.Axes.get_yaxis_text2_transform

**Axes.get_yaxis_text2_transform** (*pad_points*)

Get the transformation used for drawing the secondary y-axis labels, which will add the given amount of padding (in points) between the axes and the label. The x-direction is in axis coordinates and the y-direction is in data coordinates. Returns a 3-tuple of the form:

```
(transform, valign, halign)
```

where `valign` and `halign` are requested alignments for the text.
**Note:** This transformation is primarily used by the *Axis* class, and is meant to be overridden by new kinds of projections that may need to place axis elements in different locations.

## 19.18 Other

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Axes.zorder</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>Axes.aname</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>Axes.get_default_bbox_extra_artists</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>Axes.get_transformed_clip_path_and_affine()</code></td>
<td>Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.</td>
<td></td>
</tr>
<tr>
<td><code>Axes.has_data</code></td>
<td>Return <code>True</code> if any artists have been added to axes.</td>
<td></td>
</tr>
<tr>
<td><code>Axes.hold</code></td>
<td>Deprecated since version 2.0.</td>
<td></td>
</tr>
<tr>
<td><code>Axes.ishold</code></td>
<td>Deprecated since version 2.0.</td>
<td></td>
</tr>
</tbody>
</table>

### 19.18.1 matplotlib.axes.Axes.zorder

`Axes.zorder = 0`

### 19.18.2 matplotlib.axes.Axes.aname

`Axes.aname = ‘Axes’`

### 19.18.3 matplotlib.axes.Axes.get_default_bbox_extra_artists

`Axes.get_default_bbox_extra_artists()`

### 19.18.4 matplotlib.axes.Axes.get_transformed_clip_path_and_affine

`Axes.get_transformed_clip_path_and_affine()`

Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.
19.18.5 matplotlib.axes.Axes.has_data

Axes.has_data()

Return True if any artists have been added to axes.

This should not be used to determine whether the dataLim need to be updated, and may not actually be useful for anything.

19.18.6 matplotlib.axes.Axes.hold

Axes.hold(b=None)

Deprecated since version 2.0: axes.hold is deprecated. See the API Changes document (http://matplotlib.org/api/api_changes.html) for more details.

Set the hold state

The hold mechanism is deprecated and will be removed in v3.0. The behavior will remain consistent with the long-time default value of True.

If hold is None (default), toggle the hold state. Else set the hold state to boolean value b.

Examples:

```python
# toggle hold
hold()

# turn hold on
hold(True)

# turn hold off
hold(False)
```

When hold is True, subsequent plot commands will be added to the current axes. When hold is False, the current axes and figure will be cleared on the next plot command

19.18.7 matplotlib.axes.Axes.ishold

Axes.ishold()

Deprecated since version 2.0: The ishold function was deprecated in version 2.0.

return the HOLD status of the axes

The hold mechanism is deprecated and will be removed in v3.0.
19.19 Inheritance
Table of Contents

- Inheritance
- Axis objects
  - Formatters and Locators
  - Axis Label
  - Ticks, tick labels and Offset text
  - Data and view intervals
  - Rendering helpers
  - Interactive
  - Units
  - Incremental navigation
  - YAxis Specific
  - XAxis Specific
  - Other
  - Discouraged
- Tick objects
- Common and inherited methods
  - XTick
  - YTick
  - YAxis
  - YAxis
  - Inherited from artist
    * Ticks
Classes for the ticks and x and y axis

20.1 Inheritance

```
    matplotlib.artist.Artist
    matplotlib.axis.Axis
    matplotlib.axis.Tick
    matplotlib.axis.XAxis
    matplotlib.axis.YAxis
    matplotlib.axis.XTick
    matplotlib.axis.YTick
    matplotlib.axis.Ticker
```

20.2 Axis objects

```python
class matplotlib.axis.Axis(axes, pickradius=15)
    Public attributes
    • axes.transData - transform data coords to display coords
    • axes.transAxes - transform axis coords to display coords
    • labelpad - number of points between the axis and its label
    Init the axis with the parent Axes instance
class matplotlib.axis.XAxis(axes, pickradius=15)
    Init the axis with the parent Axes instance
class matplotlib.axis.YAxis(axes, pickradius=15)
    Init the axis with the parent Axes instance
class matplotlib.axis.Ticker
```

```
Axis.clc  clear the current axis
Axis.get_scale
```
20.2.1 matplotlib.axis.Axis.cla

Axis.cla()
    clear the current axis

20.2.2 matplotlib.axis.Axis.get_scale

Axis.get_scale()

20.2.3 Formatters and Locators

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Axis.get_major_formatter</td>
<td>Get the formatter of the major ticker</td>
</tr>
<tr>
<td>Axis.get_major_locator</td>
<td>Get the locator of the major ticker</td>
</tr>
<tr>
<td>Axis.get_minor_formatter</td>
<td>Get the formatter of the minor ticker</td>
</tr>
<tr>
<td>Axis.get_minor_locator</td>
<td>Get the locator of the minor ticker</td>
</tr>
<tr>
<td>Axis.set_major_formatter</td>
<td>Set the formatter of the major ticker</td>
</tr>
<tr>
<td>Axis.set_major_locator</td>
<td>Set the locator of the major ticker</td>
</tr>
<tr>
<td>Axis.set_minor_formatter</td>
<td>Set the formatter of the minor ticker</td>
</tr>
<tr>
<td>Axis.set_minor_locator</td>
<td>Set the locator of the minor ticker</td>
</tr>
</tbody>
</table>

matplotlib.axis.Axis.get_major_formatter

Axis.get_major_formatter()
    Get the formatter of the major ticker

matplotlib.axis.Axis.get_major_locator

Axis.get_major_locator()
    Get the locator of the major ticker

matplotlib.axis.Axis.get_minor_formatter

Axis.get_minor_formatter()
    Get the formatter of the minor ticker

matplotlib.axis.Axis.get_minor_locator

Axis.get_minor_locator()
    Get the locator of the minor ticker
matplotlib.axis.Axis.set_major_formatter

**Axis.set_major_formatter**(formatter)
Set the formatter of the major ticker

ACCEPTS: A *Formatter* instance

matplotlib.axis.Axis.set_major_locator

**Axis.set_major_locator**(locator)
Set the locator of the major ticker

ACCEPTS: a *Locator* instance

matplotlib.axis.Axis.set_minor_formatter

**Axis.set_minor_formatter**(formatter)
Set the formatter of the minor ticker

ACCEPTS: A *Formatter* instance

matplotlib.axis.Axis.set_minor_locator

**Axis.set_minor_locator**(locator)
Set the locator of the minor ticker

ACCEPTS: a *Locator* instance

### 20.2.4 Axis Label

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Axis.set_label_coords</strong></td>
<td>Set the coordinates of the label.</td>
</tr>
<tr>
<td><strong>Axis.set_label_position</strong></td>
<td>Set the label position (top or bottom)</td>
</tr>
<tr>
<td><strong>Axis.set_label_text</strong></td>
<td>Sets the text value of the axis label</td>
</tr>
<tr>
<td><strong>Axis.get_label_position</strong></td>
<td>Return the label position (top or bottom)</td>
</tr>
<tr>
<td><strong>Axis.get_label_text</strong></td>
<td>Get the text of the label</td>
</tr>
</tbody>
</table>

matplotlib.axis.Axis.set_label_coords

**Axis.set_label_coords**(x, y, transform=None)
Set the coordinates of the label. By default, the x coordinate of the y label is determined by the tick label bounding boxes, but this can lead to poor alignment of multiple ylabels if there are multiple axes. Ditto for the y coordinate of the x label.

You can also specify the coordinate system of the label with the transform. If None, the default coordinate system will be the axes coordinate system (0,0) is (left,bottom), (0.5, 0.5) is middle, etc
matplotlib.axis.Axis.set_label_position

Axis.set_label_position(position)
Set the label position (top or bottom)
ACCEPES: [‘top' | ‘bottom’]

matplotlib.axis.Axis.set_label_text

Axis.set_label_text(label, fontdict=None, **kwargs)
Sets the text value of the axis label
ACCEPES: A string value for the label

matplotlib.axis.Axis.get_label_position

Axis.get_label_position()
Return the label position (top or bottom)

matplotlib.axis.Axis.get_label_text

Axis.get_label_text()
Get the text of the label

20.2.5 Ticks, tick labels and Offset text

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Axis.get_major_ticks</td>
<td>get the tick instances; grow as necessary</td>
</tr>
<tr>
<td>Axis.get_majorticklabels</td>
<td>Return a list of Text instances for the major ticklabels</td>
</tr>
<tr>
<td>Axis.get_majorticklines</td>
<td>Return the major tick lines as a list of Line2D instances</td>
</tr>
<tr>
<td>Axis.get_majorticklocs</td>
<td>Get the major tick locations in data coordinates as a numpy array</td>
</tr>
<tr>
<td>Axis.get_minor_ticks</td>
<td>get the minor tick instances; grow as necessary</td>
</tr>
<tr>
<td>Axis.get_minorticklabels</td>
<td>Return a list of Text instances for the minor ticklabels</td>
</tr>
<tr>
<td>Axis.get_minorticklines</td>
<td>Return the minor tick lines as a list of Line2D instances</td>
</tr>
<tr>
<td>Axis.get_minorticklocs</td>
<td>Get the minor tick locations in data coordinates as a numpy array</td>
</tr>
<tr>
<td>Axis.get_offset_text</td>
<td>Return the axis offsetText as a Text instance</td>
</tr>
<tr>
<td>Axis.get_tick_padding</td>
<td></td>
</tr>
<tr>
<td>Axis.get_ticklabels</td>
<td>Get the x tick labels as a list of Text instances.</td>
</tr>
<tr>
<td>Axis.get_ticklines</td>
<td>Return the tick lines as a list of Line2D instances</td>
</tr>
<tr>
<td>Axis.get_ticklocs</td>
<td>Get the tick locations in data coordinates as a numpy array</td>
</tr>
</tbody>
</table>

Continued on next page
Table 20.4 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Axis.get_gridlines</code></td>
<td>Return the grid lines as a list of Line2D instances</td>
</tr>
<tr>
<td><code>Axis.grid</code></td>
<td>Set the axis grid on or off; <code>b</code> is a boolean.</td>
</tr>
<tr>
<td><code>Axis.iter_ticks</code></td>
<td>Iterate through all of the major and minor ticks.</td>
</tr>
<tr>
<td><code>Axis.set_tick_params</code></td>
<td>Set appearance parameters for ticks and ticklabels.</td>
</tr>
<tr>
<td><code>Axis.axis_date</code></td>
<td>Sets up x-axis ticks and labels that treat the x data as dates.</td>
</tr>
</tbody>
</table>

**matplotlib.axis.Axis.get_major_ticks**

```python
Axis.get_major_ticks(numticks=None)
```

get the tick instances; grow as necessary

**matplotlib.axis.Axis.get_majorticklabels**

```python
Axis.get_majorticklabels()
```

Return a list of Text instances for the major ticklabels

**matplotlib.axis.Axis.get_majorticklines**

```python
Axis.get_majorticklines()
```

Return the major tick lines as a list of Line2D instances

**matplotlib.axis.Axis.get_majorticklocs**

```python
Axis.get_majorticklocs()
```

Get the major tick locations in data coordinates as a numpy array

**matplotlib.axis.Axis.get_minor_ticks**

```python
Axis.get_minor_ticks(numticks=None)
```

get the minor tick instances; grow as necessary

**matplotlib.axis.Axis.get_minorticklabels**

```python
Axis.get_minorticklabels()
```

Return a list of Text instances for the minor ticklabels

**matplotlib.axis.Axis.get_minorticklines**

```python
Axis.get_minorticklines()
```

Return the minor tick lines as a list of Line2D instances
**matplotlib.axis.Axis.get_minorticklocs**

Axis.get_minorticklocs()
Get the minor tick locations in data coordinates as a numpy array

**matplotlib.axis.Axis.get_offset_text**

Axis.get_offset_text()
Return the axis offsetText as a Text instance

**matplotlib.axis.Axis.get_tick_padding**

Axis.get_tick_padding()

**matplotlib.axis.Axis.get_ticklabels**

Axis.get_ticklabels(minor=False, which=None)
Get the x tick labels as a list of Text instances.

**Parameters**

- **minor** : bool
  If True return the minor ticklabels, else return the major ticklabels

- **which** : None, (‘minor’, ‘major’, ‘both’)
  Overrides minor.
  Selects which ticklabels to return

**Returns**

- **ret** : list
  List of Text instances.

**matplotlib.axis.Axis.get_ticklines**

Axis.get_ticklines(minor=False)
Return the tick lines as a list of Line2D instances

**matplotlib.axis.Axis.get_ticklocs**

Axis.get_ticklocs(minor=False)
Get the tick locations in data coordinates as a numpy array
matplotlib.axis.Axis.get_gridlines

Axis.get_gridlines()
Return the grid lines as a list of Line2D instance

matplotlib.axis.Axis.grid

Axis.grid(b=None, which='major', **kwargs)
Set the axis grid on or off; b is a boolean. Use which = ‘major’ | ‘minor’ | ‘both’ to set the grid for major or minor ticks.
If b is None and len(kwargs)==0, toggle the grid state. If kwargs are supplied, it is assumed you want the grid on and b will be set to True.
kwargs are used to set the line properties of the grids, e.g.,
xax.grid(color='r', linestyle='-', linewidth=2)

matplotlib.axis.Axis.iter_ticks

Axis.iter_ticks()
Iterate through all of the major and minor ticks.

matplotlib.axis.Axis.set_tick_params

Axis.set_tick_params(which='major', reset=False, **kw)
Set appearance parameters for ticks and ticklabels.

For documentation of keyword arguments, see matplotlib.axes.Axes.tick_params().

matplotlib.axis.Axis.axis_date

Axis.axis_date(tz=None)
Sets up x-axis ticks and labels that treat the x data as dates. tz is a tzinfo instance or a timezone string. This timezone is used to create date labels.

20.2.6 Data and view intervals

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Axis.get_data_interval</td>
<td>return the Interval instance for this axis data limits</td>
</tr>
<tr>
<td>Axis.get_view_interval</td>
<td>return the Interval instance for this axis view limits</td>
</tr>
<tr>
<td>Axis.set_data_interval</td>
<td>set the axis data limits</td>
</tr>
<tr>
<td>Axis.set_view_interval</td>
<td></td>
</tr>
</tbody>
</table>
matplotlib.axis.Axis.get_data_interval

Axis.get_data_interval()
return the Interval instance for this axis data limits

matplotlib.axis.Axis.get_view_interval

Axis.get_view_interval()
return the Interval instance for this axis view limits

matplotlib.axis.Axis.set_data_interval

Axis.set_data_interval()
set the axis data limits

matplotlib.axis.Axis.set_view_interval

Axis.set_view_interval(vmin, vmax, ignore=False)

20.2.7 Rendering helpers

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Axis.get_minpos</td>
<td>Return the estimated number of ticks that can fit on the axis.</td>
</tr>
<tr>
<td>Axis.get_tick_space</td>
<td>Return the estimated number of ticks that can fit on the axis.</td>
</tr>
<tr>
<td>Axis.get_ticklabel_extents</td>
<td>Get the extents of the tick labels on either side of the axes.</td>
</tr>
<tr>
<td>Axis.get_tightbbox</td>
<td>Return a bounding box that encloses the axis.</td>
</tr>
</tbody>
</table>

matplotlib.axis.Axis.get_minpos

Axis.get_minpos()

matplotlib.axis.Axis.get_tick_space

Axis.get_tick_space()
Return the estimated number of ticks that can fit on the axis.

matplotlib.axis.Axis.get_ticklabel_extents

Axis.get_ticklabel_extents(renderer)
Get the extents of the tick labels on either side of the axes.
**Matplotlib, Release 2.0.2**

`matlab.axis.Axes.get_tightbbox`

*Axes.get_tightbbox/renderer)*

Return a bounding box that encloses the axis. It only accounts tick labels, axis label, and offsetText.

**20.2.8 Interactive**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Axes.get_pickradius</code></td>
<td>Return the depth of the axis used by the picker</td>
</tr>
<tr>
<td><code>Axes.set_pickradius</code></td>
<td>Set the depth of the axis used by the picker</td>
</tr>
</tbody>
</table>

`matlab.axis.Axes.get_pickradius`

*Axes.get_pickradius()*(*)

Return the depth of the axis used by the picker

`matlab.axis.Axes.set_pickradius`

*Axes.set_pickradius(pickradius)*

Set the depth of the axis used by the picker

 ACCEPTS: a distance in points

**20.2.9 Units**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Axes.convert_units</code></td>
<td>set the units for axis</td>
</tr>
<tr>
<td><code>Axes.set_units</code></td>
<td>set the units for axis</td>
</tr>
<tr>
<td><code>Axes.get_units</code></td>
<td>return the units for axis</td>
</tr>
<tr>
<td><code>Axes.update_units</code></td>
<td>introspect data for units converter and update the</td>
</tr>
</tbody>
</table>

`matlab.axis.Axes.convert_units`

*Axes.convert_units(x)*

`matlab.axis.Axes.set_units`

*Axes.set_units(u)*

set the units for axis

 ACCEPTS: a units tag
Axis.get_units()  
return the units for axis

Axis.update_units(data)  
introspect data for units converter and update the axis.converter instance if necessary. Return True if data is registered for unit conversion.

### 20.2.10 Incremental navigation

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Axis.pan</td>
<td>Pan numsteps (can be positive or negative)</td>
</tr>
<tr>
<td>Axis.zoom</td>
<td>Zoom in/out on axis; if direction is &gt;0 zoom in, else zoom out</td>
</tr>
</tbody>
</table>

### 20.2.11 YAxis Specific

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>YAxis.axis_name</td>
<td>'y'</td>
</tr>
<tr>
<td>YAxis.get_text_widths</td>
<td>Return the ticks position (left, right, both or unknown)</td>
</tr>
<tr>
<td>YAxis.get_ticks_position</td>
<td>Return the ticks position (left, right, both or unknown)</td>
</tr>
<tr>
<td>YAxis.set_offset_position</td>
<td>Set the ticks position (left, right, both, default or none)</td>
</tr>
<tr>
<td>YAxis.set_ticks_position</td>
<td>Set the ticks position (left, right, both, default or none)</td>
</tr>
<tr>
<td>YAxis.tick_left</td>
<td>use ticks only on left</td>
</tr>
<tr>
<td>YAxis.tick_right</td>
<td>use ticks only on right</td>
</tr>
</tbody>
</table>

Axis.objects 803
matplotlib.axis.YAxis.get_text_widths

YAxis.get_text_widths(renderer)

matplotlib.axis.YAxis.get_ticks_position

YAxis.get_ticks_position()

Return the ticks position (left, right, both or unknown)

matplotlib.axis.YAxis.set_offset_position

YAxis.set_offset_position(position)

matplotlib.axis.YAxis.set_ticks_position

YAxis.set_ticks_position(position)

Set the ticks position (left, right, both, default or none) ‘both’ sets the ticks to appear on both positions, but does not change the tick labels. ‘default’ resets the tick positions to the default: ticks on both positions, labels at left. ‘none’ can be used if you don’t want any ticks. ‘none’ and ‘both’ affect only the ticks, not the labels.

ACCEPTS: [‘left’ | ‘right’ | ‘both’ | ‘default’ | ‘none’]

matplotlib.axis.YAxis.tick_left

YAxis.tick_left()

use ticks only on left

matplotlib.axis.YAxis.tick_right

YAxis.tick_right()

use ticks only on right

20.2.12 XAxis Specific

<table>
<thead>
<tr>
<th>XAxis.axis_name</th>
<th>Returns the amount of space one should reserve for text above and below the axes.</th>
</tr>
</thead>
<tbody>
<tr>
<td>XAxis.get_text_heights</td>
<td></td>
</tr>
<tr>
<td>XAxis.get_ticks_position</td>
<td>Return the ticks position (top, bottom, default or unknown)</td>
</tr>
</tbody>
</table>

Continued on next page
Table 20.11 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>XAxis.set_ticks_position</code></td>
<td>Set the ticks position (top, bottom, both, default or none) both sets the</td>
</tr>
<tr>
<td></td>
<td>ticks to appear on both positions, but does not change the tick labels.</td>
</tr>
<tr>
<td><code>XAxis.tick_bottom</code></td>
<td>use ticks only on bottom</td>
</tr>
<tr>
<td><code>XAxis.tick_top</code></td>
<td>use ticks only on top</td>
</tr>
</tbody>
</table>

```python
matplotlib.axis.XAxis.axis_name
```

`XAxis.axis_name = 'x'`

```python
matplotlib.axis.XAxis.get_text_heights
```

`XAxis.get_text_heights(renderer)`

Returns the amount of space one should reserve for text above and below the axes. Returns a tuple (above, below)

```python
matplotlib.axis.XAxis.get_ticks_position
```

`XAxis.get_ticks_position()`

Return the ticks position (top, bottom, default or unknown)

```python
matplotlib.axis.XAxis.set_ticks_position
```

`XAxis.set_ticks_position(position)`

Set the ticks position (top, bottom, both, default or none) both sets the ticks to appear on both positions, but does not change the tick labels. ‘default’ resets the tick positions to the default: ticks on both positions, labels at bottom. ‘none’ can be used if you don’t want any ticks. ‘none’ and ‘both’ affect only the ticks, not the labels.

ACCEPTS: [‘top’ | ‘bottom’ | ‘both’ | ‘default’ | ‘none’ ]

```python
matplotlib.axis.XAxis.tick_bottom
```

`XAxis.tick_bottom()`

use ticks only on bottom

```python
matplotlib.axis.XAxis.tick_top
```

`XAxis.tick_top()`

use ticks only on top
20.2.13 Other

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Axis.OFFSETTEXTPAD</code></td>
<td></td>
</tr>
<tr>
<td><code>Axis.limit_range_for_scale</code></td>
<td>set the default limits for the axis data and view interval if they are not mutated</td>
</tr>
<tr>
<td><code>Axis.reset_ticks</code></td>
<td></td>
</tr>
<tr>
<td><code>Axis.set_default_intervals</code></td>
<td>set the default limits for the axis data and view interval if they are not mutated</td>
</tr>
<tr>
<td><code>Axis.get_smart_bounds</code></td>
<td>get whether the axis has smart bounds</td>
</tr>
<tr>
<td><code>Axis.set_smart_bounds(value)</code></td>
<td>set the axis to have smart bounds</td>
</tr>
</tbody>
</table>

**matplotlib.axis.Axis.OFFSETTEXTPAD**

`Axis.OFFSETTEXTPAD = 3`

**matplotlib.axis.Axis.limit_range_for_scale**

`Axis.limit_range_for_scale(vmin, vmax)`

**matplotlib.axis.Axis.reset_ticks**

`Axis.reset_ticks()`

**matplotlib.axis.Axis.set_default_intervals**

`Axis.set_default_intervals()`

**matplotlib.axis.Axis.get_smart_bounds**

`Axis.get_smart_bounds()`

**matplotlib.axis.Axis.set_smart_bounds(value)**

`Axis.set_smart_bounds(value)`
20.2.14 Discouraged

These methods implicitly use `FixedLocator` and `FixedFormatter`. They can be convenient, but if not used together may de-couple your tick labels from your data.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Axis.set_ticklabels</code></td>
<td>Set the text values of the tick labels.</td>
</tr>
<tr>
<td><code>Axis.set_ticks</code></td>
<td>Set the locations of the tick marks from sequence ticks.</td>
</tr>
</tbody>
</table>

**matplotlib.axis.Axis.set_ticklabels**

`Axis.set_ticklabels(ticklabels, *args, **kwargs)`

Set the text values of the tick labels. Return a list of Text instances. Use `kwarg minor=True` to select minor ticks. All other kwargs are used to update the text object properties. As for `get_ticklabels`, `label1` (left or bottom) is affected for a given tick only if its `label1On` attribute is True, and similarly for `label2`. The list of returned label text objects consists of all such `label1` objects followed by all such `label2` objects.

The input `ticklabels` is assumed to match the set of tick locations, regardless of the state of `label1On` and `label2On`.

ACCEPTS: sequence of strings or Text objects

**matplotlib.axis.Axis.set_ticks**

`Axis.set_ticks(ticks, minor=False)`

Set the locations of the tick marks from sequence ticks

ACCEPTS: sequence of floats

20.3 Tick objects

**class matplotlib.axis.Tick**

Abstract base class for the axis ticks, grid lines and labels

1 refers to the bottom of the plot for xticks and the left for yticks

2 refers to the top of the plot for xticks and the right for yticks

Publicly accessible attributes:

- `tick1line` a Line2D instance
- `tick2line` a Line2D instance
- `gridline` a Line2D instance
- `label1` a Text instance
label2 a Text instance
gridOn a boolean which determines whether to draw the tickline
tick1On a boolean which determines whether to draw the 1st tickline
tick2On a boolean which determines whether to draw the 2nd tickline
label1On a boolean which determines whether to draw tick label
label2On a boolean which determines whether to draw tick label
bbox is the Bound2D bounding box in display coords of the Axes
loc is the tick location in data coords
size is the tick size in points

class matplotlib.axis.XTick(axes, loc, label, size=None, width=None, color=None, tickdir=None, pad=None, labelsize=None, labelcolor=None, zorder=None, gridOn=None, tick1On=True, tick2On=True, label1On=True, label2On=False, major=True)
Contains all the Artists needed to make an x tick - the tick line, the label text and the grid line
bbox is the Bound2D bounding box in display coords of the Axes
loc is the tick location in data coords
size is the tick size in points

class matplotlib.axis.YTick(axes, loc, label, size=None, width=None, color=None, tickdir=None, pad=None, labelsize=None, labelcolor=None, zorder=None, gridOn=None, tick1On=True, tick2On=True, label1On=True, label2On=False, major=True)
Contains all the Artists needed to make a Y tick - the tick line, the label text and the grid line
bbox is the Bound2D bounding box in display coords of the Axes
loc is the tick location in data coords
size is the tick size in points

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tick.apply_tickdir</td>
<td>Calculate self._pad and self._tickmarkers</td>
</tr>
<tr>
<td>Tick.get_loc</td>
<td>Return the tick location (data coords) as a scalar</td>
</tr>
<tr>
<td>Tick.get_pad</td>
<td>Get the value of the tick label pad in points</td>
</tr>
<tr>
<td>Tick.get_pad_pixels</td>
<td>Get the length of the tick outside of the axes.</td>
</tr>
<tr>
<td>Tick.get_tick_padding</td>
<td></td>
</tr>
<tr>
<td>Tick.get_view_interval</td>
<td>return the view Interval instance for the axis this tick is ticking</td>
</tr>
<tr>
<td>Tick.set_label1</td>
<td>Set the text of ticklabel</td>
</tr>
<tr>
<td>Tick.set_label2</td>
<td>Set the text of ticklabel2</td>
</tr>
<tr>
<td>Tick.set_pad</td>
<td>Set the tick label pad in points</td>
</tr>
<tr>
<td>Tick.update_position</td>
<td>Set the location of tick in data coords with scalar loc</td>
</tr>
</tbody>
</table>

20.3.1 matplotlib.axis.Tick.apply_tickdir

Tick.apply_tickdir(tickdir)
    Calculate self._pad and self._tickmarkers
20.3.2 matplotlib.axis.Tick.get_loc

Tick.get_loc()
    Return the tick location (data coords) as a scalar

20.3.3 matplotlib.axis.Tick.get_pad

Tick.get_pad()
    Get the value of the tick label pad in points

20.3.4 matplotlib.axis.Tick.get_pad_pixels

Tick.get_pad_pixels()

20.3.5 matplotlib.axis.Tick.get_tick_padding

Tick.get_tick_padding()
    Get the length of the tick outside of the axes.

20.3.6 matplotlib.axis.Tick.get_tickdir

Tick.get_tickdir()

20.3.7 matplotlib.axis.Tick.get_view_interval

Tick.get_view_interval()
    return the view Interval instance for the axis this tick is ticking

20.3.8 matplotlib.axis.Tick.set_label1

Tick.set_label1(s)
    Set the text of ticklabel
    ACCEPTS: str

20.3.9 matplotlib.axis.Tick.set_label2

Tick.set_label2(s)
    Set the text of ticklabel2
    ACCEPTS: str
20.3.10 matplotlib.axis.Tick.set_pad

```
Tick.set_pad(val)
  Set the tick label pad in points
  ACCEPTS: float
```

20.3.11 matplotlib.axis.Tick.update_position

```
Tick.update_position(loc)
  Set the location of tick in data coords with scalar loc
```

20.4 Common and inherited methods

20.4.1 XTick

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xTick.apply_tickdir</td>
<td>Return the tick location (data coords) as a scalar</td>
</tr>
<tr>
<td>xTick.get_loc</td>
<td>return the Interval instance for this axis view limits</td>
</tr>
<tr>
<td>xTick.get_pad</td>
<td>Get the value of the tick label pad in points</td>
</tr>
<tr>
<td>xTick.get_pad_pixels</td>
<td>Get the length of the tick outside of the axes.</td>
</tr>
<tr>
<td>xTick.get_tick_padding</td>
<td>Set the tick label pad in points</td>
</tr>
<tr>
<td>xTick.get_tickdir</td>
<td>Set the location of tick in data coords with scalar loc</td>
</tr>
</tbody>
</table>

```matplotlib.axis.XTick.apply_tickdir``

```
XTick.apply_tickdir(tickdir)
```

```matplotlib.axis.XTick.get_loc``

```
XTick.get_loc()
  Return the tick location (data coords) as a scalar
```

```matplotlib.axis.XTick.get_pad``

```
XTick.get_pad()
  Get the value of the tick label pad in points
```
matplotlib.axis.XTick.get_pad_pixels

XTick.get_pad_pixels()

matplotlib.axis.XTick.get_tick_padding

XTick.get_tick_padding()
   Get the length of the tick outside of the axes.

matplotlib.axis.XTick.get_tickdir

XTick.get_tickdir()

matplotlib.axis.XTick.get_view_interval

XTick.get_view_interval()
   return the Interval instance for this axis view limits

matplotlib.axis.XTick.set_label1

XTick.set_label1(s)
   Set the text of ticklabel
   ACCEPTS: str

matplotlib.axis.XTick.set_label2

XTick.set_label2(s)
   Set the text of ticklabel2
   ACCEPTS: str

matplotlib.axis.XTick.set_pad

XTick.set_pad(val)
   Set the tick label pad in points
   ACCEPTS: float

matplotlib.axis.XTick.update_position

XTick.update_position(loc)
   Set the location of tick in data coords with scalar loc
## 20.4.2 YTick

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>YTick.apply_tickdir</code></td>
<td>Return the tick location (data coords) as a scalar</td>
</tr>
<tr>
<td><code>YTick.get_loc</code></td>
<td>Return the tick location (data coords) as a scalar</td>
</tr>
<tr>
<td><code>YTick.get_pad</code></td>
<td>Get the value of the tick label pad in points</td>
</tr>
<tr>
<td><code>YTick.get_pad_pixels</code></td>
<td>Get the length of the tick outside of the axes.</td>
</tr>
<tr>
<td><code>YTick.get_tickdir</code></td>
<td>Get the tick label pad in points</td>
</tr>
<tr>
<td><code>YTick.get_view_interval</code></td>
<td>return the Interval instance for this axis view limits</td>
</tr>
<tr>
<td><code>YTick.set_label1</code></td>
<td>Set the text of ticklabel</td>
</tr>
<tr>
<td><code>YTick.set_label2</code></td>
<td>Set the text of ticklabel2</td>
</tr>
<tr>
<td><code>YTick.set_pad</code></td>
<td>Set the location of tick in data coords with scalar loc</td>
</tr>
</tbody>
</table>

### matplotlib.axis.YTick.apply_tickdir

`YTick.apply_tickdir(tickdir)`

### matplotlib.axis.YTick.get_loc

`YTick.get_loc()`

Return the tick location (data coords) as a scalar

### matplotlib.axis.YTick.get_pad

`YTick.get_pad()`

Get the value of the tick label pad in points

### matplotlib.axis.YTick.get_pad_pixels

`YTick.get_pad_pixels()`

### matplotlib.axis.YTick.get_tickdir

`YTick.get_tickdir()`

Get the tick label pad in points

Get the length of the tick outside of the axes.
matplotlib.axis.YTick.get_view_interval

*YTick.get_view_interval()*

return the Interval instance for this axis view limits

matplotlib.axis.YTick.set_label1

*YTick.set_label1(s)*

Set the text of ticklabel

ACCEPES: str

matplotlib.axis.YTick.set_label2

*YTick.set_label2(s)*

Set the text of ticklabel2

ACCEPES: str

matplotlib.axis.YTick.set_pad

*YTick.set_pad(val)*

Set the tick label pad in points

ACCEPES: float

matplotlib.axis.YTick.update_position

*YTick.update_position(loc)*

Set the location of tick in data coords with scalar loc

20.4.3 YAxis

<table>
<thead>
<tr>
<th>YAxis.OFFSETTEXTPAD</th>
<th>Sets up x-axis ticks and labels that treat the x data as dates.</th>
</tr>
</thead>
<tbody>
<tr>
<td>YAxis.axis_date</td>
<td>clear the current axis</td>
</tr>
<tr>
<td></td>
<td>return the Interval instance for this axis data limits</td>
</tr>
<tr>
<td>YAxis.convert_units</td>
<td>Return the grid lines as a list of Line2D instance</td>
</tr>
<tr>
<td>YAxis.get_data_interval</td>
<td>Return the label position (top or bottom)</td>
</tr>
<tr>
<td>YAxis.get_gridlines</td>
<td>Get the text of the label</td>
</tr>
<tr>
<td>YAxis.get_label_position</td>
<td>Get the formatter of the major ticker</td>
</tr>
<tr>
<td>YAxis.get_major_formatter</td>
<td>Get the locator of the major ticker</td>
</tr>
</tbody>
</table>

Continued on next page
Table 20.17 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>YAxis.get_major_ticks</td>
<td>get the tick instances; grow as necessary</td>
</tr>
<tr>
<td>YAxis.get_majorTickLabels</td>
<td>Return a list of Text instances for the major ticklabels</td>
</tr>
<tr>
<td>YAxis.get_majorTickLines</td>
<td>Return the major tick lines as a list of Line2D instances</td>
</tr>
<tr>
<td>YAxis.get_majorTickLocs</td>
<td>Get the major tick locations in data coordinates as a numpy array</td>
</tr>
<tr>
<td>YAxis.get_minor_formatter</td>
<td>Get the formatter of the minor ticker</td>
</tr>
<tr>
<td>YAxis.get_minor_locator</td>
<td>Get the locator of the minor ticker</td>
</tr>
<tr>
<td>YAxis.get_minor_ticks</td>
<td>get the minor tick instances; grow as necessary</td>
</tr>
<tr>
<td>YAxis.get_minorTickLabels</td>
<td>Return a list of Text instances for the minor ticklabels</td>
</tr>
<tr>
<td>YAxis.get_minorsTickLines</td>
<td>Return the minor tick lines as a list of Line2D instances</td>
</tr>
<tr>
<td>YAxis.get_minorTickLocs</td>
<td>Get the minor tick locations in data coordinates as a numpy array</td>
</tr>
<tr>
<td>YAxis.get_offset_text</td>
<td>Return the axis offsetText as a Text instance</td>
</tr>
<tr>
<td>YAxis.get_pickradius</td>
<td>Return the depth of the axis used by the picker</td>
</tr>
<tr>
<td>YAxis.get_scale</td>
<td>get whether the axis has smart bounds</td>
</tr>
<tr>
<td>YAxis.get_tick_padding</td>
<td>Get the extents of the tick labels on either side of the axes.</td>
</tr>
<tr>
<td>YAxis.get_tick_labels</td>
<td>Get the x tick labels as a list of Text instances</td>
</tr>
<tr>
<td>YAxis.get_ticklines</td>
<td>Return the tick lines as a list of Line2D instances</td>
</tr>
<tr>
<td>YAxis.get_ticklocs</td>
<td>Get the tick locations in data coordinates as a numpy array</td>
</tr>
<tr>
<td>YAxis.get_tightbbox</td>
<td>Return a bounding box that encloses the axis</td>
</tr>
<tr>
<td>YAxis.get_units</td>
<td>return the units for axis</td>
</tr>
<tr>
<td>YAxis.get_view_interval</td>
<td>return the Interval instance for this axis view limits</td>
</tr>
<tr>
<td>YAxis.grid</td>
<td>Set the axis grid on or off; b is a boolean.</td>
</tr>
<tr>
<td>YAxis.iter_ticks</td>
<td>Iterate through all of the major and minor ticks.</td>
</tr>
<tr>
<td>YAxis.limit_range_for_scale</td>
<td>一定 the axis data limits</td>
</tr>
<tr>
<td>YAxis.set_data_interval</td>
<td>set the default limits for the axis interval if they are not mutated</td>
</tr>
<tr>
<td>YAxis.set_default_intervals</td>
<td>set the axis data limits</td>
</tr>
<tr>
<td>YAxis.set_label_coords</td>
<td>Set the coordinates of the label</td>
</tr>
<tr>
<td>YAxis.set_label_position</td>
<td>Set the label position (left or right)</td>
</tr>
<tr>
<td>YAxis.set_label_text</td>
<td>Sets the text value of the axis label</td>
</tr>
<tr>
<td>YAxis.set_major_formatter</td>
<td>Set the formatter of the major ticker</td>
</tr>
<tr>
<td>YAxis.set_major_locator</td>
<td>Set the locator of the major ticker</td>
</tr>
<tr>
<td>YAxis.set_minor_formatter</td>
<td>Set the formatter of the minor ticker</td>
</tr>
</tbody>
</table>

Continued on next page
Table 20.17 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>YAxis.set_minor_locator</td>
<td>Set the locator of the minor ticker</td>
</tr>
<tr>
<td>YAxis.set_pickradius</td>
<td>Set the depth of the axis used by the picker</td>
</tr>
<tr>
<td>YAxis.set_smart_bounds</td>
<td>set the axis to have smart bounds</td>
</tr>
<tr>
<td>YAxis.set_tick_params</td>
<td>Set appearance parameters for ticks and ticklabels.</td>
</tr>
<tr>
<td>YAxis.set_ticklabels</td>
<td>Set the text values of the tick labels.</td>
</tr>
<tr>
<td>YAxis.set_ticks</td>
<td>Set the locations of the tick marks from sequence ticks</td>
</tr>
<tr>
<td>YAxis.set_units</td>
<td>set the units for axis</td>
</tr>
<tr>
<td>YAxis.set_view_interval</td>
<td>If ignore is False, the order of vmin, vmax does not matter; the original axis orientation will be preserved.</td>
</tr>
<tr>
<td>YAxis.update_units</td>
<td>introspect data for units converter and update the</td>
</tr>
<tr>
<td>YAxis.zoom</td>
<td>Zoom in/out on axis; if direction is &gt;0 zoom in, else zoom out</td>
</tr>
</tbody>
</table>

**matplotlib.axis.YAxis.OFFSETTEXTPAD**

YAxis.OFFSETTEXTPAD = 3

**matplotlib.axis.YAxis.axis_date**

YAxis.axis_date(tz=None)

Sets up x-axis ticks and labels that treat the x data as dates. tz is a tzinfo instance or a timezone string. This timezone is used to create date labels.

**matplotlib.axis.YAxis.cla**

YAxis.cla()

clear the current axis

**matplotlib.axis.YAxis.convert_units**

YAxis.convert_units(x)

**matplotlib.axis.YAxis.get_data_interval**

YAxis.get_data_interval()

return the Interval instance for this axis data limits

**matplotlib.axis.YAxis.get_gridlines**

YAxis.get_gridlines()

Return the grid lines as a list of Line2D instance
**matplotlib.axis.YAxis.get_label_position**

YAxis.get_label_position()
Return the label position (top or bottom)

**matplotlib.axis.YAxis.get_label_text**

YAxis.get_label_text()
Get the text of the label

**matplotlib.axis.YAxis.get_major_formatter**

YAxis.get_major_formatter()
Get the formatter of the major ticker

**matplotlib.axis.YAxis.get_major_locator**

YAxis.get_major_locator()
Get the locator of the major ticker

**matplotlib.axis.YAxis.get_major_ticks**

YAxis.get_major_ticks(numticks=None)
get the tick instances; grow as necessary

**matplotlib.axis.YAxis.get_majorticklabels**

YAxis.get_majorticklabels()
Return a list of Text instances for the major ticklabels

**matplotlib.axis.YAxis.get_majorticklines**

YAxis.get_majorticklines()
Return the major tick lines as a list of Line2D instances

**matplotlib.axis.YAxis.get_majorticklocs**

YAxis.get_majorticklocs()
Get the major tick locations in data coordinates as a numpy array
matplotlib.axis.YAxis.get_minor_formatter

YAxis.get_minor_formatter()
Get the formatter of the minor ticker

matplotlib.axis.YAxis.get_minor_locator

YAxis.get_minor_locator()
Get the locator of the minor ticker

matplotlib.axis.YAxis.get_minor_ticks

YAxis.get_minor_ticks(numticks=None)
get the minor tick instances; grow as necessary

matplotlib.axis.YAxis.get_minorticklabels

YAxis.get_minorticklabels()
Return a list of Text instances for the minor ticklabels

matplotlib.axis.YAxis.get_minorticklines

YAxis.get_minorticklines()
Return the minor tick lines as a list of Line2D instances

matplotlib.axis.YAxis.get_minorticklocs

YAxis.get_minorticklocs()
Get the minor tick locations in data coordinates as a numpy array

matplotlib.axis.YAxis.get_minpos

YAxis.get_minpos()

matplotlib.axis.YAxis.get_offset_text

YAxis.get_offset_text()
Return the axis offsetText as a Text instance
Matplotlib, Release 2.0.2

**matplotlib.axis.YAxis.get_pickradius**

YAxis.get_pickradius()

Return the depth of the axis used by the picker

**matplotlib.axis.YAxis.get_scale**

YAxis.get_scale()

**matplotlib.axis.YAxis.get_smart_bounds**

YAxis.get_smart_bounds()

get whether the axis has smart bounds

**matplotlib.axis.YAxis.get_tick_padding**

YAxis.get_tick_padding()

**matplotlib.axis.YAxis.get_tick_space**

YAxis.get_tick_space()

**matplotlib.axis.YAxis.get_ticklabel_extents**

YAxis.get_ticklabel_extents(renderer)

Get the extents of the tick labels on either side of the axes.

**matplotlib.axis.YAxis.get_ticklabels**

YAxis.get_ticklabels(minor=False, which=None)

Get the x tick labels as a list of Text instances.

**Parameters**

- **minor**: bool
  
  If True return the minor ticklabels, else return the major ticklabels

- **which**: None, ‘minor’, ‘major’, ‘both’
  
  Overrides minor.

  Selects which ticklabels to return

- **Returns**

  - **ret**: list
    
    List of Text instances.
**matplotlib.axis.YAxis.get_ticklines**

YAxis.get_ticklines(minor=False)

Return the tick lines as a list of Line2D instances.

**matplotlib.axis.YAxis.get_ticklocs**

YAxis.get_ticklocs(minor=False)

Get the tick locations in data coordinates as a numpy array.

**matplotlib.axis.YAxis.get_tightbbox**

YAxis.get_tightbbox(renderer)

Return a bounding box that encloses the axis. It only accounts tick labels, axis label, and offsetText.

**matplotlib.axis.YAxis.get_units**

YAxis.get_units()

return the units for axis.

**matplotlib.axis.YAxis.get_view_interval**

YAxis.get_view_interval()

return the Interval instance for this axis view limits.

**matplotlib.axis.YAxis.grid**

YAxis.grid(b=None, which='major', **kwargs)

Set the axis grid on or off; b is a boolean. Use which = ‘major’ | ‘minor’ | ‘both’ to set the grid for major or minor ticks.

If b is None and len(kwargs)==0, toggle the grid state. If kwargs are supplied, it is assumed you want the grid on and b will be set to True.

kwargs are used to set the line properties of the grids, e.g.,

```
xax.grid(color='r', linestyle='-', linewidth=2)
```

**matplotlib.axis.YAxis.iter_ticks**

YAxis.iter_ticks()

Iterate through all of the major and minor ticks.
**matplotlib.axis.YAxis.limit_range_for_scale**

`YAxis.limit_range_for_scale(vmin, vmax)`

**matplotlib.axis.YAxis.pan**

`YAxis.pan(numsteps)`

Pan `numsteps` (can be positive or negative)

**matplotlib.axis.YAxis.reset_ticks**

`YAxis.reset_ticks()`

**matplotlib.axis.YAxis.set_data_interval**

`YAxis.set_data_interval(vmin, vmax, ignore=False)`

set the axis data limits

**matplotlib.axis.YAxis.set_default_intervals**

`YAxis.set_default_intervals()`

set the default limits for the axis interval if they are not mutated

**matplotlib.axis.YAxis.set_label_coords**

`YAxis.set_label_coords(x, y, transform=None)`

Set the coordinates of the label. By default, the x coordinate of the y label is determined by the tick label bounding boxes, but this can lead to poor alignment of multiple ylabels if there are multiple axes. Ditto for the y coordinate of the x label.

You can also specify the coordinate system of the label with the transform. If None, the default coordinate system will be the axes coordinate system (0,0) is (left,bottom), (0.5, 0.5) is middle, etc

**matplotlib.axis.YAxis.set_label_position**

`YAxis.set_label_position(position)`

Set the label position (left or right)

ACCEPTS: [ ‘left’ | ‘right’ ]
**matplotlib.axis.YAxis.set_label_text**

`YAxis.set_label_text(label, fontdict=None, **kwargs)`

Sets the text value of the axis label

ACCEPTS: A string value for the label

**matplotlib.axis.YAxis.set_major_formatter**

`YAxis.set_major_formatter(formatter)`

Set the formatter of the major ticker

ACCEPTS: A `Formatter` instance

**matplotlib.axis.YAxis.set_major_locator**

`YAxis.set_major_locator(locator)`

Set the locator of the major ticker

ACCEPTS: a `Locator` instance

**matplotlib.axis.YAxis.set_minor_formatter**

`YAxis.set_minor_formatter(formatter)`

Set the formatter of the minor ticker

ACCEPTS: A `Formatter` instance

**matplotlib.axis.YAxis.set_minor_locator**

`YAxis.set_minor_locator(locator)`

Set the locator of the minor ticker

ACCEPTS: a `Locator` instance

**matplotlib.axis.YAxis.set_pickradius**

`YAxis.set_pickradius(pickradius)`

Set the depth of the axis used by the picker

ACCEPTS: a distance in points

**matplotlib.axis.YAxis.set_smart_bounds**

`YAxis.set_smart_bounds(value)`

set the axis to have smart bounds

20.4. Common and inherited methods
**matplotlib.axis.YAxis.set_tick_params**

`YAxis.set_tick_params(which='major', reset=False, **kw)`

Set appearance parameters for ticks and ticklabels.

For documentation of keyword arguments, see `matplotlib.axes.Axes.tick_params()`.

**matplotlib.axis.YAxis.set_ticklabels**

`YAxis.set_ticklabels(ticklabels, *args, **kwargs)`

Set the text values of the tick labels. Return a list of Text instances. Use `kwarg minor=True` to select minor ticks. All other kwargs are used to update the text object properties. As for `get_ticklabels`, `label1` (left or bottom) is affected for a given tick only if its `label1On` attribute is True, and similarly for `label2`. The list of returned label text objects consists of all such `label1` objects followed by all such `label2` objects.

The input `ticklabels` is assumed to match the set of tick locations, regardless of the state of `label1On` and `label2On`.

ACCEPTS: sequence of strings or Text objects

**matplotlib.axis.YAxis.set_ticks**

`YAxis.set_ticks(ticks, minor=False)`

Set the locations of the tick marks from sequence ticks

ACCEPTS: sequence of floats

**matplotlib.axis.YAxis.set_units**

`YAxis.set_units(u)`

Set the units for axis

ACCEPTS: a units tag

**matplotlib.axis.YAxis.set_view_interval**

`YAxis.set_view_interval(vmin, vmax, ignore=False)`

If `ignore` is `False`, the order of `vmin`, `vmax` does not matter; the original axis orientation will be preserved. In addition, the view limits can be expanded, but will not be reduced. This method is for mpl internal use; for normal use, see `set_ylim()`.

**matplotlib.axis.YAxis.update_units**

`YAxis.update_units(data)`

Introspect `data` for units converter and update the axis.converter instance if necessary. Return `True` if `data` is registered for unit conversion.
**matplotlib**: 20.4.4 YAxis

YAxis.zoom(direction)

Zoom in/out on axis; if direction is >0 zoom in, else zoom out

---

### XAxis

- **XAxis.OFFSETTEXTPAD**

- **XAxis.axis_date**
  Sets up x-axis ticks and labels that treat the x data as dates.

- **XAxis.cla**
  clear the current axis

- **XAxis.convert_units**

- **XAxis.get_data_interval**
  return the Interval instance for this axis data limits

- **XAxis.get_gridlines**
  Return the grid lines as a list of Line2D instance

- **XAxis.get_label_position**
  Return the label position (top or bottom)

- **XAxis.get_label_text**
  Get the text of the label

- **XAxis.get_major_formatter**
  Get the formatter of the major ticker

- **XAxis.get_major_locator**
  Get the locator of the major ticker

- **XAxis.get_major_ticks**
  get the tick instances; grow as necessary

- **XAxis.get_majorticklabels**
  Return a list of Text instances for the major ticklabels

- **XAxis.get_majorticklines**
  Return the major tick lines as a list of Line2D instances

- **XAxis.get_majorticklocs**
  Get the major tick locations in data coordinates as a numpy array

- **XAxis.get_minor_formatter**
  Get the formatter of the minor ticker

- **XAxis.get_minor_locator**
  Get the locator of the minor ticker

- **XAxis.get_minor_ticks**
  get the minor tick instances; grow as necessary

- **XAxis.get_minorticklabels**
  Return a list of Text instances for the minor ticklabels

- **XAxis.get_minorticklines**
  Return the minor tick lines as a list of Line2D instances

- **XAxis.get_minorticklocs**
  Get the minor tick locations in data coordinates as a numpy array

- **XAxis.get_minpos**

- **XAxis.get_offset_text**
  Return the axis offsetText as a Text instance

- **XAxis.get_pickradius**
  Return the depth of the axis used by the picker

- **XAxis.get_scale**
  get whether the axis has smart bounds

- **XAxis.get_tick_padding**

- **XAxis.get_tick_space**

- **XAxis.get_ticklabel_extents**
  Get the extents of the tick labels on either side of the axes.

- **XAxis.get_ticklabels**
  Get the x tick labels as a list of Text instances.

- **XAxis.get_ticklines**
  Return the tick lines as a list of Line2D instances

Continued on next page
Table 20.18 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XAxis.get_ticklocs</td>
<td>Get the tick locations in data coordinates as a numpy array</td>
</tr>
<tr>
<td>XAxis.get_tightbbox</td>
<td>Return a bounding box that encloses the axis.</td>
</tr>
<tr>
<td>XAxis.get_units</td>
<td>Return the units for axis</td>
</tr>
<tr>
<td>XAxis.get_view_interval</td>
<td>return the Interval instance for this axis view limits</td>
</tr>
<tr>
<td>XAxis.grid</td>
<td>Set the axis grid on or off; b is a boolean.</td>
</tr>
<tr>
<td>XAxis.iter_ticks</td>
<td>Iterate through all of the major and minor ticks.</td>
</tr>
<tr>
<td>XAxis.limit_range_for_scale</td>
<td>Set the axis data limits</td>
</tr>
<tr>
<td>XAxis.pan</td>
<td>Pan numsteps (can be positive or negative)</td>
</tr>
<tr>
<td>XAxis.reset_ticks</td>
<td>set the default limits for the axis interval if they are not mutated</td>
</tr>
<tr>
<td>XAxis.set_data_interval</td>
<td>Set the coordinates of the label.</td>
</tr>
<tr>
<td>XAxis.set_default_intervals</td>
<td>Set the label position (top or bottom)</td>
</tr>
<tr>
<td>XAxis.set_label_text</td>
<td>Sets the text value of the axis label</td>
</tr>
<tr>
<td>XAxis.set_major_formatter</td>
<td>Set the formatter of the major ticker</td>
</tr>
<tr>
<td>XAxis.set_major_locator</td>
<td>Set the locator of the major ticker</td>
</tr>
<tr>
<td>XAxis.set_minor_formatter</td>
<td>Set the formatter of the minor ticker</td>
</tr>
<tr>
<td>XAxis.set_minor_locator</td>
<td>Set the locator of the minor ticker</td>
</tr>
<tr>
<td>XAxis.set_pickradius</td>
<td>Set the depth of the axis used by the picker</td>
</tr>
<tr>
<td>XAxis.set_smart_bounds</td>
<td>set the axis to have smart bounds</td>
</tr>
<tr>
<td>XAxis.set_tick_params</td>
<td>Set appearance parameters for ticks and ticklabels.</td>
</tr>
<tr>
<td>XAxis.set_ticklabels</td>
<td>Set the text values of the tick labels.</td>
</tr>
<tr>
<td>XAxis.set_ticks</td>
<td>Set the locations of the tick marks from sequence ticks</td>
</tr>
<tr>
<td>XAxis.set_units</td>
<td>set the units for axis</td>
</tr>
<tr>
<td>XAxis.set_view_interval</td>
<td>If ignore is False, the order of vmin, vmax does not matter; the original axis orientation will be preserved.</td>
</tr>
<tr>
<td>XAxis.update_units</td>
<td>introspect data for units converter and update the</td>
</tr>
<tr>
<td>XAxis.zoom</td>
<td>Zoom in/out on axis; if direction is &gt;0 zoom in, else zoom out</td>
</tr>
</tbody>
</table>

matplotlib.axis.XAxis.OFFSETTEXTPAD

XAxis.OFFSETTEXTPAD = 3

matplotlib.axis.XAxis.axis_date

XAxis.axis_date(tz=None)

Sets up x-axis ticks and labels that treat the x data as dates. tz is a tzinfo instance or a timezone string. This timezone is used to create date labels.
matplotlib.axis.XAxis.cla

XAxis.cla()
clear the current axis

matplotlib.axis.XAxis.convert_units

XAxis.convert_units(x)

matplotlib.axis.XAxis.get_data_interval

XAxis.get_data_interval()
return the Interval instance for this axis data limits

matplotlib.axis.XAxis.get_gridlines

XAxis.get_gridlines()
Return the grid lines as a list of Line2D instance

matplotlib.axis.XAxis.get_label_position

XAxis.get_label_position()
Return the label position (top or bottom)

matplotlib.axis.XAxis.get_label_text

XAxis.get_label_text()
Get the text of the label

matplotlib.axis.XAxis.get_major_formatter

XAxis.get_major_formatter()
Get the formatter of the major ticker

matplotlib.axis.XAxis.get_major_locator

XAxis.get_major_locator()
Get the locator of the major ticker
**matplotlib.axis.XAxis.get_major_ticks**

XAxis.get_major_ticks(numticks=None)
get the tick instances; grow as necessary

**matplotlib.axis.XAxis.get_major_ticklabels**

XAxis.get_major_ticklabels()
Return a list of Text instances for the major ticklabels

**matplotlib.axis.XAxis.get_major_ticklines**

XAxis.get_major_ticklines()
Return the major tick lines as a list of Line2D instances

**matplotlib.axis.XAxis.get_major_ticklocs**

XAxis.get_major_ticklocs()
Get the major tick locations in data coordinates as a numpy array

**matplotlib.axis.XAxis.get_minor_formatter**

XAxis.get_minor_formatter()
Get the formatter of the minor ticker

**matplotlib.axis.XAxis.get_minor_locator**

XAxis.get_minor_locator()
Get the locator of the minor ticker

**matplotlib.axis.XAxis.get_minor_ticks**

XAxis.get_minor_ticks(numticks=None)
get the minor tick instances; grow as necessary

**matplotlib.axis.XAxis.get_minorticklabels**

XAxis.get_minorticklabels()
Return a list of Text instances for the minor ticklabels
matplotlib.axis.XAxis.get_minorticklines

XAxis.get_minorticklines()
Return the minor tick lines as a list of Line2D instances

matplotlib.axis.XAxis.get_minorticklocs

XAxis.get_minorticklocs()
Get the minor tick locations in data coordinates as a numpy array

matplotlib.axis.XAxis.get_minpos

XAxis.get_minpos()

matplotlib.axis.XAxis.get_offset_text

XAxis.get_offset_text()
Return the axis offsetText as a Text instance

matplotlib.axis.XAxis.get_pickradius

XAxis.get_pickradius()
Return the depth of the axis used by the picker

matplotlib.axis.XAxis.get_scale

XAxis.get_scale()

matplotlib.axis.XAxis.get_smart_bounds

XAxis.get_smart_bounds()
get whether the axis has smart bounds

matplotlib.axis.XAxis.get_tick_padding

XAxis.get_tick_padding()
**matplotlib.axis.XAxis.get_tick_space**

`XAxis.get_tick_space()`

**matplotlib.axis.XAxis.get_ticklabel_extents**

`XAxis.get_ticklabel_extents(renderer)`

Get the extents of the tick labels on either side of the axes.

**matplotlib.axis.XAxis.get_ticklabels**

`XAxis.get_ticklabels(minor=False, which=None)`

Get the x tick labels as a list of `Text` instances.

- **Parameters**
  - `minor`: bool
    - If True return the minor ticklabels, else return the major ticklabels
  - `which`: None, (‘minor’, ‘major’, ‘both’)
    - Overrides `minor`.
    - Selects which ticklabels to return

- **Returns**
  - `ret`: list
    - List of `Text` instances.

**matplotlib.axis.XAxis.get_ticklines**

`XAxis.get_ticklines(minor=False)`

Return the tick lines as a list of `Line2D` instances.

**matplotlib.axis.XAxis.get_ticklocs**

`XAxis.get_ticklocs(minor=False)`

Get the tick locations in data coordinates as a numpy array.

**matplotlib.axis.XAxis.get_tightbbox**

`XAxis.get_tightbbox(renderer)`

Return a bounding box that encloses the axis. It only accounts tick labels, axis label, and offsetText.

**matplotlib.axis.XAxis.get_units**

`XAxis.get_units()`

return the units for axis
**matplotlib.axis.XAxis.get_view_interval**

XAxis.get_view_interval()
return the Interval instance for this axis view limits

**matplotlib.axis.XAxis.grid**

XAxis.grid(b=None, which='major', **kwargs)
Set the axis grid on or off; b is a boolean. Use which = ‘major’ | ‘minor’ | ‘both’ to set the grid for major or minor ticks.

If b is None and len(kwargs)==0, toggle the grid state. If kwargs are supplied, it is assumed you want the grid on and b will be set to True.

kwargs are used to set the line properties of the grids, e.g.,

```python
xax.grid(color='r', linestyle='-', linewidth=2)
```

**matplotlib.axis.XAxis.iter_ticks**

XAxis.iter_ticks()
Iterate through all of the major and minor ticks.

**matplotlib.axis.XAxis.limit_range_for_scale**

XAxis.limit_range_for_scale(vmin, vmax)

**matplotlib.axis.XAxis.pan**

XAxis.pan(numsteps)
Pan numsteps (can be positive or negative)

**matplotlib.axis.XAxis.reset_ticks**

XAxis.reset_ticks()

**matplotlib.axis.XAxis.set_data_interval**

XAxis.set_data_interval(vmin, vmax, ignore=False)
set the axis data limits
The `set_default_intervals` method of the `XAxis` class sets the default limits for the axis interval if they are not mutated.

```
XAxis.set_default_intervals()
```

The `set_label_coords` method of the `XAxis` class sets the coordinates of the label. By default, the x coordinate of the y label is determined by the tick label bounding boxes, but this can lead to poor alignment of multiple ylabels if there are multiple axes. Ditto for the y coordinate of the x label.

```
XAxis.set_label_coords(x, y, transform=None)
```

You can also specify the coordinate system of the label with the transform. If None, the default coordinate system will be the axes coordinate system (0,0) is (left, bottom), (0.5, 0.5) is middle, etc.

The `set_label_position` method of the `XAxis` class sets the label position (top or bottom).

```
XAxis.set_label_position(position)
```

Accepts: ['top' | 'bottom']

The `set_label_text` method of the `XAxis` class sets the text value of the axis label.

```
XAxis.set_label_text(label, fontdict=None, **kwargs)
```

Accepts: A string value for the label

The `set_major_formatter` method of the `XAxis` class sets the formatter of the major ticker.

```
XAxis.set_major_formatter(formatter)
```

Accepts: A `Formatter` instance

The `set_major_locator` method of the `XAxis` class sets the locator of the major ticker.

```
XAxis.set_major_locator(locator)
```

Accepts: a `Locator` instance
**matplotlib.axis.XAxis.set_minor_formatter**

`XAxis.set_minor_formatter(formatter)`
Set the formatter of the minor ticker

**ACCEPTS:** A `Formatter` instance

**matplotlib.axis.XAxis.set_minor_locator**

`XAxis.set_minor_locator(locator)`
Set the locator of the minor ticker

**ACCEPTS:** a `Locator` instance

**matplotlib.axis.XAxis.set_pickradius**

`XAxis.set_pickradius(pickradius)`
Set the depth of the axis used by the picker

**ACCEPTS:** a distance in points

**matplotlib.axis.XAxis.set_smart_bounds**

`XAxis.set_smart_bounds(value)`
set the axis to have smart bounds

**matplotlib.axis.XAxis.set_tick_params**

`XAxis.set_tick_params(which='major', reset=False, **kw)`
Set appearance parameters for ticks and ticklabels.

For documentation of keyword arguments, see `matplotlib.axes.Axes.tick_params()`.

**matplotlib.axis.XAxis.set_ticklabels**

`XAxis.set_ticklabels(ticklabels, *args, **kwargs)`
Set the text values of the tick labels. Return a list of Text instances. Use `kwarg minor=True` to select minor ticks. All other kwargs are used to update the text object properties. As for `get_ticklabels`, `label1` (left or bottom) is affected for a given tick only if its `label1On` attribute is True, and similarly for `label2`. The list of returned label text objects consists of all such `label1` objects followed by all such `label2` objects.

The input `ticklabels` is assumed to match the set of tick locations, regardless of the state of `label1On` and `label2On`.

**ACCEPTS:** sequence of strings or Text objects
matplotlib.axis.XAxis.set_ticks

XAxis.set_ticks(ticks, minor=False)

Set the locations of the tick marks from sequence ticks
ACCEPENTS: sequence of floats

matplotlib.axis.XAxis.set_units

XAxis.set_units(u)

set the units for axis
ACCEPENTS: a units tag

matplotlib.axis.XAxis.set_view_interval

XAxis.set_view_interval(vmin, vmax, ignore=False)

If ignore is False, the order of vmin, vmax does not matter; the original axis orientation will be preserved. In addition, the view limits can be expanded, but will not be reduced. This method is for mpl internal use; for normal use, see set_xlim().

matplotlib.axis.XAxis.update_units

XAxis.update_units(data)

introspect data for units converter and update the axis.converter instance if necessary. Return True if data is registered for unit conversion.

matplotlib.axis.XAxis.zoom

XAxis.zoom(direction)

Zoom in/out on axis; if direction is >0 zoom in, else zoom out

20.4.5 Inherited from artist

Ticks

<table>
<thead>
<tr>
<th>Tick.add_callback</th>
<th>Adds a callback function that will be called whenever one of the Artist's properties changes.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tick.aname</td>
<td></td>
</tr>
<tr>
<td>Tick.axes</td>
<td>The Axes instance the artist resides in, or None.</td>
</tr>
<tr>
<td>Tick.contains</td>
<td>Test whether the mouse event occurred in the Tick marks.</td>
</tr>
<tr>
<td>Tick.convert_xunits</td>
<td>For artists in an axes, if the xaxis has units support,</td>
</tr>
</tbody>
</table>
Table 20.19 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Tick.convert_yunits</code></td>
<td>For artists in an axes, if the yaxis has units support,</td>
</tr>
<tr>
<td><code>Tick.draw</code></td>
<td></td>
</tr>
<tr>
<td><code>Tick.findobj</code></td>
<td>Find artist objects.</td>
</tr>
<tr>
<td><code>Tick.format_cursor_data</code></td>
<td>Return cursor data string formatted.</td>
</tr>
<tr>
<td><code>Tick.get_agg_filter</code></td>
<td>Return filter function to be used for agg filter.</td>
</tr>
<tr>
<td><code>Tick.get_alpha</code></td>
<td>Return the alpha value used for blending - not supported on all.</td>
</tr>
<tr>
<td><code>Tick.get_animated</code></td>
<td>Return the artist’s animated state.</td>
</tr>
<tr>
<td><code>Tick.get_axes</code></td>
<td>Return the Axes instance the artist resides in, or None.</td>
</tr>
<tr>
<td><code>Tick.get_children</code></td>
<td></td>
</tr>
<tr>
<td><code>Tick.get_clip_box</code></td>
<td>Return artist clipbox.</td>
</tr>
<tr>
<td><code>Tick.get_clip_on</code></td>
<td>Return whether artist uses clipping.</td>
</tr>
<tr>
<td><code>Tick.get_clip_path</code></td>
<td>Return artist clip path.</td>
</tr>
<tr>
<td><code>Tick.get_contains</code></td>
<td>Return the _contains test used by the artist, or None for default.</td>
</tr>
<tr>
<td><code>Tick.get_cursor_data</code></td>
<td>Get the cursor data for a given event.</td>
</tr>
<tr>
<td><code>Tick.get_figure</code></td>
<td>Return the Figure instance the artist belongs to.</td>
</tr>
<tr>
<td><code>Tick.get_gid</code></td>
<td>Returns the group id.</td>
</tr>
<tr>
<td><code>Tick.get_label</code></td>
<td>Get the label used for this artist in the legend.</td>
</tr>
<tr>
<td><code>Tick.get_path_effects</code></td>
<td></td>
</tr>
<tr>
<td><code>Tick.get_picker</code></td>
<td>Return the picker object used by this artist.</td>
</tr>
<tr>
<td><code>Tick.get_rasterized</code></td>
<td>Return True if the artist is to be rasterized.</td>
</tr>
<tr>
<td><code>Tick.get_sketch_params</code></td>
<td>Returns the sketch parameters for the artist.</td>
</tr>
<tr>
<td><code>Tick.get_snap</code></td>
<td>Returns the snap setting which may be:</td>
</tr>
<tr>
<td><code>Tick.get_transform</code></td>
<td>Return the Transform instance used by this artist.</td>
</tr>
<tr>
<td><code>Tick.get_transformed_clip_path_and_affine</code></td>
<td>Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.</td>
</tr>
<tr>
<td><code>Tick.get_url</code></td>
<td>Returns the url.</td>
</tr>
<tr>
<td><code>Tick.get_visible</code></td>
<td>Return the artist’s visiblity.</td>
</tr>
<tr>
<td><code>Tick.get_window_extent</code></td>
<td>Get the axes bounding box in display space.</td>
</tr>
<tr>
<td><code>Tick.get_zorder</code></td>
<td>Return the Artist’s zorder.</td>
</tr>
<tr>
<td><code>Tick.have_units</code></td>
<td>Return True if units are set on the x or y axes.</td>
</tr>
<tr>
<td><code>Tick_hitlist</code></td>
<td>List the children of the artist which contain the mouse event event.</td>
</tr>
<tr>
<td><code>Tick.is_figure_set</code></td>
<td>Returns True if the artist is assigned to a Figure.</td>
</tr>
<tr>
<td><code>Tick.is_transform_set</code></td>
<td>Returns True if Artist has a transform explicitly set.</td>
</tr>
<tr>
<td><code>Tick.mouseover</code></td>
<td></td>
</tr>
<tr>
<td><code>Tick.pchanged</code></td>
<td>Fire an event when property changed, calling all of the registered callbacks.</td>
</tr>
<tr>
<td><code>Tick.pick</code></td>
<td>Process pick event.</td>
</tr>
<tr>
<td><code>Tick.pickable</code></td>
<td>Return True if Artist is pickable.</td>
</tr>
<tr>
<td><code>Tick.properties</code></td>
<td>Return a dictionary mapping property name -&gt; value for all Artist props.</td>
</tr>
</tbody>
</table>

Continued on next page
Table 20.19 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tick.remove</td>
<td>Remove the artist from the figure if possible.</td>
</tr>
<tr>
<td>Tick.remove_callback</td>
<td>Remove a callback based on its id.</td>
</tr>
<tr>
<td>Tick.set</td>
<td>A property batch setter.</td>
</tr>
<tr>
<td>Tick.set_agg_filter</td>
<td>Set agg_filter function.</td>
</tr>
<tr>
<td>Tick.set_alpha</td>
<td>Set the alpha value used for blending - not supported on all backends.</td>
</tr>
<tr>
<td>Tick.set_animated</td>
<td>Set the artist’s animation state.</td>
</tr>
<tr>
<td>Tick.set_axes</td>
<td>Set the Axes instance in which the artist resides, if any.</td>
</tr>
<tr>
<td>Tick.set_clip_box</td>
<td>Set the artist’s clip Bbox.</td>
</tr>
<tr>
<td>Tick.set_clip_on</td>
<td>Set whether artist uses clipping.</td>
</tr>
<tr>
<td>Tick.set_clip_path</td>
<td>Set the artist’s clip path, which may be:</td>
</tr>
<tr>
<td>Tick.set_contains</td>
<td>Replace the contains test used by this artist.</td>
</tr>
<tr>
<td>Tick.set_figure</td>
<td>Set the Figure instance the artist belongs to.</td>
</tr>
<tr>
<td>Tick.set_gid</td>
<td>Sets the (group) id for the artist</td>
</tr>
<tr>
<td>Tick.set_label</td>
<td>Set the text of ticklabel</td>
</tr>
<tr>
<td>Tick.set_path_effects</td>
<td>set path_effects, which should be a list of instances of</td>
</tr>
<tr>
<td>Tick.set_picker</td>
<td>Set the epsilon for picking used by this artist</td>
</tr>
<tr>
<td>Tick.set_rasterized</td>
<td>Force rasterized (bitmap) drawing in vector backend output.</td>
</tr>
<tr>
<td>Tick.set_sketch_params</td>
<td>Sets the sketch parameters.</td>
</tr>
<tr>
<td>Tick.set_snap</td>
<td>Sets the snap setting which may be:</td>
</tr>
<tr>
<td>Tick.set_transform</td>
<td>Set the Transform instance used by this artist.</td>
</tr>
<tr>
<td>Tick.set_url</td>
<td>Sets the url for the artist</td>
</tr>
<tr>
<td>Tick.set_visible</td>
<td>Set the artist’s visiblity.</td>
</tr>
<tr>
<td>Tick.set_zorder</td>
<td>Set the zorder for the artist</td>
</tr>
<tr>
<td>Tick.stale</td>
<td>If the artist is ‘stale’ and needs to be re-drawn for the output to match the internal state of the artist.</td>
</tr>
<tr>
<td>Tick.update</td>
<td>Update the properties of this Artist from the dictionary prop.</td>
</tr>
<tr>
<td>Tick.update_from</td>
<td>Copy properties from other to self.</td>
</tr>
<tr>
<td>Tick.zorder</td>
<td>Adds a callback function that will be called whenever one of the Artist’s properties changes.</td>
</tr>
<tr>
<td>XTick.add_callback</td>
<td>Adds a callback function that will be called whenever one of the Artist’s properties changes.</td>
</tr>
<tr>
<td>XTick.aname</td>
<td>The Axes instance the artist resides in, or None.</td>
</tr>
<tr>
<td>XTick.axes</td>
<td>The Axes instance the artist resides in, or None.</td>
</tr>
<tr>
<td>XTick.contains</td>
<td>Test whether the mouse event occurred in the Tick marks.</td>
</tr>
<tr>
<td>XTick.convert_xunits</td>
<td>For artists in an axes, if the xaxis has units support,</td>
</tr>
<tr>
<td>XTick.convert_yunits</td>
<td>For artists in an axes, if the yaxis has units support,</td>
</tr>
<tr>
<td>XTick.draw</td>
<td>Find artist objects.</td>
</tr>
<tr>
<td>XTick.findobj</td>
<td>Return cursor data string formatted.</td>
</tr>
<tr>
<td>XTick.get_agg_filter</td>
<td>return filter function to be used for agg filter</td>
</tr>
</tbody>
</table>
Table 20.19 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>XTick.get_alpha</code></td>
<td>Return the alpha value used for blending - not supported on all</td>
</tr>
<tr>
<td><code>XTick.get_animated</code></td>
<td>Return the artist’s animated state</td>
</tr>
<tr>
<td><code>XTick.get_axes</code></td>
<td>Return the <code>Axes</code> instance the artist resides in, or <code>None</code>.</td>
</tr>
<tr>
<td><code>XTick.get_children</code></td>
<td></td>
</tr>
<tr>
<td><code>XTick.get_clip_box</code></td>
<td>Return artist clipbox</td>
</tr>
<tr>
<td><code>XTick.get_clip_on</code></td>
<td>Return whether artist uses clipping</td>
</tr>
<tr>
<td><code>XTick.get_clip_path</code></td>
<td>Return artist clip path</td>
</tr>
<tr>
<td><code>XTick.get_contains</code></td>
<td>Return the _contains test used by the artist, or <code>None</code> for default.</td>
</tr>
<tr>
<td><code>XTick.get_cursor_data</code></td>
<td>Get the cursor data for a given event.</td>
</tr>
<tr>
<td><code>XTick.get_figure</code></td>
<td>Return the <code>Figure</code> instance the artist belongs to.</td>
</tr>
<tr>
<td><code>XTick.get_gid</code></td>
<td>Returns the group id</td>
</tr>
<tr>
<td><code>XTick.get_label</code></td>
<td>Get the label used for this artist in the legend.</td>
</tr>
<tr>
<td><code>XTick.get_path_effects</code></td>
<td></td>
</tr>
<tr>
<td><code>XTick.get_picker</code></td>
<td>Return the picker object used by this artist</td>
</tr>
<tr>
<td><code>XTick.get_rasterized</code></td>
<td>return True if the artist is to be rasterized</td>
</tr>
<tr>
<td><code>XTick.get_sketch_params</code></td>
<td>Returns the sketch parameters for the artist.</td>
</tr>
<tr>
<td><code>XTick.get_snap</code></td>
<td>Returns the snap setting which may be:</td>
</tr>
<tr>
<td><code>XTick.get_transform</code></td>
<td>Return the <code>Transform</code> instance used by this artist.</td>
</tr>
<tr>
<td><code>XTick.get_transformed_clip_path_and_affine</code></td>
<td>Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.</td>
</tr>
<tr>
<td><code>XTick.get_url</code></td>
<td>Returns the url</td>
</tr>
<tr>
<td><code>XTick.get_visible</code></td>
<td>Return the artist’s visibility</td>
</tr>
<tr>
<td><code>XTick.get_window_extent</code></td>
<td>Get the axes bounding box in display space.</td>
</tr>
<tr>
<td><code>XTick.get_zorder</code></td>
<td>Return the Artist’s zorder</td>
</tr>
<tr>
<td><code>XTick.have_units</code></td>
<td>Return <code>True</code> if units are set on the x or y axes</td>
</tr>
<tr>
<td><code>XTick.hitlist</code></td>
<td>List the children of the artist which contain the mouse event <code>event</code>.</td>
</tr>
<tr>
<td><code>XTick.is_figure_set</code></td>
<td>Returns True if the artist is assigned to a <code>Figure</code>.</td>
</tr>
<tr>
<td><code>XTick.is_transform_set</code></td>
<td>Returns <code>True</code> if <code>Artist</code> has a transform explicitly set.</td>
</tr>
<tr>
<td><code>XTick.mouseover</code></td>
<td></td>
</tr>
<tr>
<td><code>XTick.pchanged</code></td>
<td>Fire an event when property changed, calling all of the registered callbacks.</td>
</tr>
<tr>
<td><code>XTick.pick</code></td>
<td>Process pick event</td>
</tr>
<tr>
<td><code>XTick.pickle</code></td>
<td>Return <code>True</code> if <code>Artist</code> is pickable.</td>
</tr>
<tr>
<td><code>XTick.properties</code></td>
<td>return a dictionary mapping property name - <code>value</code> for all <code>Artist</code> props</td>
</tr>
<tr>
<td><code>XTick.remove</code></td>
<td>Remove the artist from the figure if possible.</td>
</tr>
<tr>
<td><code>XTick.remove_callback</code></td>
<td>Remove a callback based on its <code>id</code>.</td>
</tr>
<tr>
<td><code>XTick.set</code></td>
<td>A property batch setter.</td>
</tr>
<tr>
<td><code>XTick.set_agg_filter</code></td>
<td>set agg_filter fuction.</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>XTick.set_alpha</code></td>
<td>Set the alpha value used for blending - not supported on all backends.</td>
</tr>
<tr>
<td><code>XTick.set_animated</code></td>
<td>Set the artist’s animation state.</td>
</tr>
<tr>
<td><code>XTick.set_axes</code></td>
<td>Set the <code>Axes</code> instance in which the artist resides, if any.</td>
</tr>
<tr>
<td><code>XTick.set_clip_box</code></td>
<td>Set the artist’s clip <code>Bbox</code>.</td>
</tr>
<tr>
<td><code>XTick.set_clip_on</code></td>
<td>Set whether artist uses clipping.</td>
</tr>
<tr>
<td><code>XTick.set_clip_path</code></td>
<td>Replace the contains test used by this artist.</td>
</tr>
<tr>
<td><code>XTick.set_figure</code></td>
<td>Set the <code>Figure</code> instance the artist belongs to.</td>
</tr>
<tr>
<td><code>XTick.set_gid</code></td>
<td>Sets the (group) id for the artist.</td>
</tr>
<tr>
<td><code>XTick.set_label</code></td>
<td>Set the text of ticklabel.</td>
</tr>
<tr>
<td><code>XTick.set_path_effects</code></td>
<td>Set path effects, which should be a list of instances of</td>
</tr>
<tr>
<td><code>XTick.set_picker</code></td>
<td>Set the epsilon for picking used by this artist.</td>
</tr>
<tr>
<td><code>XTick.set_rasterized</code></td>
<td>Force rasterized (bitmap) drawing in vector backend output.</td>
</tr>
<tr>
<td><code>XTick.set_sketch_params</code></td>
<td>Sets the sketch parameters.</td>
</tr>
<tr>
<td><code>XTick.set_snap</code></td>
<td>Sets the snap setting which may be:</td>
</tr>
<tr>
<td><code>XTick.set_transform</code></td>
<td>Set the <code>Transform</code> instance used by this artist.</td>
</tr>
<tr>
<td><code>XTick.set_url</code></td>
<td>Sets the url for the artist.</td>
</tr>
<tr>
<td><code>XTick.set_visible</code></td>
<td>Set the artist’s visibility.</td>
</tr>
<tr>
<td><code>XTick.set_zorder</code></td>
<td>Set the zorder for the artist.</td>
</tr>
<tr>
<td><code>XTick.stale</code></td>
<td>If the artist is ‘stale’ and needs to be re-drawn for the output to match the internal state of the artist.</td>
</tr>
<tr>
<td><code>XTick.update</code></td>
<td>Update the properties of this <code>Artist</code> from the dictionary <code>prop</code>.</td>
</tr>
<tr>
<td><code>XTick.update_from</code></td>
<td>Copy properties from other to <code>self</code>.</td>
</tr>
<tr>
<td><code>XTick.zorder</code></td>
<td></td>
</tr>
<tr>
<td><code>YTick.add_callback</code></td>
<td>Adds a callback function that will be called whenever one of the <code>Artist</code>’s properties changes.</td>
</tr>
<tr>
<td><code>YTick.aname</code></td>
<td></td>
</tr>
<tr>
<td><code>YTick.axes</code></td>
<td>The <code>Axes</code> instance the artist resides in, or <code>None</code>.</td>
</tr>
<tr>
<td><code>YTick.contains</code></td>
<td>Test whether the mouse event occurred in the Tick marks.</td>
</tr>
<tr>
<td><code>YTick.convert_xunits</code></td>
<td>For artists in an axes, if the xaxis has units support.</td>
</tr>
<tr>
<td><code>YTick.convert_yunits</code></td>
<td>For artists in an axes, if the yaxis has units support.</td>
</tr>
<tr>
<td><code>YTick.draw</code></td>
<td></td>
</tr>
<tr>
<td><code>YTick.findobj</code></td>
<td>Find artist objects.</td>
</tr>
<tr>
<td><code>YTick.format_cursor_data</code></td>
<td>Return <code>cursor data</code> string formatted.</td>
</tr>
<tr>
<td><code>YTick.get_agg_filter</code></td>
<td>return filter function to be used for agg filter</td>
</tr>
<tr>
<td><code>YTick.get_alpha</code></td>
<td>Return the alpha value used for blending - not supported on all.</td>
</tr>
<tr>
<td><code>YTick.get_animated</code></td>
<td>Return the artist’s animated state.</td>
</tr>
<tr>
<td><code>YTick.get_axes</code></td>
<td>Return the <code>Axes</code> instance the artist resides in, or <code>None</code>.</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>YTick.get_children</code></td>
<td>Return artist clipbox</td>
</tr>
<tr>
<td><code>YTick.get_clip_box</code></td>
<td>Return whether artist uses clipping</td>
</tr>
<tr>
<td><code>YTick.get_clip_on</code></td>
<td>Return artist clip path</td>
</tr>
<tr>
<td><code>YTick.get_clip_path</code></td>
<td>Return the _contains test used by the artist, or <em>None</em> for default.</td>
</tr>
<tr>
<td><code>YTick.get_cursor_data</code></td>
<td>Get the cursor data for a given event.</td>
</tr>
<tr>
<td><code>YTick.get_figure</code></td>
<td>Return the <em>Figure</em> instance the artist belongs to.</td>
</tr>
<tr>
<td><code>YTick.get_gid</code></td>
<td>Returns the group id</td>
</tr>
<tr>
<td><code>YTick.get_label</code></td>
<td>Get the label used for this artist in the legend.</td>
</tr>
<tr>
<td><code>YTick.get_path_effects</code></td>
<td></td>
</tr>
<tr>
<td><code>YTick.get_picker</code></td>
<td>Return the picker object used by this artist</td>
</tr>
<tr>
<td><code>YTick.get_rasterized</code></td>
<td>return True if the artist is to be rasterized</td>
</tr>
<tr>
<td><code>YTick.get_sketch_params</code></td>
<td>Returns the sketch parameters for the artist.</td>
</tr>
<tr>
<td><code>YTick.get_snap</code></td>
<td>Returns the snap setting which may be:</td>
</tr>
<tr>
<td><code>YTick.get_transform</code></td>
<td>Return the <em>Transform</em> instance used by this artist.</td>
</tr>
<tr>
<td><code>YTick.get_transformed_clip_path_and_affine</code></td>
<td>Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.</td>
</tr>
<tr>
<td><code>YTick.get_url</code></td>
<td>Returns the url</td>
</tr>
<tr>
<td><code>YTick.get_visible</code></td>
<td>Return the artist’s visibility</td>
</tr>
<tr>
<td><code>YTick.get_window_extent</code></td>
<td>Get the axes bounding box in display space.</td>
</tr>
<tr>
<td><code>YTick.get_zorder</code></td>
<td>Return the Artist’s zorder.</td>
</tr>
<tr>
<td><code>YTick.have_units</code></td>
<td>Return <em>True</em> if units are set on the x or y axes</td>
</tr>
<tr>
<td><code>YTick.hitlist</code></td>
<td>List the children of the artist which contain the mouse event <em>event</em>.</td>
</tr>
<tr>
<td><code>YTick.is_figure_set</code></td>
<td>Returns <em>True</em> if the artist is assigned to a <em>Figure</em>.</td>
</tr>
<tr>
<td><code>YTick.is_transform_set</code></td>
<td>Returns <em>True</em> if <em>Artist</em> has a transform explicitly set.</td>
</tr>
<tr>
<td><code>YTick.mouseover</code></td>
<td></td>
</tr>
<tr>
<td><code>YTick.pchanged</code></td>
<td>Fire an event when property changed, calling all of the registered callbacks.</td>
</tr>
<tr>
<td><code>YTick.pick</code></td>
<td>Process pick event</td>
</tr>
<tr>
<td><code>YTick.pickable</code></td>
<td>Return <em>True</em> if <em>Artist</em> is pickable.</td>
</tr>
<tr>
<td><code>YTick.properties</code></td>
<td>return a dictionary mapping property name -&gt; value for all <em>Artist</em> props</td>
</tr>
<tr>
<td><code>YTick.remove</code></td>
<td>Remove the artist from the figure if possible.</td>
</tr>
<tr>
<td><code>YTick.remove_callback</code></td>
<td>Remove a callback based on its <em>id</em>.</td>
</tr>
<tr>
<td><code>YTick.set</code></td>
<td>A property batch setter.</td>
</tr>
<tr>
<td><code>YTick.set_agg_filter</code></td>
<td>set agg_filter function</td>
</tr>
<tr>
<td><code>YTick.set_alpha</code></td>
<td>Set the alpha value used for blending - not supported on all backends.</td>
</tr>
<tr>
<td><code>YTick.set_animated</code></td>
<td>Set the artist’s animation state.</td>
</tr>
<tr>
<td><code>YTick.set_axes</code></td>
<td>Set the <em>Axes</em> instance in which the artist resides, if any.</td>
</tr>
</tbody>
</table>

Continued on next page
Table 20.19 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>YTick.set_clip_box</code></td>
<td>Set the artist’s clip <code>Bbox</code>.</td>
</tr>
<tr>
<td><code>YTick.set_clip_on</code></td>
<td>Set whether artist uses clipping.</td>
</tr>
<tr>
<td><code>YTick.set_clip_path</code></td>
<td>Set the artist’s clip path, which may be:</td>
</tr>
<tr>
<td><code>YTick.set_contains</code></td>
<td>Replace the contains test used by this artist.</td>
</tr>
<tr>
<td><code>YTick.set_figure</code></td>
<td>Set the <code>Figure</code> instance the artist belongs to.</td>
</tr>
<tr>
<td><code>YTick.set_gid</code></td>
<td>Sets the (group) id for the artist</td>
</tr>
<tr>
<td><code>YTick.set_label</code></td>
<td>Set the text of ticklabel</td>
</tr>
<tr>
<td><code>YTick.set_path_effects</code></td>
<td>Set <code>path_effects</code>, which should be a list of instances of <code>PathEffects</code></td>
</tr>
<tr>
<td><code>YTick.set_picker</code></td>
<td>Set the epsilon for picking used by this artist</td>
</tr>
<tr>
<td><code>YTick.set_rasterized</code></td>
<td>Force rasterized (bitmap) drawing in vector backend output.</td>
</tr>
<tr>
<td><code>YTick.set_sketch_params</code></td>
<td>Set the sketch parameters.</td>
</tr>
<tr>
<td><code>YTick.set_snap</code></td>
<td>Sets the snap setting which may be:</td>
</tr>
<tr>
<td><code>YTick.set_transform</code></td>
<td>Set the <code>Transform</code> instance used by this artist</td>
</tr>
<tr>
<td><code>YTick.set_url</code></td>
<td>Sets the url for the artist</td>
</tr>
<tr>
<td><code>YTick.set_visible</code></td>
<td>Set the artist’s visibility.</td>
</tr>
<tr>
<td><code>YTick.set_zorder</code></td>
<td>Set the zorder for the artist</td>
</tr>
<tr>
<td><code>YTick.stale</code></td>
<td>If the artist is ‘stale’ and needs to be re-drawn for the output to match</td>
</tr>
<tr>
<td></td>
<td>the internal state of the artist.</td>
</tr>
<tr>
<td><code>YTick.update</code></td>
<td>Update the properties of this <code>Artist</code> from the dictionary <code>prop</code>.</td>
</tr>
<tr>
<td><code>YTick.update_from</code></td>
<td>Copy properties from <code>other</code> to <code>self</code>.</td>
</tr>
<tr>
<td><code>YTick.zorder</code></td>
<td></td>
</tr>
</tbody>
</table>

**matplotlib.axis.Tick.add_callback**

Tick.add_callback(*func*)

Adds a callback function that will be called whenever one of the `Artist`'s properties changes.

Returns an `id` that is useful for removing the callback with remove_callback() later.

**matplotlib.axis.Tick.aname**

Tick.aname = ‘Artist’

**matplotlib.axis.Tick.axes**

Tick.axes

The `Axes` instance the artist resides in, or `None`. 
**matplotlib.axis.Tick.contains**

Tick.contains(mouseevent)

Test whether the mouse event occurred in the Tick marks.

This function always returns false. It is more useful to test if the axis as a whole contains the mouse rather than the set of tick marks.

**matplotlib.axis.Tick.convert_xunits**

Tick.convert_xunits(x)

For artists in an axes, if the xaxis has units support, convert x using xaxis unit type

**matplotlib.axis.Tick.convert_yunits**

Tick.convert_yunits(y)

For artists in an axes, if the yaxis has units support, convert y using yaxis unit type

**matplotlib.axis.Tick.draw**

Tick.draw(artist, renderer, *args, **kwargs)

**matplotlib.axis.Tick.findobj**

Tick.findobj(match=None, include_self=True)

Find artist objects.

Recursively find all *Artist* instances contained in self.

match can be

- None: return all objects contained in artist.
- function with signature boolean = match(artist) used to filter matches
- class instance: e.g., Line2D. Only return artists of class type.

If include_self is True (default), include self in the list to be checked for a match.

**matplotlib.axis.Tick.format_cursor_data**

Tick.format_cursor_data(data)

Return cursor data string formatted.
**matplotlib.axis.Tick.get_agg_filter**

`Tick.get_agg_filter()`  
return filter function to be used for agg filter

**matplotlib.axis.Tick.get_alpha**

`Tick.get_alpha()`  
Return the alpha value used for blending - not supported on all backends

**matplotlib.axis.Tick.get_animated**

`Tick.get_animated()`  
Return the artist’s animated state

**matplotlib.axis.Tick.get_axes**

`Tick.get_axes()`  
Return the `Axes` instance the artist resides in, or `None`.  
This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

**matplotlib.axis.Tick.get_clip_box**

`Tick.get_clip_box()`  
Return artist clipbox

**matplotlib.axis.Tick.get_clip_on**

`Tick.get_clip_on()`  
Return whether artist uses clipping

**matplotlib.axis.Tick.get_clip_path**

`Tick.get_clip_path()`  
Return artist clip path
matplotlib.axis.Tick.get_contains

Tick.get_contains()
    Return the _contains test used by the artist, or None for default.

matplotlib.axis.Tick.get_cursor_data

Tick.get_cursor_data(event)
    Get the cursor data for a given event.

matplotlib.axis.Tick.get_figure

Tick.get_figure()
    Return the Figure instance the artist belongs to.

matplotlib.axis.Tick.get_gid

Tick.get_gid()
    Returns the group id

matplotlib.axis.Tick.get_label

Tick.get_label()
    Get the label used for this artist in the legend.

matplotlib.axis.Tick.get_path_effects

Tick.get_path_effects()

matplotlib.axis.Tick.get_picker

Tick.get_picker()
    Return the picker object used by this artist

matplotlib.axis.Tick.get_rasterized

Tick.get_rasterized()
    return True if the artist is to be rasterized
**matplotlib.axis.Tick.get_sketch_params**

Tick.get_sketch_params()

Returns the sketch parameters for the artist.

**Returns** `sketch_params` : tuple or `None`

A 3-tuple with the following elements:

- `scale`: The amplitude of the wiggle perpendicular to the source line.
- `length`: The length of the wiggle along the line.
- `randomness`: The scale factor by which the length is shrunken or expanded.

May return `None` if no sketch parameters were set.

**matplotlib.axis.Tick.get_snap**

Tick.get_snap()

Returns the snap setting which may be:

- `True`: snap vertices to the nearest pixel center
- `False`: leave vertices as-is
- `None`: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

**matplotlib.axis.Tick.get_transform**

Tick.get_transform()

Return the `Transform` instance used by this artist.

**matplotlib.axis.Tick.get_transformed_clip_path_and_affine**

Tick.get_transformed_clip_path_and_affine()

Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

**matplotlib.axis.Tick.get_url**

Tick.get_url()

Returns the url
Matplotlib, Release 2.0.2

**matplotlib.axis.Tick.get_visible**

Tick.get_visible()

Return the artist’s visibility

**matplotlib.axis.Tick.get_window_extent**

Tick.get_window_extent(renderer)

Get the axes bounding box in display space. Subclasses should override for inclusion in the bounding box “tight” calculation. Default is to return an empty bounding box at 0, 0.

Be careful when using this function, the results will not update if the artist window extent of the artist changes. The extent can change due to any changes in the transform stack, such as changing the axes limits, the figure size, or the canvas used (as is done when saving a figure). This can lead to unexpected behavior where interactive figures will look fine on the screen, but will save incorrectly.

**matplotlib.axis.Tick.get_zorder**

Tick.get_zorder()

Return the Artist’s zorder.

**matplotlib.axis.Tick.have_units**

Tick.have_units()

Return True if units are set on the x or y axes

**matplotlib.axis.Tick.hitlist**

Tick.hitlist(event)

List the children of the artist which contain the mouse event event.

**matplotlib.axis.Tick.is_figure_set**

Tick.is_figure_set()

Returns True if the artist is assigned to a Figure.

**matplotlib.axis.Tick.is_transform_set**

Tick.is_transform_set()

Returns True if Artist has a transform explicitly set.
**matplotlib.axis.Tickmouseover**

Tick.mouseover

**matplotlib.axis.Tick.pchanged**

Tick.pchanged()

Fire an event when property changed, calling all of the registered callbacks.

**matplotlib.axis.Tick.pick**

Tick.pick(mouseevent)

Process pick event

each child artist will fire a pick event if mouseevent is over the artist and the artist has picker set

**matplotlib.axis.Tick.pickable**

Tick.pickable()

Return True if Artist is pickable.

**matplotlib.axis.Tick.properties**

Tick.properties()

return a dictionary mapping property name -> value for all Artist props

**matplotlib.axis.Tick.remove**

Tick.remove()

Remove the artist from the figure if possible. The effect will not be visible until the figure is redrawn, e.g., with matplotlib.axes.Axes.draw_idle(). Call matplotlib.axes.Axes.relim() to update the axes limits if desired.

Note: relim() will not see collections even if the collection was added to axes with autolim = True.

Note: there is no support for removing the artist’s legend entry.

**matplotlib.axis.Tick.remove_callback**

Tick.remove_callback(oid)

Remove a callback based on its id.

See also:
.. add_callback():: For adding callbacks

.. function:: matplotlib.axis.Tick.set(**kwargs)

   A property batch setter. Pass **kwargs to set properties.

.. function:: matplotlib.axis.Tick.set_agg_filter(filter_func)

   Set agg_filter function.

.. function:: matplotlib.axis.Tick.set_alpha(alpha)

   Set the alpha value used for blending - not supported on all backends.

   ACCEPTS: float (0.0 transparent through 1.0 opaque)

.. function:: matplotlib.axis.Tick.set_animated(b)

   Set the artist’s animation state.

   ACCEPTS: [True | False]

.. function:: matplotlib.axis.Tick.set_axes(axes)

   Set the `Axes` instance in which the artist resides, if any.

   This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

   ACCEPTS: an `Axes` instance

.. function:: matplotlib.axis.Tick.set_clip_box(clipbox)

   Set the artist’s clip `Bbox`.

   ACCEPTS: a `matplotlib.transforms.Bbox` instance
**matplotlib.axis.Tick.set_clip_on**

Tick.set_clip_on(b)

Set whether artist uses clipping.

When False artists will be visible out side of the axes which can lead to unexpected results.

ACCEPTS: [True | False]

**matplotlib.axis.Tick.set_clip_path**

Tick.set_clip_path(clippath, transform=None)

Set the artist’s clip path, which may be:

- a Patch (or subclass) instance
- a Path instance, in which case an optional Transform instance may be provided, which will be applied to the path before using it for clipping.
- None, to remove the clipping path

For efficiency, if the path happens to be an axis-aligned rectangle, this method will set the clipping box to the corresponding rectangle and set the clipping path to None.

ACCEPTS: [(Path, Transform) | Patch | None ]

**matplotlib.axis.Tick.set_contains**

Tick.set_contains(picker)

Replace the contains test used by this artist. The new picker should be a callable function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

If the mouse event is over the artist, return hit = True and props is a dictionary of properties you want returned with the contains test.

ACCEPTS: a callable function

**matplotlib.axis.Tick.set_figure**

Tick.set_figure(fig)

Set the Figure instance the artist belongs to.

ACCEPTS: a matplotlib.figure.Figure instance
matplotlib.axis.Tick.set_gid

Tick.set_gid(gid)
Sets the (group) id for the artist

ACCEPTS: an id string

matplotlib.axis.Tick.set_label

Tick.set_label(s)
Set the text of ticklabel

ACCEPTS: str

matplotlib.axis.Tick.set_path_effects

Tick.set_path_effects(path_effects)
set path_effects, which should be a list of instances of matplotlib.path_effect._Base class or its derivatives.

matplotlib.axis.Tick.set_picker

Tick.set_picker(picker)
Set the epsilon for picking used by this artist

picker can be one of the following:

• None: picking is disabled for this artist (default)
• A boolean: if True then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist
• A float: if picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if it’s data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event
• A function: if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```python
hit, props = picker(artist, mouseevent)
```

to determine the hit test. if the mouse event is over the artist, return hit=True and props is a dictionary of properties you want added to the PickEvent attributes.

ACCEPTS: [None|float|boolean|callable]
**matplotlib.axis.Tick.set_rasterized**

`Tick.set_rasterized(rasterized)`

Force rasterized (bitmap) drawing in vector backend output.

Defaults to None, which implies the backend’s default behavior

ACCEPTS: [True | False | None]

**matplotlib.axis.Tick.set_sketch_params**

`Tick.set_sketch_params(scale=None, length=None, randomness=None)`

Sets the sketch parameters.

**Parameters**

- `scale` : float, optional
  The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is `None`, or not provided, no sketch filter will be provided.

- `length` : float, optional
  The length of the wiggle along the line, in pixels (default 128.0)

- `randomness` : float, optional
  The scale factor by which the length is shrunk or expanded (default 16.0)

**matplotlib.axis.Tick.set_snap**

`Tick.set_snap(snapshot)`

Sets the snap setting which may be:

- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

**matplotlib.axis.Tick.set_transform**

`Tick.set_transform(t)`

Set the `Transform` instance used by this artist.

ACCEPTS: `Transform` instance
**Matplotlib, Release 2.0.2**

`matplotlib.axis.Tick.set_url`

`Tick.set_url(url)`
Sets the url for the artist

**ACCEPTS:** a url string

`matplotlib.axis.Tick.set_visible`

`Tick.set_visible(b)`
Set the artist’s visibility.

**ACCEPTS:** [True | False]

`matplotlib.axis.Tick.set_zorder`

`Tick.set_zorder(level)`
Set the zorder for the artist. Artists with lower zorder values are drawn first.

**ACCEPTS:** any number

`matplotlib.axis.Tick.stale`

`Tick.stale`
If the artist is ‘stale’ and needs to be re-drawn for the output to match the internal state of the artist.

`matplotlib.axis.Tick.update`

`Tick.update(props)`
Update the properties of this Artist from the dictionary prop.

`matplotlib.axis.Tick.update_from`

`Tick.update_from(other)`
Copy properties from other to self.

`matplotlib.axis.Tick.zorder`

`Tick.zorder = 0`
**matplotlib.axis.XTick.add_callback**

**XTick.add_callback(func)**

Adds a callback function that will be called whenever one of the Artist’s properties changes.

Returns an id that is useful for removing the callback with `remove_callback()` later.

**matplotlib.axis.XTick.aname**

**XTick.aname = ‘Artist’**

**matplotlib.axis.XTick.axes**

**XTick.axes**

The Axes instance the artist resides in, or None.

**matplotlib.axis.XTick.contains**

**XTick.contains(mouseevent)**

Test whether the mouse event occurred in the Tick marks.

This function always returns false. It is more useful to test if the axis as a whole contains the mouse rather than the set of tick marks.

**matplotlib.axis.XTick.convert_xunits**

**XTick.convert_xunits(x)**

For artists in an axes, if the xaxis has units support, convert x using xaxis unit type

**matplotlib.axis.XTick.convert_yunits**

**XTick.convert_yunits(y)**

For artists in an axes, if the yaxis has units support, convert y using yaxis unit type

**matplotlib.axis.XTick.draw**

**XTick.draw(artist, renderer, *args, **kwargs)**
**matplotlib.axis.XTick.findobj**

`XTick.findobj(match=None, include_self=True)`

Find artist objects.

Recursively find all *Artist* instances contained in self.

`match` can be

- None: return all objects contained in artist.
- function with signature `boolean = match(artist)` used to filter matches
- class instance: e.g., Line2D. Only return artists of class type.

If `include_self` is True (default), include self in the list to be checked for a match.

**matplotlib.axis.XTick.format_cursor_data**

`XTick.format_cursor_data(data)`

Return *cursor data* string formatted.

**matplotlib.axis.XTick.get_agg_filter**

`XTick.get_agg_filter()`

Return filter function to be used for agg filter

**matplotlib.axis.XTick.get_alpha**

`XTick.get_alpha()`

Return the alpha value used for blending - not supported on all backends

**matplotlib.axis.XTick.get_animated**

`XTick.get_animated()`

Return the artist’s animated state

**matplotlib.axis.XTick.get_axes**

`XTick.get_axes()`

Return the *Axes* instance the artist resides in, or *None*.

This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.
Matplotlib, Release 2.0.2

`matplotlib.axis.XTick.get_children`

`XTick.get_children()`

`matplotlib.axis.XTick.get_clip_box`

`XTick.get_clip_box()`
  Return artist clipbox

`matplotlib.axis.XTick.get_clip_on`

`XTick.get_clip_on()`
  Return whether artist uses clipping

`matplotlib.axis.XTick.get_clip_path`

`XTick.get_clip_path()`
  Return artist clip path

`matplotlib.axis.XTick.get_contains`

`XTick.get_contains()`
  Return the _contains test used by the artist, or None for default.

`matplotlib.axis.XTick.get_cursor_data`

`XTick.get_cursor_data(event)`
  Get the cursor data for a given event.

`matplotlib.axis.XTick.get_figure`

`XTick.get_figure()`
  Return the Figure instance the artist belongs to.

`matplotlib.axis.XTick.get_gid`

`XTick.get_gid()`
  Returns the group id
**matplotlib.axis.XTick.get_label**

```
XTick.get_label()
```

Get the label used for this artist in the legend.

**matplotlib.axis.XTick.get_path_effects**

```
XTick.get_path_effects()
```

**matplotlib.axis.XTick.get_picker**

```
XTick.get_picker()
```

Return the picker object used by this artist

**matplotlib.axis.XTick.get_rasterized**

```
XTick.get_rasterized()
```

Return True if the artist is to be rasterized

**matplotlib.axis.XTick.get_sketch_params**

```
XTick.get_sketch_params()
```

Returns the sketch parameters for the artist.

Returns `sketch_params` : tuple or `None`

A 3-tuple with the following elements:

- `scale`: The amplitude of the wiggle perpendicular to the source line.
- `length`: The length of the wiggle along the line.
- `randomness`: The scale factor by which the length is shrunken or expanded.

May return `None` if no sketch parameters were set.

**matplotlib.axis.XTick.get_snap**

```
XTick.get_snap()
```

Returns the snap setting which may be:

- `True`: snap vertices to the nearest pixel center
- `False`: leave vertices as-is
- `None`: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

20.4. Common and inherited methods
Only supported by the Agg and MacOSX backends.

**matplotlib.axis.XTick.get_transform**

`XTick.get_transform()`
Return the `Transform` instance used by this artist.

**matplotlib.axis.XTick.get_transformed_clip_path_and_affine**

`XTick.get_transformed_clip_path_and_affine()`
Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

**matplotlib.axis.XTick.get_url**

`XTick.get_url()`
Returns the url

**matplotlib.axis.XTick.get_visible**

`XTick.get_visible()`
Return the artist’s visibility

**matplotlib.axis.XTick.get_window_extent**

`XTick.get_window_extent(renderer)`
Get the axes bounding box in display space. Subclasses should override for inclusion in the bounding box “tight” calculation. Default is to return an empty bounding box at 0, 0.

Be careful when using this function, the results will not update if the artist window extent of the artist changes. The extent can change due to any changes in the transform stack, such as changing the axes limits, the figure size, or the canvas used (as is done when saving a figure). This can lead to unexpected behavior where interactive figures will look fine on the screen, but will save incorrectly.

**matplotlib.axis.XTick.get_zorder**

`XTick.get_zorder()`
Return the Artist’s zorder.
**matplotlib.axis.XTick**

### have_units

**XTick**.have_units()

Return *True* if units are set on the *x* or *y* axes.

### hitlist

**XTick**.hitlist(*event*)

List the children of the artist which contain the mouse event *event*.

### is_figure_set

**XTick**.is_figure_set()

Returns *True* if the artist is assigned to a *Figure*.

### is_transform_set

**XTick**.is_transform_set()

Returns *True* if *Artist* has a transform explicitly set.

### mouseover

**XTick**.mouseover

### pchanged

**XTick**.pchanged()

Fire an event when property changed, calling all of the registered callbacks.

### pick

**XTick**.pick(*mouseevent*)

Process pick event

each child artist will fire a pick event if *mouseevent* is over the artist and the artist has picker set

### pickable

**XTick**.pickable()

Return *True* if *Artist* is pickable.

---

**20.4. Common and inherited methods** 855
matplotlib.axis.XTick.properties

XTick.properties()
    return a dictionary mapping property name -> value for all Artist props

matplotlib.axis.XTick.remove

XTick.remove()
    Remove the artist from the figure if possible. The effect will not be visible until the figure is redrawn, e.g., with matplotlib.axes.Axes.draw_idle(). Call matplotlib.axes.Axes.relim() to update the axes limits if desired.
    
    Note: relim() will not see collections even if the collection was added to axes with autolim = True.
    Note: there is no support for removing the artist’s legend entry.

matplotlib.axis.XTick.remove_callback

XTick.remove_callback(oid)
    Remove a callback based on its id.
    
    See also:

    add_callback() For adding callbacks

matplotlib.axis.XTick.set

XTick.set(**kwargs)
    A property batch setter. Pass kwargs to set properties.

matplotlib.axis.XTick.set_agg_filter

XTick.set_agg_filter(filter_func)
    set agg_filter function.

matplotlib.axis.XTick.set_alpha

XTick.set_alpha(alpha)
    Set the alpha value used for blending - not supported on all backends.
    
    ACCEPTS: float (0.0 transparent through 1.0 opaque)
**Matplotlib, Release 2.0.2**

**matplotlib.axis.XTick.set_animated**

`XTick.set_animated(b)`  
Set the artist’s animation state.  
ACCEPTS: [True | False]

**matplotlib.axis.XTick.set_axes**

`XTick.set_axes(axes)`  
Set the `Axes` instance in which the artist resides, if any.  
This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.  
ACCEPTS: an `Axes` instance

**matplotlib.axis.XTick.set_clip_box**

`XTick.set_clip_box(clipbox)`  
Set the artist’s clip `Bbox`.  
ACCEPTS: a `matplotlib.transforms.Bbox` instance

**matplotlib.axis.XTick.set_clip_on**

`XTick.set_clip_on(b)`  
Set whether artist uses clipping.  
When False artists will be visible out side of the axes which can lead to unexpected results.  
ACCEPTS: [True | False]

**matplotlib.axis.XTick.set_clip_path**

`XTick.set_clip_path(clippath, transform=None)`  
Set the artist’s clip path, which may be:  
- a `Patch` (or subclass) instance  
- a `Path` instance, in which case an optional `Transform` instance may be provided, which will be applied to the path before using it for clipping.  
- `None`, to remove the clipping path  
For efficiency, if the path happens to be an axis-aligned rectangle, this method will set the clipping box to the corresponding rectangle and set the clipping path to `None`.  
ACCEPTS: [ `(Path, Transform)` | `Patch` | `None` ]
 XTICK.set_contains(picker)

Replace the contains test used by this artist. The new picker should be a callable function which
determines whether the artist is hit by the mouse event:

```python
hit, props = picker(artist, mouseevent)
```

If the mouse event is over the artist, return hit = True and props is a dictionary of properties you want
returned with the contains test.

ACCEPTS: a callable function

XTICK.set_figure(fig)

Set the Figure instance the artist belongs to.

ACCEPTS: a `matplotlib.figure.Figure` instance

XTICK.set_gid(gid)

Sets the (group) id for the artist

ACCEPTS: an id string

XTICK.set_label(s)

Set the text of ticklabel

ACCEPTS: str

XTICK.set_path_effects(path_effects)

set path_effects, which should be a list of instances of matplotlib.pathffect._Base class or its deriva-
tives.

XTICK.set_picker(picker)

Set the epsilon for picking used by this artist

picker can be one of the following:
• *None*: picking is disabled for this artist (default)

• A boolean: if *True* then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist

• A float: if picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if it’s data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event

• A function: if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```python
hit, props = picker(artist, mouseevent)
```

To determine the hit test. If the mouse event is over the artist, return *hit=True* and props is a dictionary of properties you want added to the PickEvent attributes.

**ACCEPTS:** [None|float|boolean|callable]

**matplotlib.axis.XTick.set_rasterized**

```
XTick.set_rasterized(rasterized)
```

Force rasterized (bitmap) drawing in vector backend output.

Defaults to None, which implies the backend’s default behavior

**ACCEPTS:** [True | False | None]

**matplotlib.axis.XTick.set_sketch_params**

```
XTick.set_sketch_params(scale=None, length=None, randomness=None)
```

Sets the sketch parameters.

**Parameters**

- **scale**: float, optional
  
  The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is None, or not provided, no sketch filter will be provided.

- **length**: float, optional
  
  The length of the wiggle along the line, in pixels (default 128.0)

- **randomness**: float, optional
  
  The scale factor by which the length is shrunken or expanded (default 16.0)

**matplotlib.axis.XTick.set_snap**

```
XTick.set_snap(snap)
```

Sets the snap setting which may be:
• True: snap vertices to the nearest pixel center
• False: leave vertices as-is
• None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

**matplotlib.axis.XTick.set_transform**

`XTick.set_transform(t)`

Set the *Transform* instance used by this artist.

**ACCEPTS:** *Transform* instance

**matplotlib.axis.XTick.set_url**

`XTick.set_url(url)`

Sets the url for the artist

**ACCEPTS:** a url string

**matplotlib.axis.XTick.set_visible**

`XTick.set_visible(b)`

Set the artist’s visibility.

**ACCEPTS:** [True | False]

**matplotlib.axis.XTick.set_zorder**

`XTick.set_zorder(level)`

Set the zorder for the artist. Artists with lower zorder values are drawn first.

**ACCEPTS:** any number

**matplotlib.axis.XTick.stale**

`XTick.stale`

If the artist is ‘stale’ and needs to be re-drawn for the output to match the internal state of the artist.

**matplotlib.axis.XTick.update**

`XTick.update(props)`

Update the properties of this *Artist* from the dictionary *prop*.
`matplotlib.axis.XTick.update_from`

```python
XTick.update_from(other)
    Copy properties from other to self.
```

`matplotlib.axis.XTick.zorder`

```python
XTick.zorder = 0
```

`matplotlib.axis.YTick.add_callback`

```python
YTick.add_callback(func)
    Adds a callback function that will be called whenever one of the Artist's properties changes.
    Returns an id that is useful for removing the callback with remove_callback() later.
```

`matplotlib.axis.YTick.aname`

```python
YTick.aname = 'Artist'
```

`matplotlib.axis.YTick.axes`

```python
YTick.axes
    The Axes instance the artist resides in, or None.
```

`matplotlib.axis.YTick.contains`

```python
YTick.contains(mouseevent)
    Test whether the mouse event occurred in the Tick marks.
    This function always returns false. It is more useful to test if the axis as a whole contains the mouse
    rather than the set of tick marks.
```

`matplotlib.axis.YTick.convert_xunits`

```python
YTick.convert_xunits(x)
    For artists in an axes, if the xaxis has units support, convert x using xaxis unit type
```
**matplotlib.axis.YTick.convert_yunits**

`YTick.convert_yunits(y)`

For artists in an axes, if the y-axis has units support, convert `y` using y-axis unit type

**matplotlib.axis.YTick.draw**

`YTick.draw(artist, renderer, *args, **kwargs)`

**matplotlib.axis.YTick.findobj**

`YTick.findobj(match=None, include_self=True)`

Find artist objects.

Recursively find all `Artist` instances contained in self.

`match` can be

- None: return all objects contained in artist.
- function with signature `boolean = match(artist)` used to filter matches
- class instance: e.g., Line2D. Only return artists of class type.

If `include_self` is True (default), include self in the list to be checked for a match.

**matplotlib.axis.YTick.format_cursor_data**

`YTick.format_cursor_data(data)`

Return `cursor data` string formatted.

**matplotlib.axis.YTick.get_agg_filter**

`YTick.get_agg_filter()`

return filter function to be used for agg filter

**matplotlib.axis.YTick.get_alpha**

`YTick.get_alpha()`

Return the alpha value used for blending - not supported on all backends
**matplotlib.axis.YTick.get_animated**

YTick.get_animated()

Return the artist’s animated state

**matplotlib.axis.YTick.get_axes**

YTick.get_axes()

Return the Axes instance the artist resides in, or `None`.

This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

**matplotlib.axis.YTick.get_children**

YTick.get_children()

**matplotlib.axis.YTick.get_clip_box**

YTick.get_clip_box()

Return artist clipbox

**matplotlib.axis.YTick.get_clip_on**

YTick.get_clip_on()

Return whether artist uses clipping

**matplotlib.axis.YTick.get_clip_path**

YTick.get_clip_path()

Return artist clip path

**matplotlib.axis.YTick.get_contains**

YTick.get_contains()

Return the _contains test used by the artist, or `None` for default.

**matplotlib.axis.YTick.get_cursor_data**

YTick.get_cursor_data(event)

Get the cursor data for a given event.
**matplotlib.axis.YTick.get_figure**

```
YTick.get_figure()
```

Return the `Figure` instance the artist belongs to.

**matplotlib.axis.YTick.get_gid**

```
YTick.get_gid()
```

Returns the group id

**matplotlib.axis.YTick.get_label**

```
YTick.get_label()
```

Get the label used for this artist in the legend.

**matplotlib.axis.YTick.get_path_effects**

```
YTick.get_path_effects()
```

**matplotlib.axis.YTick.get_picker**

```
YTick.get_picker()
```

Return the picker object used by this artist

**matplotlib.axis.YTick.get_rasterized**

```
YTick.get_rasterized()
```

return True if the artist is to be rasterized

**matplotlib.axis.YTick.get_sketch_params**

```
YTick.get_sketch_params()
```

Returns the sketch parameters for the artist.

Returns `sketch_params` : tuple or `None`

A 3-tuple with the following elements:

- `scale` : The amplitude of the wiggle perpendicular to the source line.
- `length` : The length of the wiggle along the line.
- `randomness` : The scale factor by which the length is shrunken or expanded.
Matplotlib, Release 2.0.2

May return `None` if no sketch parameters were set.

```
matplotlib.axis.YTick.get_snap
```

`YTick.get_snap()`

Returns the snap setting which may be:

- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

```
matplotlib.axis.YTick.get_transform
```

`YTick.get_transform()`

Return the `Transform` instance used by this artist.

```
matplotlib.axis.YTick.get_transformed_clip_path_and_affine
```

`YTick.get_transformed_clip_path_and_affine()`

Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

```
matplotlib.axis.YTick.get_url
```

`YTick.get_url()`

Returns the url

```
matplotlib.axis.YTick.get_visible
```

`YTick.get_visible()`

Return the artist’s visibility

```
matplotlib.axis.YTick.get_window_extent
```

`YTick.get_window_extent(renderer)`

Get the axes bounding box in display space. Subclasses should override for inclusion in the bounding box “tight” calculation. Default is to return an empty bounding box at 0, 0.

Be careful when using this function, the results will not update if the artist window extent of the artist changes. The extent can change due to any changes in the transform stack, such as changing the

20.4. Common and inherited methods
axes limits, the figure size, or the canvas used (as is done when saving a figure). This can lead to unexpected behavior where interactive figures will look fine on the screen, but will save incorrectly.

**matplotlib.axis.YTick.get_zorder**

`YTick.get_zorder()`  
Return the Artist's zorder.

**matplotlib.axis.YTick.have_units**

`YTick.have_units()`  
Return True if units are set on the x or y axes

**matplotlib.axis.YTick.hitlist**

`YTick.hitlist(event)`  
List the children of the artist which contain the mouse event event.

**matplotlib.axis.YTick.is_figure_set**

`YTick.is_figure_set()`  
Returns True if the artist is assigned to a Figure.

**matplotlib.axis.YTick.is_transform_set**

`YTick.is_transform_set()`  
Returns True if Artist has a transform explicitly set.

**matplotlib.axis.YTick.mouseover**

`YTick.mouseover`

**matplotlib.axis.YTick.pchanged**

`YTick.pchanged()`  
Fire an event when property changed, calling all of the registered callbacks.
**matplotlib.axis.YTick.pick**

`YTICK.pick(mouseevent)`  
Process pick event  
each child artist will fire a pick event if `mouseevent` is over the artist and the artist has picker set

**matplotlib.axis.YTick.pickable**

`YTICK.pickable()`  
Return `True` if `Artist` is pickable.

**matplotlib.axis.YTick.properties**

`YTICK.properties()`  
return a dictionary mapping property name -> value for all Artist props

**matplotlib.axis.YTick.remove**

`YTICK.remove()`  
Remove the artist from the figure if possible. The effect will not be visible until the figure is redrawn, e.g., with `matplotlib.axes.Axes.draw_idle()`. Call `matplotlib.axes.Axes.relim()` to update the axes limits if desired.  
Note: `relim()` will not see collections even if the collection was added to axes with `autolim = True`.  
Note: there is no support for removing the artist’s legend entry.

**matplotlib.axis.YTick.remove_callback**

`YTICK.remove_callback(id)`  
Remove a callback based on its `id`.  
See also:  

`add_callback()` For adding callbacks

**matplotlib.axis.YTick.set**

`YTICK.set(**kwargs)`  
A property batch setter. Pass `kwargs` to set properties.
**matplotlib.axis.YTick.set_agg_filter**

YTick.set_agg_filter(filter_func)

Set the agg_filter function.

**matplotlib.axis.YTick.set_alpha**

YTick.set_alpha(alpha)

Set the alpha value used for blending - not supported on all backends.

ACCEPETS: float (0.0 transparent through 1.0 opaque)

**matplotlib.axis.YTick.set_animated**

YTick.set_animated(b)

Set the artist’s animation state.

ACCEPETS: [True | False]

**matplotlib.axis.YTick.set_axes**

YTick.set_axes(axes)

Set the Axes instance in which the artist resides, if any.

This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

ACCEPETS: an Axes instance

**matplotlib.axis.YTick.set_clip_box**

YTick.set_clip_box(clipbox)

Set the artist’s clip Bbox.

ACCEPETS: a matplotlib.transforms.Bbox instance

**matplotlib.axis.YTick.set_clip_on**

YTick.set_clip_on(b)

Set whether artist uses clipping.

When False artists will be visible outside of the axes which can lead to unexpected results.

ACCEPETS: [True | False]
**matplotlib.axis.YTick.set_clip_path**

**YTick.set_clip_path(clippath, transform=None)**

Set the artist’s clip path, which may be:

- a `Patch` (or subclass) instance
- a `Path` instance, in which case an optional `Transform` instance may be provided, which will be applied to the path before using it for clipping.
- `None`, to remove the clipping path

For efficiency, if the path happens to be an axis-aligned rectangle, this method will set the clipping box to the corresponding rectangle and set the clipping path to `None`.

**ACCEPTS:** [(Path, Transform)| Patch | None ]

**matplotlib.axis.YTick.set_contains**

**YTick.set_contains(picker)**

Replace the contains test used by this artist. The new picker should be a callable function which determines whether the artist is hit by the mouse event:

```python
hit, props = picker(artist, mouseevent)
```

If the mouse event is over the artist, return `hit = True` and `props` is a dictionary of properties you want returned with the contains test.

**ACCEPTS:** a callable function

**matplotlib.axis.YTick.set_figure**

**YTick.set_figure(fig)**

Set the `Figure` instance the artist belongs to.

**ACCEPTS:** a `matplotlib.figure.Figure` instance

**matplotlib.axis.YTick.set_gid**

**YTick.set_gid(gid)**

Sets the (group) id for the artist

**ACCEPTS:** an id string

**matplotlib.axis.YTick.set_label**

**YTick.set_label(s)**

Set the text of ticklabel
ACCEPPTS: str

**matplotlib.axis.YTick.set_path_effects**

**YTick.set_path_effects**(path effects)

set path_effects, which should be a list of instances of matplotlib.patheffect._Base class or its derivatives.

**matplotlib.axis.YTick.set_picker**

**YTick.set_picker**(picker)

Set the epsilon for picking used by this artist

*picker* can be one of the following:

- *None*: picking is disabled for this artist (default)
- A boolean: if True then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist
- A float: if picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if it’s data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event
- A function: if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

  ```python
  hit, props = picker(artist, mouseevent)
  ```

  to determine the hit test. if the mouse event is over the artist, return hit=True and props is a dictionary of properties you want added to the PickEvent attributes.

ACCEPPTS: [None|float|boolean|callable]

**matplotlib.axis.YTick.set_rasterized**

**YTick.set_rasterized**(rasterized)

Force rasterized (bitmap) drawing in vector backend output.

Defaults to None, which implies the backend’s default behavior

ACCEPPTS: [True | False | None]

**matplotlib.axis.YTick.set_sketch_params**

**YTick.set_sketch_params**(scale=None, length=None, randomness=None)

Sets the sketch parameters.
Parameters scale : float, optional

The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is None, or not provided, no sketch filter will be provided.

length : float, optional

The length of the wiggle along the line, in pixels (default 128.0)

randomness : float, optional

The scale factor by which the length is shrunken or expanded (default 16.0)

**matplotlib.axis.YTick.set_snap**

YTick.set_snap(snapshot)

Sets the snap setting which may be:

• True: snap vertices to the nearest pixel center
• False: leave vertices as-is
• None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

**matplotlib.axis.YTick.set_transform**

YTick.set_transform(transform)

Set the Transform instance used by this artist.

ACCEPTS: Transform instance

**matplotlib.axis.YTick.set_url**

YTick.set_url(url)

Sets the url for the artist

ACCEPTS: a url string

**matplotlib.axis.YTick.set_visible**

YTick.set_visible(flag)

Set the artist’s visibility.

ACCEPTS: [True | False]
matplotlib.axis.YTick.set_zorder

```
YTick.set_zorder(level)
    Set the zorder for the artist. Artists with lower zorder values are drawn first.
    ACCEPTS: any number
```

matplotlib.axis.YTick.stale

```
YTick.stale
    If the artist is ‘stale’ and needs to be re-drawn for the output to match the internal state of the artist.
```

matplotlib.axis.YTick.update

```
YTick.update(props)
    Update the properties of this Artist from the dictionary prop.
```

matplotlib.axis.YTick.update_from

```
YTick.update_from(other)
    Copy properties from other to self.
```

matplotlib.axis.YTick.zorder

```
YTick.zorder = 0
```

**Axis**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Axis.add_callback</td>
<td>Adds a callback function that will be called whenever one of the Artist’s properties changes.</td>
</tr>
<tr>
<td>Axis.aname</td>
<td></td>
</tr>
<tr>
<td>Axis.axes</td>
<td>The Axes instance the artist resides in, or None.</td>
</tr>
<tr>
<td>Axis.contains</td>
<td>Test whether the artist contains the mouse event.</td>
</tr>
<tr>
<td>Axis.convert_xunits</td>
<td>For artists in an axes, if the xaxis has units support,</td>
</tr>
<tr>
<td>Axis.convert_yunits</td>
<td>For artists in an axes, if the yaxis has units support,</td>
</tr>
<tr>
<td>Axis.draw</td>
<td>Draw the axis lines, grid lines, tick lines and labels</td>
</tr>
<tr>
<td>Axis.findobj</td>
<td>Find artist objects.</td>
</tr>
<tr>
<td>Axis.format_cursor_data</td>
<td>Return cursor data string formatted.</td>
</tr>
<tr>
<td>Axis.get_agg_filter</td>
<td>return filter function to be used for agg filter</td>
</tr>
<tr>
<td>Axis.get_alpha</td>
<td>Return the alpha value used for blending - not supported on all</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Axis.get_animated</code></td>
<td>Return the artist’s animated state</td>
</tr>
<tr>
<td><code>Axis.get_axes</code></td>
<td>Return the Axes instance the artist resides in, or None.</td>
</tr>
<tr>
<td><code>Axis.get_children</code></td>
<td></td>
</tr>
<tr>
<td><code>Axis.get_clip_box</code></td>
<td>Return artist clipbox</td>
</tr>
<tr>
<td><code>Axis.get_clip_on</code></td>
<td>Return whether artist uses clipping</td>
</tr>
<tr>
<td><code>Axis.get_clip_path</code></td>
<td>Return artist clip path</td>
</tr>
<tr>
<td><code>Axis.get_contains</code></td>
<td>Return the _contains test used by the artist, or None for default.</td>
</tr>
<tr>
<td><code>Axis.get_cursor_data</code></td>
<td>Get the cursor data for a given event.</td>
</tr>
<tr>
<td><code>Axis.get_figure</code></td>
<td>Return the Figure instance the artist belongs to.</td>
</tr>
<tr>
<td><code>Axis.get_gid</code></td>
<td>Return the group id</td>
</tr>
<tr>
<td><code>Axis.get_label</code></td>
<td>Return the axis label as a Text instance</td>
</tr>
<tr>
<td><code>Axis.get_path_effects</code></td>
<td></td>
</tr>
<tr>
<td><code>Axis.get_picker</code></td>
<td>Return the picker object used by this artist</td>
</tr>
<tr>
<td><code>Axis.get_rasterized</code></td>
<td>return True if the artist is to be rasterized</td>
</tr>
<tr>
<td><code>Axis.get_sketch_params</code></td>
<td>Returns the sketch parameters for the artist.</td>
</tr>
<tr>
<td><code>Axis.get_snap</code></td>
<td>Returns the snap setting which may be:</td>
</tr>
<tr>
<td><code>Axis.get_transform</code></td>
<td></td>
</tr>
<tr>
<td><code>Axis.get_transformed_clip_path_and_affine</code></td>
<td>Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.</td>
</tr>
<tr>
<td><code>Axis.get_url</code></td>
<td>Returns the url</td>
</tr>
<tr>
<td><code>Axis.get_visible</code></td>
<td>Return the artist’s visibility</td>
</tr>
<tr>
<td><code>Axis.get_window_extent</code></td>
<td>Get the axes bounding box in display space.</td>
</tr>
<tr>
<td><code>Axis.get_zorder</code></td>
<td>Return the Artist’s zorder</td>
</tr>
<tr>
<td><code>Axis.have_units</code></td>
<td></td>
</tr>
<tr>
<td><code>Axis.hitlist</code></td>
<td>List the children of the artist which contain the mouse event event.</td>
</tr>
<tr>
<td><code>Axis.is_figure_set</code></td>
<td>Returns True if the artist is assigned to a Figure.</td>
</tr>
<tr>
<td><code>Axis.is_transform_set</code></td>
<td>Returns True if Artist has a transform explicitly set.</td>
</tr>
<tr>
<td><code>Axis.mouseover</code></td>
<td></td>
</tr>
<tr>
<td><code>Axis.pchanged</code></td>
<td>Fire an event when property changed, calling all of the registered callbacks.</td>
</tr>
<tr>
<td><code>Axis.pick</code></td>
<td>Process pick event</td>
</tr>
<tr>
<td><code>Axis.pickable</code></td>
<td>Return True if Artist is pickable.</td>
</tr>
<tr>
<td><code>Axis.properties</code></td>
<td>return a dictionary mapping property name -&gt; value for all Artist props</td>
</tr>
<tr>
<td><code>Axis.remove</code></td>
<td>Remove the artist from the figure if possible.</td>
</tr>
<tr>
<td><code>Axis.remove_callback</code></td>
<td>Remove a callback based on its id.</td>
</tr>
<tr>
<td><code>Axis.set</code></td>
<td>A property batch setter.</td>
</tr>
<tr>
<td><code>Axis.set_agg_filter</code></td>
<td>set agg_filter fuction.</td>
</tr>
<tr>
<td><code>Axis.set_alpha</code></td>
<td>Set the alpha value used for blending - not supported on all backends.</td>
</tr>
<tr>
<td><code>Axis.set_animated</code></td>
<td>Set the artist’s animation state.</td>
</tr>
</tbody>
</table>

Continued on next page
Table 20.20 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Axes.set_axes</code></td>
<td>Set the <em>Axes</em> instance in which the artist resides, if any.</td>
</tr>
<tr>
<td><code>Axes.set_clip_box</code></td>
<td>Set the artist’s clip <em>Bbox</em>.</td>
</tr>
<tr>
<td><code>Axes.set_clip_on</code></td>
<td>Set whether artist uses clipping.</td>
</tr>
<tr>
<td><code>Axes.set_clip_path</code></td>
<td></td>
</tr>
<tr>
<td><code>Axes.set_contains</code></td>
<td>Replace the contains test used by this artist.</td>
</tr>
<tr>
<td><code>Axes.set_figure</code></td>
<td>Set the <em>Figure</em> instance the artist belongs to.</td>
</tr>
<tr>
<td><code>Axes.set_gid</code></td>
<td>Sets the (group) id for the artist</td>
</tr>
<tr>
<td><code>Axes.set_label</code></td>
<td>Set the label to <code>s</code> for auto legend.</td>
</tr>
<tr>
<td><code>Axes.set_path_effects</code></td>
<td>Set path effects, which should be a list of instances of <code>path_effects</code>.</td>
</tr>
<tr>
<td><code>Axes.set_picker</code></td>
<td>Set the epsilon for picking used by this artist.</td>
</tr>
<tr>
<td><code>Axes.set_rasterized</code></td>
<td>Force rasterized (bitmap) drawing in vector backend output.</td>
</tr>
<tr>
<td><code>Axes.set_sketch_params</code></td>
<td>Sets the sketch parameters.</td>
</tr>
<tr>
<td><code>Axes.set_snap</code></td>
<td>Sets the snap setting which may be:</td>
</tr>
<tr>
<td><code>Axes.set_transform</code></td>
<td>Set the <em>Transform</em> instance used by this artist.</td>
</tr>
<tr>
<td><code>Axes.set_url</code></td>
<td>Sets the url for the artist</td>
</tr>
<tr>
<td><code>Axes.set_visible</code></td>
<td>Set the artist’s visibility</td>
</tr>
<tr>
<td><code>Axes.set_zorder</code></td>
<td>Set the zorder for the artist</td>
</tr>
<tr>
<td><code>Axes.stale</code></td>
<td>If the artist is ‘stale’ and needs to be re-drawn for the output to match the internal state of the artist.</td>
</tr>
<tr>
<td><code>Axes.update</code></td>
<td>Update the properties of this <em>Artist</em> from the dictionary <code>prop</code>.</td>
</tr>
<tr>
<td><code>Axes.update_from</code></td>
<td>Copy properties from <code>other</code> to <code>self</code>.</td>
</tr>
<tr>
<td><code>Axes.zorder</code></td>
<td></td>
</tr>
<tr>
<td><code>XAxis.add_callback</code></td>
<td>Adds a callback function that will be called whenever one of the <em>Artist</em>’s properties changes.</td>
</tr>
<tr>
<td><code>XAxis.aname</code></td>
<td></td>
</tr>
<tr>
<td><code>XAxis.axes</code></td>
<td>The <em>Axes</em> instance the artist resides in, or <em>None</em>.</td>
</tr>
<tr>
<td><code>XAxis.contains</code></td>
<td>Test whether the mouse event occurred in the x axis.</td>
</tr>
<tr>
<td><code>XAxis.convert_xunits</code></td>
<td>For artists in an axes, if the xaxis has units support.</td>
</tr>
<tr>
<td><code>XAxis.convert_yunits</code></td>
<td>For artists in an axes, if the yaxis has units support.</td>
</tr>
<tr>
<td><code>XAxis.draw</code></td>
<td>Draw the axis lines, grid lines, tick lines and labels</td>
</tr>
<tr>
<td><code>XAxis.findobj</code></td>
<td>Find artist objects.</td>
</tr>
<tr>
<td><code>XAxis.format_cursor_data</code></td>
<td>Return <em>cursor data</em> string formatted.</td>
</tr>
<tr>
<td><code>XAxis.get_agg_filter</code></td>
<td>return filter function to be used for agg filter</td>
</tr>
<tr>
<td><code>XAxis.get_alpha</code></td>
<td>Return the alpha value used for blending - not supported on all</td>
</tr>
<tr>
<td><code>XAxis.get_animated</code></td>
<td>Return the artist’s animated state</td>
</tr>
<tr>
<td><code>XAxis.get_axes</code></td>
<td>Return the <em>Axes</em> instance the artist resides in, or <em>None</em>.</td>
</tr>
<tr>
<td><code>XAxis.get_children</code></td>
<td></td>
</tr>
<tr>
<td><code>XAxis.get_clip_box</code></td>
<td>Return artist clipbox</td>
</tr>
<tr>
<td><code>XAxis.get_clip_on</code></td>
<td>Return whether artist uses clipping</td>
</tr>
<tr>
<td><code>XAxis.get_clip_path</code></td>
<td>Return artist clip path</td>
</tr>
</tbody>
</table>
### Table 20.20 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>XAxis.get_contains</code></td>
<td>Return the _contains test used by the artist, or None for default.</td>
</tr>
<tr>
<td><code>XAxis.get_cursor_data</code></td>
<td>Get the cursor data for a given event.</td>
</tr>
<tr>
<td><code>XAxis.get_figure</code></td>
<td>Return the Figure instance the artist belongs to.</td>
</tr>
<tr>
<td><code>XAxis.get_gid</code></td>
<td>Returns the group id</td>
</tr>
<tr>
<td><code>XAxis.get_label</code></td>
<td>Return the axis label as a Text instance</td>
</tr>
<tr>
<td><code>XAxis.get_path_effects</code></td>
<td></td>
</tr>
<tr>
<td><code>XAxis.get_picker</code></td>
<td>Return the picker object used by this artist</td>
</tr>
<tr>
<td><code>XAxis.get_rasterized</code></td>
<td>return True if the artist is to be rasterized</td>
</tr>
<tr>
<td><code>XAxis.get_sketch_params</code></td>
<td>Returns the sketch parameters for the artist.</td>
</tr>
<tr>
<td><code>XAxis.get_snap</code></td>
<td>Returns the snap setting which may be:</td>
</tr>
<tr>
<td><code>XAxis.get_transform</code></td>
<td></td>
</tr>
<tr>
<td><code>XAxis.get_transformed_clip_path_and_affine</code></td>
<td>Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.</td>
</tr>
<tr>
<td><code>XAxis.get_url</code></td>
<td>Returns the url</td>
</tr>
<tr>
<td><code>XAxis.get_visible</code></td>
<td>Return the artist’s visibility</td>
</tr>
<tr>
<td><code>XAxis.get_window_extent</code></td>
<td>Get the axes bounding box in display space.</td>
</tr>
<tr>
<td><code>XAxis.get_zorder</code></td>
<td>Return the Artist’s zorder</td>
</tr>
<tr>
<td><code>XAxis.have_units</code></td>
<td></td>
</tr>
<tr>
<td><code>XAxis.hitlist</code></td>
<td>List the children of the artist which contain the mouse event event.</td>
</tr>
<tr>
<td><code>XAxis.is_figure_set</code></td>
<td>Returns True if the artist is assigned to a Figure.</td>
</tr>
<tr>
<td><code>XAxis.is_transform_set</code></td>
<td>Returns True if Artist has a transform explicitly set.</td>
</tr>
<tr>
<td><code>XAxis.mouseover</code></td>
<td>Fire an event when property changed, calling all of the registered callbacks.</td>
</tr>
<tr>
<td><code>XAxis.pchanged</code></td>
<td></td>
</tr>
<tr>
<td><code>XAxis.pick</code></td>
<td>Process pick event</td>
</tr>
<tr>
<td><code>XAxis.pickable</code></td>
<td>Return True if Artist is pickable.</td>
</tr>
<tr>
<td><code>XAxis.properties</code></td>
<td>return a dictionary mapping property name -&gt; value for all Artist props</td>
</tr>
<tr>
<td><code>XAxis.remove</code></td>
<td>Remove the artist from the figure if possible.</td>
</tr>
<tr>
<td><code>XAxis.remove_callback</code></td>
<td>Remove a callback based on its id.</td>
</tr>
<tr>
<td><code>XAxis.set</code></td>
<td>A property batch setter.</td>
</tr>
<tr>
<td><code>XAxis.set_agg_filter</code></td>
<td>set agg_filter fuction.</td>
</tr>
<tr>
<td><code>XAxis.set_alpha</code></td>
<td>Set the alpha value used for blending - not supported on all backends.</td>
</tr>
<tr>
<td><code>XAxis.set_animated</code></td>
<td>Set the artist’s animation state.</td>
</tr>
<tr>
<td><code>XAxis.set_axes</code></td>
<td>Set the Axes instance in which the artist resides, if any.</td>
</tr>
<tr>
<td><code>XAxis.set_clip_box</code></td>
<td>Set the artist’s clip Bbox.</td>
</tr>
<tr>
<td><code>XAxis.set_clip_on</code></td>
<td>Set whether artist uses clipping.</td>
</tr>
<tr>
<td><code>XAxis.set_clip_path</code></td>
<td></td>
</tr>
<tr>
<td><code>XAxis.set_contains</code></td>
<td>Replace the contains test used by this artist.</td>
</tr>
</tbody>
</table>

Continued on next page
Table 20.20 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>XAxis.set_figure</code></td>
<td>Set the <code>Figure</code> instance the artist belongs to.</td>
</tr>
<tr>
<td><code>XAxis.set_gid</code></td>
<td>Sets the (group) id for the artist.</td>
</tr>
<tr>
<td><code>XAxis.set_label</code></td>
<td>Set the label to <code>s</code> for auto legend.</td>
</tr>
<tr>
<td><code>XAxis.set_path_effects</code></td>
<td>Set path effects, which should be a list of instances of</td>
</tr>
<tr>
<td><code>XAxis.set_picker</code></td>
<td>Set the epsilon for picking used by this artist.</td>
</tr>
<tr>
<td><code>XAxis.set_rasterized</code></td>
<td>Force rasterized (bitmap) drawing in vector backend output.</td>
</tr>
<tr>
<td><code>XAxis.set_sketch_params</code></td>
<td>Sets the sketch parameters.</td>
</tr>
<tr>
<td><code>XAxis.set_snap</code></td>
<td>Sets the snap setting which may be:</td>
</tr>
<tr>
<td><code>XAxis.set_transform</code></td>
<td>Set the <code>Transform</code> instance used by this artist.</td>
</tr>
<tr>
<td><code>XAxis.set_url</code></td>
<td>Sets the url for the artist.</td>
</tr>
<tr>
<td><code>XAxis.set_visible</code></td>
<td>Set the artist’s visibility.</td>
</tr>
<tr>
<td><code>XAxis.set_zorder</code></td>
<td>Set the zorder for the artist.</td>
</tr>
<tr>
<td><code>XAxis.stale</code></td>
<td>If the artist is ‘stale’ and needs to be re-drawn for the output to match the internal state of the artist.</td>
</tr>
<tr>
<td><code>XAxis.update</code></td>
<td>Update the properties of this <code>Artist</code> from the dictionary <code>prop</code>.</td>
</tr>
<tr>
<td><code>XAxis.update_from</code></td>
<td>Copy properties from <code>other</code> to <code>self</code>.</td>
</tr>
<tr>
<td><code>XAxis.zorder</code></td>
<td></td>
</tr>
<tr>
<td><code>YAxis.add_callback</code></td>
<td>Adds a callback function that will be called whenever one of the <code>Artist</code>’s properties changes.</td>
</tr>
<tr>
<td><code>YAxis.aname</code></td>
<td></td>
</tr>
<tr>
<td><code>YAxis.axes</code></td>
<td>The <code>Axes</code> instance the artist resides in, or <code>None</code>.</td>
</tr>
<tr>
<td><code>YAxis.contains</code></td>
<td>Test whether the mouse event occurred in the y axis.</td>
</tr>
<tr>
<td><code>YAxis.convert_xunits</code></td>
<td>For artists in an axes, if the xaxis has units support,</td>
</tr>
<tr>
<td><code>YAxis.convert_yunits</code></td>
<td>For artists in an axes, if the yaxis has units support,</td>
</tr>
<tr>
<td><code>YAxis.draw</code></td>
<td>Draw the axis lines, grid lines, tick lines and labels.</td>
</tr>
<tr>
<td><code>YAxis.findobj</code></td>
<td>Find artist objects.</td>
</tr>
<tr>
<td><code>YAxis.format_cursor_data</code></td>
<td>Return <code>cursor data</code> string formatted.</td>
</tr>
<tr>
<td><code>YAxis.get_agg_filter</code></td>
<td>Return filter function to be used for agg filter.</td>
</tr>
<tr>
<td><code>YAxis.get_alpha</code></td>
<td>Return the alpha value used for blending - not supported on all.</td>
</tr>
<tr>
<td><code>YAxis.get_animated</code></td>
<td>Return the artist’s animated state.</td>
</tr>
<tr>
<td><code>YAxis.get_axes</code></td>
<td>Return the <code>Axes</code> instance the artist resides in, or <code>None</code>.</td>
</tr>
<tr>
<td><code>YAxis.get_children</code></td>
<td>Return artist clipbox.</td>
</tr>
<tr>
<td><code>YAxis.get_clip_box</code></td>
<td>Return whether artist uses clipping.</td>
</tr>
<tr>
<td><code>YAxis.get_clip_on</code></td>
<td>Return artist clip path.</td>
</tr>
<tr>
<td><code>YAxis.get_clip_path</code></td>
<td>Return the <code>_contains</code> test used by the artist, or <code>None</code> for default.</td>
</tr>
<tr>
<td><code>YAxis.get_contains</code></td>
<td>Get the cursor data for a given event.</td>
</tr>
<tr>
<td><code>YAxis.get_cursor_data</code></td>
<td>Return the <code>Figure</code> instance the artist belongs to.</td>
</tr>
<tr>
<td><code>YAxis.get_figure</code></td>
<td>Returns the group id.</td>
</tr>
<tr>
<td><code>YAxis.get_gid</code></td>
<td>Return the axis label as a <code>Text</code> instance.</td>
</tr>
<tr>
<td><code>YAxis.get_label</code></td>
<td></td>
</tr>
</tbody>
</table>

Continued on next page
### Table 20.20 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>YAxis.get_path_effects</code></td>
<td>Return the path effects used by this artist.</td>
</tr>
<tr>
<td><code>YAxis.get_picker</code></td>
<td>Return the picker object used by this artist.</td>
</tr>
<tr>
<td><code>YAxis.get_rasterized</code></td>
<td>Return True if the artist is to be rasterized.</td>
</tr>
<tr>
<td><code>YAxis.get_sketch_params</code></td>
<td>Returns the sketch parameters for the artist.</td>
</tr>
<tr>
<td><code>YAxis.get_snap</code></td>
<td>Return the snap setting which may be:</td>
</tr>
<tr>
<td><code>YAxis.get_transform</code></td>
<td>Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.</td>
</tr>
<tr>
<td><code>YAxis.get_transformed_clip_path_and_affine</code></td>
<td>Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.</td>
</tr>
<tr>
<td><code>YAxis.get_url</code></td>
<td>Return the url.</td>
</tr>
<tr>
<td><code>YAxis.get_visible</code></td>
<td>Return the artist’s visibility.</td>
</tr>
<tr>
<td><code>YAxis.get_window_extent</code></td>
<td>Get the axes bounding box in display space.</td>
</tr>
<tr>
<td><code>YAxis.get_zorder</code></td>
<td>Return the artist’s zorder.</td>
</tr>
<tr>
<td><code>YAxis.have_units</code></td>
<td></td>
</tr>
<tr>
<td><code>YAxis.hitlist</code></td>
<td>List the children of the artist which contain the mouse event.</td>
</tr>
<tr>
<td><code>YAxis.is_figure_set</code></td>
<td>Returns True if the artist is assigned to a <code>Figure</code>.</td>
</tr>
<tr>
<td><code>YAxis.is_transform_set</code></td>
<td>Returns <code>True</code> if <code>Artist</code> has a transform explicitly set.</td>
</tr>
<tr>
<td><code>YAxis.mouseover</code></td>
<td></td>
</tr>
<tr>
<td><code>YAxis.pchanged</code></td>
<td>Fire an event when property changed, calling all of the registered callbacks.</td>
</tr>
<tr>
<td><code>YAxis.pick</code></td>
<td>Process pick event.</td>
</tr>
<tr>
<td><code>YAxis.pickable</code></td>
<td>Return <code>True</code> if <code>Artist</code> is pickable.</td>
</tr>
<tr>
<td><code>YAxis.properties</code></td>
<td>Return a dictionary mapping property name -&gt; value for all <code>Artist</code> props.</td>
</tr>
<tr>
<td><code>YAxis.remove</code></td>
<td>Remove the artist from the figure if possible.</td>
</tr>
<tr>
<td><code>YAxis.remove_callback</code></td>
<td>Remove a callback based on its <code>id</code>.</td>
</tr>
<tr>
<td><code>YAxis.set</code></td>
<td>A property batch setter.</td>
</tr>
<tr>
<td><code>YAxis.set_agg_filter</code></td>
<td>Set <code>agg_filter</code> function.</td>
</tr>
<tr>
<td><code>YAxis.set_alpha</code></td>
<td>Set the alpha value used for blending - not supported on all backends.</td>
</tr>
<tr>
<td><code>YAxis.set_animated</code></td>
<td>Set the artist’s animation state.</td>
</tr>
<tr>
<td><code>YAxis.set_axes</code></td>
<td>Set the <code>Axes</code> instance in which the artist resides, if any.</td>
</tr>
<tr>
<td><code>YAxis.set_clip_box</code></td>
<td>Set the artist’s clip <code>Bbox</code>.</td>
</tr>
<tr>
<td><code>YAxis.set_clip_on</code></td>
<td>Set whether artist uses clipping.</td>
</tr>
<tr>
<td><code>YAxis.set_clip_path</code></td>
<td></td>
</tr>
<tr>
<td><code>YAxis.set_contains</code></td>
<td>Replace the contains test used by this artist.</td>
</tr>
<tr>
<td><code>YAxis.set_figure</code></td>
<td>Set the <code>Figure</code> instance the artist belongs to.</td>
</tr>
<tr>
<td><code>YAxis.set_gid</code></td>
<td>Sets the (group) id for the artist.</td>
</tr>
<tr>
<td><code>YAxis.set_label</code></td>
<td>Set the label to <code>s</code> for auto legend.</td>
</tr>
<tr>
<td><code>YAxis.set_path_effects</code></td>
<td>Set path effects, which should be a list of instances of <code>PathEffect</code>.</td>
</tr>
<tr>
<td><code>YAxis.set_picker</code></td>
<td>Set the epsilon for picking used by this artist.</td>
</tr>
</tbody>
</table>
### Table 20.20 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>YAxis.set_rasterized</code></td>
<td>Force rasterized (bitmap) drawing in vector backend output.</td>
</tr>
<tr>
<td><code>YAxis.set_sketch_params</code></td>
<td>Sets the sketch parameters.</td>
</tr>
<tr>
<td><code>YAxis.set_snap</code></td>
<td>Sets the snap setting which may be:</td>
</tr>
<tr>
<td><code>YAxis.set_transform</code></td>
<td>Set the <code>Transform</code> instance used by this artist.</td>
</tr>
<tr>
<td><code>YAxis.set_url</code></td>
<td>Sets the url for the artist.</td>
</tr>
<tr>
<td><code>YAxis.set_visible</code></td>
<td>Set the artist’s visibility.</td>
</tr>
<tr>
<td><code>YAxis.set_zorder</code></td>
<td>Set the zorder for the artist.</td>
</tr>
<tr>
<td><code>YAxis.stale</code></td>
<td>If the artist is ‘stale’ and needs to be re-drawn for the output to match the internal state of the artist.</td>
</tr>
<tr>
<td><code>YAxis.update</code></td>
<td>Update the properties of this <code>Artist</code> from the dictionary <code>prop</code>.</td>
</tr>
<tr>
<td><code>YAxis.update_from</code></td>
<td>Copy properties from <code>other</code> to <code>self</code>.</td>
</tr>
<tr>
<td><code>YAxis.zorder</code></td>
<td></td>
</tr>
</tbody>
</table>

**matplotlib.axis.Axis.add_callback**

`Axis.add_callback(func)`

Adds a callback function that will be called whenever one of the `Artist`’s properties changes.

Returns an `id` that is useful for removing the callback with `remove_callback()` later.

**matplotlib.axis.Axis.aname**

`Axis.aname = ‘Artist’`

**matplotlib.axis.Axis.axes**

`Axis.axes`

The `Axes` instance the artist resides in, or `None`.

**matplotlib.axis.Axis.contains**

`Axis.contains(mouseevent)`

Test whether the artist contains the mouse event.

Returns the truth value and a dictionary of artist specific details of selection, such as which points are contained in the pick radius. See individual artists for details.
**matplotlib.axis.Axis.convert_xunits**

Axis.**convert_xunits**(x)
For artists in an axes, if the xaxis has units support, convert x using xaxis unit type

**matplotlib.axis.Axis.convert_yunits**

Axis.**convert_yunits**(y)
For artists in an axes, if the yaxis has units support, convert y using yaxis unit type

**matplotlib.axis.Axis.draw**

Axis.**draw**(artist, renderer, *args, **kwargs)
Draw the axis lines, grid lines, tick lines and labels

**matplotlib.axis.Axis.findobj**

Axis.**findobj**(match=None, include_self=True)
Find artist objects.
Recursively find all Artist instances contained in self.

*match* can be

- None: return all objects contained in artist.
- function with signature boolean = match(artist) used to filter matches
- class instance: e.g., Line2D. Only return artists of class type.

If *include_self* is True (default), include self in the list to be checked for a match.

**matplotlib.axis.Axis.format_cursor_data**

Axis.**format_cursor_data**(data)
Return cursor data string formatted.

**matplotlib.axis.Axis.get_agg_filter**

Axis.**get_agg_filter**()
return filter function to be used for agg filter
Matplotlib, Release 2.0.2

**matplotlib.axis.Axis.get_alpha**

Axis.get_alpha()

Return the alpha value used for blending - not supported on all backends

**matplotlib.axis.Axis.get_animated**

Axis.get_animated()

Return the artist's animated state

**matplotlib.axis.Axis.get_axes**

Axis.get_axes()

Return the Axes instance the artist resides in, or None.

This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

**matplotlib.axis.Axis.get_children**

Axis.get_children()

**matplotlib.axis.Axis.get_clip_box**

Axis.get_clip_box()

Return artist clipbox

**matplotlib.axis.Axis.get_clip_on**

Axis.get_clip_on()

Return whether artist uses clipping

**matplotlib.axis.Axis.get_clip_path**

Axis.get_clip_path()

Return artist clip path

**matplotlib.axis.Axis.get_contains**

Axis.get_contains()

Return the _contains test used by the artist, or None for default.
**matplotlib.axis.Axis.get_cursor_data**

`Axis.get_cursor_data(event)`  
Get the cursor data for a given event.

**matplotlib.axis.Axis.get_figure**

`Axis.get_figure()`  
Return the `Figure` instance the artist belongs to.

**matplotlib.axis.Axis.get_gid**

`Axis.get_gid()`  
Returns the group id

**matplotlib.axis.Axis.get_label**

`Axis.get_label()`  
Return the axis label as a `Text` instance

**matplotlib.axis.Axis.get_path_effects**

`Axis.get_path_effects()`

**matplotlib.axis.Axis.get_picker**

`Axis.get_picker()`  
Return the picker object used by this artist

**matplotlib.axis.Axis.get_rasterized**

`Axis.get_rasterized()`  
return True if the artist is to be rasterized

**matplotlib.axis.Axis.get_sketch_params**

`Axis.get_sketch_params()`  
Returns the sketch parameters for the artist.
Returns sketch_params : tuple or None

A 3-tuple with the following elements:

- scale: The amplitude of the wiggle perpendicular to the source line.
- length: The length of the wiggle along the line.
- randomness: The scale factor by which the length is shrunken or expanded.

May return None if no sketch parameters were set.

matplotlib.axis.Axis.get_snap

Axis.get_snap()

Returns the snap setting which may be:

- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

matplotlib.axis.Axis.get_transform

Axis.get_transform()

matplotlib.axis.Axis.get_transformed_clip_path_and_affine

Axis.get_transformed_clip_path_and_affine()

Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

matplotlib.axis.Axis.get_url

Axis.get_url()

Returns the url

matplotlib.axis.Axis.get_visible

Axis.get_visible()

Return the artist’s visibility
**matplotlib.axis.Axis.get_window_extent**

`Axis.get_window_extent(renderer)`

Get the axes bounding box in display space. Subclasses should override for inclusion in the bounding box “tight” calculation. Default is to return an empty bounding box at 0, 0.

Be careful when using this function, the results will not update if the artist window extent of the artist changes. The extent can change due to any changes in the transform stack, such as changing the axes limits, the figure size, or the canvas used (as is done when saving a figure). This can lead to unexpected behavior where interactive figures will look fine on the screen, but will save incorrectly.

**matplotlib.axis.Axis.get_zorder**

`Axis.get_zorder()`

Return the Artist's zorder.

**matplotlib.axis.Axis.have_units**

`Axis.have_units()`

**matplotlib.axis.Axis.hitlist**

`Axis.hitlist(event)`

List the children of the artist which contain the mouse event `event`.

**matplotlib.axis.Axis.is_figure_set**

`Axis.is_figure_set()`

Returns True if the artist is assigned to a `Figure`.

**matplotlib.axis.Axis.is_transform_set**

`Axis.is_transform_set()`

Returns True if Artist has a transform explicitly set.

**matplotlib.axis.Axis.mouseover**

`Axis.mouseover`
`matplotlib.axis.Axis.pchanged`

Axis.pchanged()

Fire an event when property changed, calling all of the registered callbacks.

`matplotlib.axis.Axis.pick`

Axis.pick(mouseevent)

Process pick event

each child artist will fire a pick event if `mouseevent` is over the artist and the artist has picker set

`matplotlib.axis.Axis.pickable`

Axis.pickable()

Return True if Artist is pickable.

`matplotlib.axis.Axis.properties`

Axis.properties()

return a dictionary mapping property name -> value for all Artist props

`matplotlib.axis.Axis.remove`

Axis.remove()

Remove the artist from the figure if possible. The effect will not be visible until the figure is redrawn, e.g., with `matplotlib.axes.Axes.draw_idle()`. Call `matplotlib.axes.Axes.relim()` to update the axes limits if desired.

Note: `relim()` will not see collections even if the collection was added to axes with `autolim = True`.

Note: there is no support for removing the artist’s legend entry.

`matplotlib.axis.Axis.remove_callback`

Axis.remove_callback(oid)

Remove a callback based on its id.

See also:

`add_callback()` For adding callbacks
matplotlib.axis.Axis.set

Axis.set(**kwargs)
A property batch setter. Pass kwargs to set properties.

matplotlib.axis.Axis.set_agg_filter

Axis.set_agg_filter(filter_func)
set agg_filter function.

matplotlib.axis.Axis.set_alpha

Axis.set_alpha(alpha)
Set the alpha value used for blending - not supported on all backends.
ACCEPTS: float (0.0 transparent through 1.0 opaque)

matplotlib.axis.Axis.set_animated

Axis.set_animated(b)
Set the artist’s animation state.
ACCEPTS: [True | False]

matplotlib.axis.Axis.set_axes

Axis.set_axes(axes)
Set the Axes instance in which the artist resides, if any.
This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.
ACCEPTS: an Axes instance

matplotlib.axis.Axis.set_clip_box

Axis.set_clip_box(clipbox)
Set the artist’s clip Bbox.
ACCEPTS: a matplotlib.transforms.Bbox instance

matplotlib.axis.Axis.set_clip_on

Axis.set_clip_on(b)
Set whether artist uses clipping.
When False artists will be visible outside of the axes which can lead to unexpected results.

ACCEPTS: [True | False]

```python
matplotlib.axis.Axis.set_clip_path
```

```python
Axis.set_clip_path(clippath, transform=None)
```

```python
matplotlib.axis.Axis.set_contains
```

```python
Axis.set_contains(picker)
```

Replace the contains test used by this artist. The new picker should be a callable function which determines whether the artist is hit by the mouse event:

```python
hit, props = picker(artist, mouseevent)
```

If the mouse event is over the artist, return `hit = True` and `props` is a dictionary of properties you want returned with the contains test.

ACCEPTS: a callable function

```python
matplotlib.axis.Axis.set_figure
```

```python
Axis.set_figure(fig)
```

Set the `Figure` instance the artist belongs to.

ACCEPTS: a `matplotlib.figure.Figure` instance

```python
matplotlib.axis.Axis.set_gid
```

```python
Axis.set_gid(gid)
```

Sets the (group) id for the artist

ACCEPTS: an id string

```python
matplotlib.axis.Axis.set_label
```

```python
Axis.set_label(s)
```

Set the label to `s` for auto legend.

ACCEPTS: string or anything printable with ‘%s’ conversion.
**matplotlib.axis.Axis.set_path_effects**

Axis.set_path_effects(path_effects)

set path_effects, which should be a list of instances of matplotlib.patheffect._Base class or its derivatives.

**matplotlib.axis.Axis.set_picker**

Axis.set_picker(picker)

Set the epsilon for picking used by this artist

picker can be one of the following:

- *None*: picking is disabled for this artist (default)
- A boolean: if True then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist
- A float: if picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if it’s data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event
- A function: if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```python
hit, props = picker(artist, mouseevent)
```

to determine the hit test. if the mouse event is over the artist, return hit=True and props is a dictionary of properties you want added to the PickEvent attributes.

ACCEPTS: [None|float|boolean| callable]

**matplotlib.axis.Axis.set_rasterized**

Axis.set_rasterized(rasterized)

Force rasterized (bitmap) drawing in vector backend output.

Defaults to None, which implies the backend’s default behavior

ACCEPTS: [True | False | None]

**matplotlib.axis.Axis.set_sketch_params**

Axis.set_sketch_params(scale=None, length=None, randomness=None)

Sets the sketch parameters.

Parameters scale : float, optional
The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is None, or not provided, no sketch filter will be provided.

**length**: float, optional

The length of the wiggle along the line, in pixels (default 128.0)

**randomness**: float, optional

The scale factor by which the length is shrunken or expanded (default 16.0)

```python
matplotlib.axis.Axis.set_snap
```

Axis.set_snap(snap)

Sets the snap setting which may be:

- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

```python
matplotlib.axis.Axis.set_transform
```

Axis.set_transform(t)

Set the Transform instance used by this artist.

ACCEPTS: Transform instance

```python
matplotlib.axis.Axis.set_url
```

Axis.set_url(url)

Sets the url for the artist

ACCEPTS: a url string

```python
matplotlib.axis.Axis.set_visible
```

Axis.set_visible(b)

Set the artist’s visibility.

ACCEPTS: [True | False]
matplotlib.axis.Axis.set_zorder

Axis.set_zorder(level)
   Set the zorder for the artist. Artists with lower zorder values are drawn first.
   ACCEPTS: any number

matplotlib.axis.Axis.stale

Axis.stale
   If the artist is ‘stale’ and needs to be re-drawn for the output to match the internal state of the artist.

matplotlib.axis.Axis.update

Axis.update(props)
   Update the properties of this Artist from the dictionary prop.

matplotlib.axis.Axis.update_from

Axis.update_from(other)
   Copy properties from other to self.

matplotlib.axis.Axis.zorder

Axis.zorder = 0

matplotlib.axis.XAxis.add_callback

XAxis.add_callback(func)
   Adds a callback function that will be called whenever one of the Artist’s properties changes.
   Returns an id that is useful for removing the callback with remove_callback() later.

matplotlib.axis.XAxis.aname

XAxis.aname = ‘Artist’
**matplotlib.axis.XAxis.axes**

The *Axes* instance the artist resides in, or *None*.

**matplotlib.axis.XAxis.contains**

*XAxis.contains*(mouseevent)

Test whether the mouse event occurred in the x axis.

**matplotlib.axis.XAxis.convert_xunits**

*XAxis.convert_xunits*(x)

For artists in an axes, if the xaxis has units support, convert *x* using xaxis unit type.

**matplotlib.axis.XAxis.convert_yunits**

*XAxis.convert_yunits*(y)

For artists in an axes, if the yaxis has units support, convert *y* using yaxis unit type.

**matplotlib.axis.XAxis.draw**

*XAxis.draw*(artist, renderer, *args, **kwargs)

Draw the axis lines, grid lines, tick lines and labels.

**matplotlib.axis.XAxis.findobj**

*XAxis.findobj*(match=None, include_self=True)

Find artist objects.

Recursively find all *Artist* instances contained in self.

*match* can be

- None: return all objects contained in artist.
- function with signature `boolean = match(artist)` used to filter matches
- class instance: e.g., Line2D. Only return artists of class type.

If *include_self* is True (default), include self in the list to be checked for a match.
**matplotlib.axis.XAxis.format_cursor_data**

XAxis.format_cursor_data(data)
Return cursor data string formatted.

**matplotlib.axis.XAxis.get_agg_filter**

XAxis.get_agg_filter()
return filter function to be used for agg filter

**matplotlib.axis.XAxis.get_alpha**

XAxis.get_alpha()
Return the alpha value used for blending - not supported on all backends

**matplotlib.axis.XAxis.get_animated**

XAxis.get_animated()
Return the artist’s animated state

**matplotlib.axis.XAxis.get_axes**

XAxis.get_axes()
Return the Axes instance the artist resides in, or None.
This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

**matplotlib.axis.XAxis.get_children**

XAxis.get_children()

**matplotlib.axis.XAxis.get_clip_box**

XAxis.get_clip_box()
Return artist clipbox

**matplotlib.axis.XAxis.get_clip_on**

XAxis.get_clip_on()
Return whether artist uses clipping
matplotlib.axis.XAxis.get_clip_path

XAxis.get_clip_path()
    Return artist clip path

matplotlib.axis.XAxis.get_contains

XAxis.get_contains()
    Return the _contains test used by the artist, or None for default.

matplotlib.axis.XAxis.get_cursor_data

XAxis.get_cursor_data(event)
    Get the cursor data for a given event.

matplotlib.axis.XAxis.get_figure

XAxis.get_figure()
    Return the Figure instance the artist belongs to.

matplotlib.axis.XAxis.get_gid

XAxis.get_gid()
    Returns the group id

matplotlib.axis.XAxis.get_label

XAxis.get_label()
    Return the axis label as a Text instance

matplotlib.axis.XAxis.get_path_effects

XAxis.get_path_effects()

matplotlib.axis.XAxis.get_picker

XAxis.get_picker()
    Return the picker object used by this artist
matplotlib.axis.XAxis.get_rasterized

XAxis.get_rasterized()
return True if the artist is to be rasterized

matplotlib.axis.XAxis.get_sketch_params

XAxis.get_sketch_params()
Returns the sketch parameters for the artist.

Returns sketch_params : tuple or None
A 3-tuple with the following elements:

- scale: The amplitude of the wiggle perpendicular to the source line.
- length: The length of the wiggle along the line.
- randomness: The scale factor by which the length is shrunken or expanded.

May return None if no sketch parameters were set.

matplotlib.axis.XAxis.get_snap

XAxis.get_snap()
Returns the snap setting which may be:

- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

matplotlib.axis.XAxis.get_transform

XAxis.get_transform()

matplotlib.axis.XAxis.get_transformed_clip_path_and_affine

XAxis.get_transformed_clip_path_and_affine()
Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.
Matplotlib, Release 2.0.2

matplotlib.axis.XAxis.get_url

XAxis.get_url()
Returns the url

matplotlib.axis.XAxis.get_visible

XAxis.get_visible()
Return the artist’s visibility

matplotlib.axis.XAxis.get_window_extent

XAxis.get_window_extent(renderer)
Get the axes bounding box in display space. Subclasses should override for inclusion in the bounding box “tight” calculation. Default is to return an empty bounding box at 0, 0.

Be careful when using this function, the results will not update if the artist window extent of the artist changes. The extent can change due to any changes in the transform stack, such as changing the axes limits, the figure size, or the canvas used (as is done when saving a figure). This can lead to unexpected behavior where interactive figures will look fine on the screen, but will save incorrectly.

matplotlib.axis.XAxis.get_zorder

XAxis.get_zorder()
Return the Artist's zorder.

matplotlib.axis.XAxis.have_units

XAxis.have_units()

matplotlib.axis.XAxis.hitlist

XAxis.hitlist(event)
List the children of the artist which contain the mouse event event.

matplotlib.axis.XAxis.is_figure_set

XAxis.is_figure_set()
Returns True if the artist is assigned to a Figure.
matplotlib.axis.XAxis.is_transform_set

XAxis.is_transform_set()

Returns True if Artist has a transform explicitly set.

matplotlib.axis.XAxis.mouseover

XAxis.mouseover

matplotlib.axis.XAxis.pchanged

XAxis.pchanged()

Fire an event when property changed, calling all of the registered callbacks.

matplotlib.axis.XAxis.pick

XAxis.pick(mouseevent)

Process pick event

each child artist will fire a pick event if mouseevent is over the artist and the artist has picker set

matplotlib.axis.XAxis.pickable

XAxis.pickable()

Return True if Artist is pickable.

matplotlib.axis.XAxis.properties

XAxis.properties()

return a dictionary mapping property name -> value for all Artist props

matplotlib.axis.XAxis.remove

XAxis.remove()

Remove the artist from the figure if possible. The effect will not be visible until the figure is redrawn, e.g., with matplotlib.axes.Axes.draw_idle(). Call matplotlib.axes.Axes.relim() to update the axes limits if desired.

Note: relim() will not see collections even if the collection was added to axes with autolim = True.

Note: there is no support for removing the artist’s legend entry.
**matplotlib.axis.XAxis.remove_callback**

`XAxis.remove_callback(oid)`

Remove a callback based on its *id*.

**See also:**

*add_callback()* For adding callbacks

**matplotlib.axis.XAxis.set**

`XAxis.set(**kwargs)`

A property batch setter. Pass *kwargs* to set properties.

**matplotlib.axis.XAxis.set_agg_filter**

`XAxis.set_agg_filter(filter_func)`

set agg_filter fuction.

**matplotlib.axis.XAxis.set_alpha**

`XAxis.set_alpha(alpha)`

Set the alpha value used for blending - not supported on all backends.

*ACCEPTS*: float (0.0 transparent through 1.0 opaque)

**matplotlib.axis.XAxis.set_animated**

`XAxis.set_animated(b)`

Set the artist’s animation state.

*ACCEPTS*: [True | False]

**matplotlib.axis.XAxis.set_axes**

`XAxis.set_axes(axes)`

Set the *Axes* instance in which the artist resides, if any.

This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

*ACCEPTS*: an *Axes* instance
matplotlib.axis.XAxis.set_clip_box

XAxis.set_clip_box(clipbox)
Set the artist’s clip Bbox.
ACCEPTS: a matplotlib.transforms.Bbox instance

matplotlib.axis.XAxis.set_clip_on

XAxis.set_clip_on(b)
Set whether artist uses clipping.
When False artists will be visible out side of the axes which can lead to unexpected results.
ACCEPTS: [True | False]

matplotlib.axis.XAxis.set_clip_path

XAxis.set_clip_path(clippath, transform=None)

matplotlib.axis.XAxis.set_contains

XAxis.set_contains(picker)
Replace the contains test used by this artist. The new picker should be a callable function which determines whether the artist is hit by the mouse event:

```python
hit, props = picker(artist, mouseevent)
```
If the mouse event is over the artist, return hit = True and props is a dictionary of properties you want returned with the contains test.
ACCEPTS: a callable function

matplotlib.axis.XAxis.set_figure

XAxis.set_figure(fig)
Set the Figure instance the artist belongs to.
ACCEPTS: a matplotlib.figure.Figure instance

matplotlib.axis.XAxis.set_gid

XAxis.set_gid(gid)
Sets the (group) id for the artist
ACCEPTS: an id string

**matplotlib.axis.XAxis.set_label**

XAxis.set_label(s)
Set the label to s for auto legend.
ACCEPTS: string or anything printable with ‘%s’ conversion.

**matplotlib.axis.XAxis.set_path_effects**

XAxis.set_path_effects(path_effects)
set path_effects, which should be a list of instances of matplotlib.pathEffects._Base class or its derivatives.

**matplotlib.axis.XAxis.set_picker**

XAxis.set_picker(picker)
Set the epsilon for picking used by this artist

picker can be one of the following:

- **None**: picking is disabled for this artist (default)
- A boolean: if True then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist
- A float: if picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if it’s data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event
- A function: if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:
  
  ```python
  hit, props = picker(artist, mouseevent)
  ```

  to determine the hit test. if the mouse event is over the artist, return hit=True and props is a dictionary of properties you want added to the PickEvent attributes.

ACCEPTS: [None|float|boolean|callable]

**matplotlib.axis.XAxis.set_rasterized**

XAxis.set_rasterized(rasterized)
Force rasterized (bitmap) drawing in vector backend output.
Defaults to None, which implies the backend’s default behavior
ACCEPTS: [True | False | None]

**matplotlib.axis.XAxis.set_sketch_params**

```python
def XAxis.set_sketch_params(scale=None, length=None, randomness=None):
```

Sets the sketch parameters.

**Parameters**

- **scale** : float, optional
  
  The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is None, or not provided, no sketch filter will be provided.

- **length** : float, optional
  
  The length of the wiggle along the line, in pixels (default 128.0)

- **randomness** : float, optional
  
  The scale factor by which the length is shrunken or expanded (default 16.0)

**matplotlib.axis.XAxis.set_snap**

```python
def XAxis.set_snap(snap):
```

Sets the snap setting which may be:

- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

**matplotlib.axis.XAxis.set_transform**

```python
def XAxis.set_transform(t):
```

Set the `Transform` instance used by this artist.

ACCEPTS: `Transform` instance

**matplotlib.axis.XAxis.set_url**

```python
def XAxis.set_url(url):
```

Sets the url for the artist

ACCEPTS: a url string
**Matplotlib, Release 2.0.2**

**matplotlib.axis.XAxis.set_visible**

```python
XAxis.set_visible(b)
```

Set the artist's visibility.

ACCE此类：[True | False]

**matplotlib.axis.XAxis.set_zorder**

```python
XAxis.set_zorder(level)
```

Set the zorder for the artist. Artists with lower zorder values are drawn first.

ACCE此类：any number

**matplotlib.axis.XAxis.stale**

```python
XAxis.stale
```

If the artist is ‘stale’ and needs to be re-drawn for the output to match the internal state of the artist.

**matplotlib.axis.XAxis.update**

```python
XAxis.update(props)
```

Update the properties of this Artist from the dictionary prop.

**matplotlib.axis.XAxis.update_from**

```python
XAxis.update_from(other)
```

Copy properties from other to self.

**matplotlib.axis.XAxis.zorder**

```python
XAxis.zorder = 0
```

**matplotlib.axis.YAxis.add_callback**

```python
YAxis.add_callback(func)
```

Adds a callback function that will be called whenever one of the Artist's properties changes.

Returns an id that is useful for removing the callback with remove_callback() later.
**matplotlib.axis.YAxis.aname**

\[ \text{YAxis.aname} = \text{‘Artist’} \]

**matplotlib.axis.YAxis.axes**

\[ \text{YAxis.axes} \]

The \textit{Axes} instance the artist resides in, or \textit{None}.

**matplotlib.axis.YAxis.contains**

\[ \text{YAxis.contains}(\text{mouseevent}) \]

Test whether the mouse event occurred in the y axis.

Returns \textit{True} \mid \textit{False}

**matplotlib.axis.YAxis.convert_xunits**

\[ \text{YAxis.convert_xunits}(x) \]

For artists in an axes, if the xaxis has units support, convert \(x\) using xaxis unit type

**matplotlib.axis.YAxis.convert_yunits**

\[ \text{YAxis.convert_yunits}(y) \]

For artists in an axes, if the yaxis has units support, convert \(y\) using yaxis unit type

**matplotlib.axis.YAxis.draw**

\[ \text{YAxis.draw}(\text{artist}, \text{renderer}, *\text{args}, **\text{kwargs}) \]

Draw the axis lines, grid lines, tick lines and labels

**matplotlib.axis.YAxis.findobj**

\[ \text{YAxis.findobj}(\text{match}=\text{None}, \text{include_self}=\text{True}) \]

Find artist objects.

Recursively find all \textit{Artist} instances contained in self.

\textit{match} can be

- \text{None}: return all objects contained in artist.

- \text{function with signature boolean } = \text{match(\text{artist})} used to filter matches
class instance: e.g., Line2D. Only return artists of class type.

If include_self is True (default), include self in the list to be checked for a match.

**matplotlib.axis.YAxis.format_cursor_data**

YAxis.format_cursor_data(data)

Return cursor data string formatted.

**matplotlib.axis.YAxis.get_agg_filter**

YAxis.get_agg_filter()

return filter function to be used for agg filter

**matplotlib.axis.YAxis.get_alpha**

YAxis.get_alpha()

Return the alpha value used for blending - not supported on all backends

**matplotlib.axis.YAxis.get_animated**

YAxis.get_animated()

Return the artist’s animated state

**matplotlib.axis.YAxis.get_axes**

YAxis.get_axes()

Return the Axes instance the artist resides in, or None.

This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

**matplotlib.axis.YAxis.get_children**

YAxis.get_children()

**matplotlib.axis.YAxis.get_clip_box**

YAxis.get_clip_box()

Return artist clipbox
**matplotlib.axis.YAxis.get_clip_on**

`YAxis.get_clip_on()`  
Return whether artist uses clipping

**matplotlib.axis.YAxis.get_clip_path**

`YAxis.get_clip_path()`  
Return artist clip path

**matplotlib.axis.YAxis.get_contains**

`YAxis.get_contains()`  
Return the _contains test used by the artist, or *None* for default.

**matplotlib.axis.YAxis.get_cursor_data**

`YAxis.get_cursor_data(event)`  
Get the cursor data for a given event.

**matplotlib.axis.YAxis.get_figure**

`YAxis.get_figure()`  
Return the *Figure* instance the artist belongs to.

**matplotlib.axis.YAxis.get_gid**

`YAxis.get_gid()`  
Returns the group id

**matplotlib.axis.YAxis.get_label**

`YAxis.get_label()`  
Return the axis label as a Text instance

**matplotlib.axis.YAxis.get_path_effects**

`YAxis.get_path_effects()`
**matplotlib.axis.YAxis.get_picker**

YAxis.get_picker()

Return the picker object used by this artist

**matplotlib.axis.YAxis.get_rasterized**

YAxis.get_rasterized()

return True if the artist is to be rasterized

**matplotlib.axis.YAxis.get_sketch_params**

YAxis.get_sketch_params()

Returns the sketch parameters for the artist.

**Returns sketch_params**: tuple or None

A 3-tuple with the following elements:

- scale: The amplitude of the wiggle perpendicular to the source line.
- length: The length of the wiggle along the line.
- randomness: The scale factor by which the length is shrunken or expanded.

May return None if no sketch parameters were set.

**matplotlib.axis.YAxis.get_snap**

YAxis.get_snap()

Returns the snap setting which may be:

- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

**matplotlib.axis.YAxis.get_transform**

YAxis.get_transform()
matplotlib.axis.YAxis.get_transformed_clip_path_and_affine

YAxis.get_transformed_clip_path_and_affine()

Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

matplotlib.axis.YAxis.get_url

YAxis.get_url()

Returns the url

matplotlib.axis.YAxis.get_visible

YAxis.get_visible()

Return the artist’s visibility

matplotlib.axis.YAxis.get_window_extent

YAxis.get_window_extent(renderer)

Get the axes bounding box in display space. Subclasses should override for inclusion in the bounding box “tight” calculation. Default is to return an empty bounding box at 0, 0.

Be careful when using this function, the results will not update if the artist window extent of the artist changes. The extent can change due to any changes in the transform stack, such as changing the axes limits, the figure size, or the canvas used (as is done when saving a figure). This can lead to unexpected behavior where interactive figures will look fine on the screen, but will save incorrectly.

matplotlib.axis.YAxis.get_zorder

YAxis.get_zorder()

Return the Artist’s zorder.

matplotlib.axis.YAxis.have_units

YAxis.have_units()

matplotlib.axis.YAxis.hitlist

YAxis.hitlist(event)

List the children of the artist which contain the mouse event event.
matplotlib.axis.YAxis.is_figure_set

YAxis.is_figure_set()

Returns True if the artist is assigned to a Figure.

matplotlib.axis.YAxis.is_transform_set

YAxis.is_transform_set()

Returns True if Artist has a transform explicitly set.

matplotlib.axis.YAxis.mouseover

YAxis.mouseover

matplotlib.axis.YAxis.pchanged

YAxis.pchanged()

Fire an event when property changed, calling all of the registered callbacks.

matplotlib.axis.YAxis.pick

YAxis.pick(mouseevent)

Process pick event

each child artist will fire a pick event if mouseevent is over the artist and the artist has picker set

matplotlib.axis.YAxis.pickable

YAxis.pickable()

Return True if Artist is pickable.

matplotlib.axis.YAxis.properties

YAxis.properties()

return a dictionary mapping property name -> value for all Artist props

matplotlib.axis.YAxis.remove

YAxis.remove()

Remove the artist from the figure if possible. The effect will not be visible until the figure is redrawn,
e.g., with `matplotlib.axes.Axes.draw_idle()`. Call `matplotlib.axes.Axes.relim()` to update the axes limits if desired.

Note: `relim()` will not see collections even if the collection was added to axes with `autolim = True`.

Note: there is no support for removing the artist’s legend entry.

```matlab
matplotlib.axis.YAxis.remove_callback
```

`YAxis.remove_callback(oid)`

Remove a callback based on its `id`.

See also:

`add_callback()` For adding callbacks

```matlab
matplotlib.axis.YAxis.set
```

`YAxis.set(**kwargs)`

A property batch setter. Pass `kwargs` to set properties.

```matlab
matplotlib.axis.YAxis.set_agg_filter
```

`YAxis.set_agg_filter(filter_func)`

Set agg_filter function.

```matlab
matplotlib.axis.YAxis.set_alpha
```

`YAxis.set_alpha(alpha)`

Set the alpha value used for blending - not supported on all backends.

ACCEPTS: float (0.0 transparent through 1.0 opaque)

```matlab
matplotlib.axis.YAxis.set_animated
```

`YAxis.set_animated(b)`

Set the artist’s animation state.

ACCEPTS: [True | False]

```matlab
matplotlib.axis.YAxis.set_axes
```

`YAxis.set_axes(axes)`

Set the `Axes` instance in which the artist resides, if any.

20.4. Common and inherited methods
Matplotlib, Release 2.0.2

This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

ACCEPTS: an Axes instance

**matplotlib.axis.YAxis.set_clip_box**

YAxis.set_clip_box(clipbox)
Set the artist’s clip Bbox.

ACCEPTS: a matplotlib.transforms.Bbox instance

**matplotlib.axis.YAxis.set_clip_on**

YAxis.set_clip_on(b)
Set whether artist uses clipping.
When False artists will be visible out side of the axes which can lead to unexpected results.

ACCEPTS: [True | False]

**matplotlib.axis.YAxis.set_clip_path**

YAxis.set_clip_path(clippath, transform=None)

**matplotlib.axis.YAxis.set_contains**

YAxis.set_contains(picker)
Replace the contains test used by this artist. The new picker should be a callable function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

If the mouse event is over the artist, return hit = True and props is a dictionary of properties you want returned with the contains test.

ACCEPTS: a callable function

**matplotlib.axis.YAxis.set_figure**

YAxis.set_figure(fig)
Set the Figure instance the artist belongs to.

ACCEPTS: a matplotlib.figure.Figure instance
**matplotlib.axis.YAxis.set_gid**

YAxis.set_gid(gid)
Sets the (group) id for the artist

ACCEPTS: an id string

**matplotlib.axis.YAxis.set_label**

YAxis.set_label(s)
Set the label to s for auto legend.

ACCEPTS: string or anything printable with ‘%s’ conversion.

**matplotlib.axis.YAxis.set_path_effects**

YAxis.set_path_effects(path_effects)
set path_effects, which should be a list of instances of matplotlib.path.effects._Base class or its derivatives.

**matplotlib.axis.YAxis.set_picker**

YAxis.set_picker(picker)
Set the epsilon for picking used by this artist

picker can be one of the following:

- *None*: picking is disabled for this artist (default)
- *A boolean*: if True then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist
- *A float*: if picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if it’s data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event
- *A function*: if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```python
hit, props = picker(artist, mouseevent)
```

to determine the hit test. if the mouse event is over the artist, return hit=True and props is a dictionary of properties you want added to the PickEvent attributes.

ACCEPTS: [None|float|boolean|callable]
**matplotlib.axis.YAxis.set_rasterized**

YAxis.set_rasterized(rasterized)

Force rasterized (bitmap) drawing in vector backend output.
Defaults to None, which implies the backend’s default behavior
ACCEPTS: [True | False | None]

**matplotlib.axis.YAxis.set_sketch_params**

YAxis.set_sketch_params(scale=None, length=None, randomness=None)

Sets the sketch parameters.

- **Parameters**
  - scale : float, optional
    The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is None, or not provided, no sketch filter will be provided.
  - length : float, optional
    The length of the wiggle along the line, in pixels (default 128.0)
  - randomness : float, optional
    The scale factor by which the length is shrunken or expanded (default 16.0)

**matplotlib.axis.YAxis.set_snap**

YAxis.set_snap(snapshot)

Sets the snap setting which may be:
- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

**matplotlib.axis.YAxis.set_transform**

YAxis.set_transform(t)

Set the Transform instance used by this artist.

ACCEPTS: Transform instance
```python
matplotlib.axis.YAxis.set_url

YAxis.set_url(url)
    Sets the url for the artist
    ACCEPTS: a url string
```

```python
matplotlib.axis.YAxis.set_visible

YAxis.set_visible(b)
    Set the artist’s visibility.
    ACCEPTS: [True | False]
```

```python
matplotlib.axis.YAxis.set_zorder

YAxis.set_zorder(level)
    Set the zorder for the artist. Artists with lower zorder values are drawn first.
    ACCEPTS: any number
```

```python
matplotlib.axis.YAxis.stale

YAxis.stale
    If the artist is ‘stale’ and needs to be re-drawn for the output to match the internal state of the artist.
```

```python
matplotlib.axis.YAxis.update

YAxis.update(props)
    Update the properties of this Artist from the dictionary prop.
```

```python
matplotlib.axis.YAxis.update_from

YAxis.update_from(other)
    Copy properties from other to self.
```

```python
matplotlib.axis.YAxis.zorder

YAxis.zorder = 0
```
CHAPTER TWENTYONE

BACKENDS

21.1 matplotlib.backend_bases

Abstract base classes define the primitives that renderers and graphics contexts must implement to serve as a matplotlib backend

RendererBase  An abstract base class to handle drawing/rendering operations.

FigureCanvasBase The abstraction layer that separates the matplotlib.figure.Figure from the backend specific details like a user interface drawing area

GraphicsContextBase An abstract base class that provides color, line styles, etc...

Event The base class for all of the matplotlib event handling. Derived classes such as KeyEvent and MouseEvent store the meta data like keys and buttons pressed, x and y locations in pixel and Axes coordinates.

ShowBase The base class for the Show class of each interactive backend; the 'show' callable is then set to Show.__call__, inherited from ShowBase.

ToolContainerBase The base class for the Toolbar class of each interactive backend.

StatusbarBase The base class for the messaging area.

class matplotlib.backend_bases.CloseEvent(name, canvas, guiEvent=None)
Bases: matplotlib.backend_bases.Event

An event triggered by a figure being closed

In addition to the Event attributes, the following event attributes are defined:

class matplotlib.backend_bases.DrawEvent(name, canvas, renderer)
Bases: matplotlib.backend_bases.Event

An event triggered by a draw operation on the canvas

In addition to the Event attributes, the following event attributes are defined:

renderer the RendererBase instance for the draw event

class matplotlib.backend_bases.Event(name, canvas, guiEvent=None)
Bases: object
A matplotlib event. Attach additional attributes as defined in `FigureCanvasBase.mpl_connect()`. The following attributes are defined and shown with their default values

**name** the event name

**canvas** the FigureCanvas instance generating the event

**guiEvent** the GUI event that triggered the matplotlib event

```python
class matplotlib.backend_bases.FigureCanvasBase(figure)
Bases: object
The canvas the figure renders into.
Public attributes

    figure A matplotlib.figure.Figure instance

    blit(bbox=None)
    blit the canvas in bbox (default entire canvas)

    button_press_event(x, y, button, dblclick=False, guiEvent=None)
    Backend derived classes should call this function on any mouse button press. x,y are the canvas coords: 0,0 is lower, left. button and key are as defined in MouseEvent.
    This method will be call all functions connected to the ‘button_press_event’ with a MouseEvent instance.

    button_release_event(x, y, button, guiEvent=None)
    Backend derived classes should call this function on any mouse button release.
    x the canvas coordinates where 0=left
    y the canvas coordinates where 0=bottom
    guiEvent the native UI event that generated the mpl event
    This method will be call all functions connected to the ‘button_release_event’ with a MouseEvent instance.

    close_event(guiEvent=None)
    This method will be called by all functions connected to the ‘close_event’ with a CloseEvent

    draw(*args, **kwargs)
    Render the Figure

    draw_cursor(event)
    Draw a cursor in the event.axes if inaxes is not None. Use native GUI drawing for efficiency if possible

    draw_event(renderer)
    This method will be call all functions connected to the ‘draw_event’ with a DrawEvent

    draw_idle(*args, **kwargs)
    draw() only if idle; defaults to draw but backends can override

    enter_notify_event(guiEvent=None, xy=None)
    Backend derived classes should call this function when entering canvas
```
**guiEvent** the native UI event that generated the mpl event

**xy** the coordinate location of the pointer when the canvas is entered

events = ['resize_event', 'draw_event', 'key_press_event', 'key_release_event', 'button_press_event', 'button_release_event', ...

filetypes = {'ps': 'Postscript', 'eps': 'Encapsulated Postscript', 'pdf': 'Portable Document Format', 'pgf': 'PGF code for LaTeX', ...

fixed_dpi = None

flush_events()
Flush the GUI events for the figure. Implemented only for backends with GUIs.

get_default_filename()
Return a string, which includes extension, suitable for use as a default filename.

classmethod get_default_filetype()
Get the default savefig file format as specified in rcParam savefig.format. Returned string excludes period. Overridden in backends that only support a single file type.

classmethod get_supported_filetypes()
Return dict of savefig file formats supported by this backend

classmethod get_supported_filetypes_grouped()
Return a dict of savefig file formats supported by this backend, where the keys are a file type name, such as ‘Joint Photographic Experts Group’, and the values are a list of filename extensions used for that filetype, such as ['jpg', 'jpeg'].

get_width_height()
Return the figure width and height in points or pixels (depending on the backend), truncated to integers

get_window_title()
Get the title text of the window containing the figure. Return None if there is no window (e.g., a PS backend).

grab_mouse(ax)
Set the child axes which are currently grabbing the mouse events. Usually called by the widgets themselves. It is an error to call this if the mouse is already grabbed by another axes.

idle_event(guiEvent=None)
Called when GUI is idle.

is_saving()
Returns True when the renderer is in the process of saving to a file, rather than rendering for an on-screen buffer.

key_press_event(key, guiEvent=None)
This method will be call all functions connected to the ‘key_press_event’ with a KeyEvent

key_release_event(key, guiEvent=None)
This method will be call all functions connected to the ‘key_release_event’ with a KeyEvent
**leave_notify_event**(guiEvent=None)

Backend derived classes should call this function when leaving canvas

*guiEvent* the native UI event that generated the mpl event

**motion_notify_event**(x, y, guiEvent=None)

Backend derived classes should call this function on any motion-notify-event.

*x* the canvas coordinates where 0=left

*y* the canvas coordinates where 0=bottom

*guiEvent* the native UI event that generated the mpl event

This method will be call all functions connected to the ‘motion_notify_event’ with a *MouseEvent* instance.

**mpl_connect**(s, func)

Connect event with string *s* to *func*. The signature of *func* is:

```
def func(event)
```

where event is a *matplotlib.backend_bases.Event*. The following events are recognized

- ‘button_press_event’
- ‘button_release_event’
- ‘draw_event’
- ‘key_press_event’
- ‘key_release_event’
- ‘motion_notify_event’
- ‘pick_event’
- ‘resize_event’
- ‘scroll_event’
- ‘figure_enter_event’,
- ‘figure_leave_event’,
- ‘axes_enter_event’,
- ‘axes_leave_event’
- ‘close_event’

For the location events (button and key press/release), if the mouse is over the axes, the variable event.inaxes will be set to the *Axes* the event occurs is over, and additionally, the variables event.xdata and event.ydata will be defined. This is the mouse location in data coords. See *KeyEvent* and *MouseEvent* for more info.

Return value is a connection id that can be used with mpl_disconnect().

Example usage:
```python
def on_press(event):
    print('you pressed', event.button, event.xdata, event.ydata)

cid = canvas.mpl_connect('button_press_event', on_press)
```

**mpl_disconnect(cid)**
Disconnect callback id cid

Example usage:
```
cid = canvas.mpl_connect('button_press_event', on_press)
#...later
canvas.mpl_disconnect(cid)
```

**new_timer(*args, **kwargs)**
Creates a new backend-specific subclass of `backend_bases.Timer`. This is useful for getting periodic events through the backend’s native event loop. Implemented only for backends with GUIs.

optional arguments:
- `interval` Timer interval in milliseconds
- `callbacks` Sequence of (func, args, kwargs) where `func(args, **kwargs)` will be executed by the timer every `*interval`.

**onHilite(ev)**
Mouse event processor which highlights the artists under the cursor. Connect this to the `motion_notify_event` using:
```
canvas.mpl_connect('motion_notify_event', canvas.onHilite)
```

**onRemove(ev)**
Mouse event processor which removes the top artist under the cursor. Connect this to the `mouse_press_event` using:
```
canvas.mpl_connect('mouse_press_event', canvas.onRemove)
```

**pick(mouseevent)**

**pick_event(mouseevent, artist, **kwargs)**
This method will be called by artists who are picked and will fire off `PickEvent` callbacks registered listeners

**print_figure(filename, dpi=None, facecolor=None, edgecolor=None, orientation='portrait', format=None, **kwargs)**
Render the figure to hardcopy. Set the figure patch face and edge colors. This is useful because some of the GUIs have a gray figure face color background and you’ll probably want to override this on hardcopy.

Arguments are:
- `filename` can also be a file object on image backends
**orientation** only currently applies to PostScript printing.

**dpi** the dots per inch to save the figure in; if None, use savefig.dpi

**facecolor** the facecolor of the figure; if None, defaults to savefig.facecolor

**edgecolor** the edgecolor of the figure; if None, defaults to savefig.edgecolor

**orientation** landscape’ | ‘portrait’ (not supported on all backends)

**format** when set, forcibly set the file format to save to

**bbox_inches** Bbox in inches. Only the given portion of the figure is saved. If ‘tight’, try to figure out the tight bbox of the figure. If None, use savefig.bbox

**pad_inches** Amount of padding around the figure when bbox_inches is ‘tight’. If None, use savefig.pad_inches

**bbox_extra_artists** A list of extra artists that will be considered when the tight bbox is calculated.

**release_mouse**(*ax*)
Release the mouse grab held by the axes, ax. Usually called by the widgets. It is ok to call this even if you ax doesn’t have the mouse grab currently.

**resize**(*w, h*)
set the canvas size in pixels

**resize_event**()
This method will be call all functions connected to the ‘resize_event’ with a ResizeEvent

**scroll_event**(*x, y, step, guiEvent=None*)
Backend derived classes should call this function on any scroll wheel event. x,y are the canvas coords: 0,0 is lower, left. button and key are as defined in MouseEvent.

This method will be call all functions connected to the ‘scroll_event’ with a MouseEvent instance.

**set_window_title**(*title*)
Set the title text of the window containing the figure. Note that this has no effect if there is no window (e.g., a PS backend).

**start_event_loop**(*timeout*)
Start an event loop. This is used to start a blocking event loop so that interactive functions, such as ginput and waitforbuttonpress, can wait for events. This should not be confused with the main GUI event loop, which is always running and has nothing to do with this.

This is implemented only for backends with GUls.

**start_event_loop_default**(*timeout=0*)
Start an event loop. This is used to start a blocking event loop so that interactive functions, such as ginput and waitforbuttonpress, can wait for events. This should not be confused with the main GUI event loop, which is always running and has nothing to do with this.

This function provides default event loop functionality based on time.sleep that is meant to be used until event loop functions for each of the GUI backends can be written. As such, it throws a deprecated warning.
This call blocks until a callback function triggers `stop_event_loop()` or `timeout` is reached. If `timeout` is <=0, never timeout.

**stop_event_loop()**
Stop an event loop. This is used to stop a blocking event loop so that interactive functions, such as `ginput` and `waitforbuttonpress`, can wait for events.

This is implemented only for backends with GUIs.

**stop_event_loop_default()**
Stop an event loop. This is used to stop a blocking event loop so that interactive functions, such as `ginput` and `waitforbuttonpress`, can wait for events.

**supports_blit = True**

**switch_backends(FigureCanvasClass)**
Instantiate an instance of `FigureCanvasClass`

This is used for backend switching, e.g., to instantiate a `FigureCanvasPS` from a `FigureCanvasGTK`. Note, deep copying is not done, so any changes to one of the instances (e.g., setting figure size or line props), will be reflected in the other

**class matplotlib.backend_bases.FigureManagerBase(canvas, num)**
Bases: object

Helper class for `pyplot` mode, wraps everything up into a neat bundle

Public attributes:

- **canvas** A `FigureCanvasBase` instance
- **num** The figure number

**destroy()**

**full_screen_toggle()**

**get_window_title()**
Get the title text of the window containing the figure. Return `None` for non-GUI backends (e.g., a PS backend).

**key_press(event)**
Implement the default mpl key bindings defined at *Navigation Keyboard Shortcuts*

**resize(w, h)**
“For gui backends, resize the window (in pixels).

**set_window_title(title)**
Set the title text of the window containing the figure. Note that this has no effect for non-GUI backends (e.g., a PS backend).

**show()**
For GUI backends, show the figure window and redraw. For non-GUI backends, raise an exception to be caught by `show()`, for an optional warning.
show_popup(msg)
    Display message in a popup – GUI only

class matplotlib.backend_bases.GraphicsContextBase
    Bases: object
    An abstract base class that provides color, line styles, etc...

copy_properties(gc)
    Copy properties from gc to self

get_alpha()
    Return the alpha value used for blending - not supported on all backends

get_antialiased()
    Return true if the object should try to do antialiased rendering

get_capstyle()
    Return the capstyle as a string in ('butt', 'round', 'projecting')

get_clip_path()
    Return the clip path in the form (path, transform), where path is a Path instance, and transform is an affine transform to apply to the path before clipping.

get_clip_rectangle()
    Return the clip rectangle as a Bbox instance

get_dashes()
    Return the dash information as an offset dashlist tuple.
    The dash list is a even size list that gives the ink on, ink off in pixels.
    See p107 of to PostScript BLUEBOOK for more info.
    Default value is None

get_forced_alpha()
    Return whether the value given by get_alpha() should be used to override any other alpha-channel values.

get_gid()
    Return the object identifier if one is set, None otherwise.

get_hatch()
    Gets the current hatch style

get_hatch_color()
    Gets the color to use for hatching.

get_hatch_linewidth()
    Gets the linewidth to use for hatching.

get_hatch_path(density=6.0)
    Returns a Path for the current hatch.

get_joinstyle()
    Return the line join style as one of ('miter', 'round', 'bevel')
get_linestyle(style)
Return the linestyle: one of ('solid', 'dashed', 'dashdot', 'dotted').

get_linewidth()
Return the line width in points as a scalar

get_rgb()
returns a tuple of three or four floats from 0-1.

get_sketch_params()
Returns the sketch parameters for the artist.

Returns sketch_params : tuple or None
A 3-tuple with the following elements:

• scale: The amplitude of the wiggle perpendicular to the source line.
• length: The length of the wiggle along the line.
• randomness: The scale factor by which the length is shrunken or expanded.

May return None if no sketch parameters were set.

get_snap()
returns the snap setting which may be:

• True: snap vertices to the nearest pixel center
• False: leave vertices as-is
• None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

get_url()
returns a url if one is set, None otherwise

restore()
Restore the graphics context from the stack - needed only for backends that save graphics contexts on a stack

set_alpha(alpha)
Set the alpha value used for blending - not supported on all backends. If alpha=None (the default), the alpha components of the foreground and fill colors will be used to set their respective transparencies (where applicable); otherwise, alpha will override them.

set_antialiased(b)
True if object should be drawn with antialiased rendering

set_capstyle(cs)
Set the capstyle as a string in ('butt', 'round', 'projecting')

set_clip_path(path)
Set the clip path and transformation. Path should be a TransformedPath instance.

set_clip_rectangle(rectangle)
Set the clip rectangle with sequence (left, bottom, width, height)
**set_dashes** *(dash_offset, dash_list)*

Set the dash style for the gc.

-dash_offset is the offset (usually 0).

dash_list specifies the on-off sequence as points. (None, None) specifies a solid line.

**set_foreground** *(fg, isRGBA=False)*

Set the foreground color. fg can be a MATLAB format string, a html hex color string, an rgb or rgba unit tuple, or a float between 0 and 1. In the latter case, grayscale is used.

If you know fg is rgba, set isRGBA=True for efficiency.

**set_gid** *(id)*

Sets the id.

**set_graylevel** *(frac)*

Set the foreground color to be a gray level with frac.

**set_hatch** *(hatch)*

Sets the hatch style for filling.

**set_hatch_color** *(hatch_color)*

Sets the color to use for hatching.

**set_joinstyle** *(js)*

Set the join style to be one of (‘miter’, ‘round’, ‘bevel’)

**set_linestyle** *(style)*

Set the linestyle to be one of (‘solid’, ‘dashed’, ‘dashdot’, ‘dotted’). These are defined in the rc-Params `lines.dashed_pattern`, `lines.dashdot_pattern` and `lines.dotted_pattern`. One may also specify customized dash styles by providing a tuple of (offset, dash pairs).

**set_linewidth** *(w)*

Set the linewidth in points.

**set_sketch_params** *(scale=None, length=None, randomness=None)*

Sets the sketch parameters.

Parameters:

- **scale** : float, optional
  The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is None, or not provided, no sketch filter will be provided.

- **length** : float, optional
  The length of the wiggle along the line, in pixels (default 128.0)

- **randomness** : float, optional
  The scale factor by which the length is shrunken or expanded (default 16.0)

**set_snap** *(snap)*

Sets the snap setting which may be:

- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
• None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

```python
set_url(url)
```
Sets the url for links in compatible backends

```python
class matplotlib.backend_bases.IdleEvent(name, canvas, guiEvent=None)
```
Bases: `matplotlib.backend_bases.Event`
An event triggered by the GUI backend when it is idle – useful for passive animation

```python
class matplotlib.backend_bases.KeyEvent(name, canvas, key, x=0, y=0, guiEvent=None)
```
Bases: `matplotlib.backend_bases.LocationEvent`
A key event (key press, key release).

Attach additional attributes as defined in `FigureCanvasBase.mpl_connect()`.

In addition to the `Event` and `LocationEvent` attributes, the following attributes are defined:

**key** the key(s) pressed. Could be **None**, a single case sensitive ascii character ("g", "G", ",", etc.), a special key ("control", "shift", "f1", "up", etc.) or a combination of the above (e.g., "ctrl+alt+g", "ctrl+alt+G").

**Note:** Modifier keys will be prefixed to the pressed key and will be in the order “ctrl”, “alt”, “super”.

The exception to this rule is when the pressed key is itself a modifier key, therefore “ctrl+alt” and “alt+control” can both be valid key values.

Example usage:

```python
def on_key(event):
    print('you pressed', event.key, event.xdata, event.ydata)

cid = fig.canvas.mpl_connect('key_press_event', on_key)
```

```python
class matplotlib.backend_bases.LocationEvent(name, canvas, x, y, guiEvent=None)
```
Bases: `matplotlib.backend_bases.Event`
An event that has a screen location

The following additional attributes are defined and shown with their default values.

In addition to the `Event` attributes, the following event attributes are defined:

- **x** x position - pixels from left of canvas
- **y** y position - pixels from bottom of canvas
- **inaxes** the Axes instance if mouse is over axes
- **xdata** x coord of mouse in data coords
- **ydata** y coord of mouse in data coords
- **x**, **y** in figure coords, 0,0 = bottom, left
**Class**: `matplotlib.backend_bases.MouseEvent`

- **name**
- **canvas**
- **x**
- **y**
- **button**
- **key**
- **step**
- **dblclick**
- **guiEvent**

**Bases**: `matplotlib.backend_bases.LocationEvent`

A mouse event ('button_press_event', 'button_release_event', 'scroll_event', 'motion_notify_event').

In addition to the `Event` and `LocationEvent` attributes, the following attributes are defined:

- **button**
- **key**
- **step**

Key the key depressed when the mouse event triggered (see `KeyEvent`).

Button button pressed None, 1, 2, 3, 'up', 'down' (up and down are used for scroll events). Note that in the hfig backend, both the middle and right clicks return 3 since right clicking will bring up the context menu in some browsers.

Example usage:

```python
def on_press(event):
    print(f'you pressed {event.button}, {event.xdata}, {event.ydata}')

cid = fig.canvas.mpl_connect('button_press_event', on_press)
```

In the above example, `on_press` is defined to print the button pressed, as well as the x and y coordinates of the press event. The `fig.canvas.mpl_connect` method is used to connect the `button_press_event` signal to the `on_press` function.

**Attributes**:

- `inaxes`:
- `lastevent`:
- `x`:
- `y`:
- `xdata`:
- `ydata`:
- `step`:
- `dblclick`:
- `guiEvent`:

**Note**:

In the nbagg backend, both the middle and right clicks return 3 since right clicking will bring up the context menu in some browsers.
\texttt{x} = \texttt{None}

\texttt{xdata} = \texttt{None}

\texttt{y} = \texttt{None}

\texttt{ydata} = \texttt{None}

class \texttt{matplotlib.backend_bases.NavigationToolbar2(\textit{canvas})}

Bases: \texttt{object}

Base class for the navigation cursor, version 2

backends must implement a canvas that handles connections for ‘button_press_event’ and ‘button_release_event’. See \texttt{FigureCanvasBase.mpl_connect()} for more information

They must also define

- \texttt{save_figure()} save the current figure
- \texttt{set_cursor()} if you want the pointer icon to change
- \texttt{\_init\_toolbar()} create your toolbar widget
- \texttt{draw_rubberband()} (optional) draw the zoom to rect “rubberband” rectangle
- \texttt{press()} (optional) whenever a mouse button is pressed, you’ll be notified with the event
- \texttt{release()} (optional) whenever a mouse button is released, you’ll be notified with the event
- \texttt{dynamic\_update()} (optional) dynamically update the window while navigating
- \texttt{set\_message()} (optional) display message
- \texttt{set\_history\_buttons()} (optional) you can change the history back / forward buttons to indicate disabled / enabled state.

That’s it, we’ll do the rest!

\texttt{back(*args)}

move back up the view lim stack

\texttt{drag\_pan(\textit{event})}

the drag callback in pan/zoom mode

\texttt{drag\_zoom(\textit{event})}

the drag callback in zoom mode

\texttt{draw()}

Redraw the canvases, update the locators

\texttt{draw\_rubberband(\textit{event, x0, y0, x1, y1})}

Draw a rectangle rubberband to indicate zoom limits
dynamic_update()

forward(*args)
    Move forward in the view lim stack
home(*args)
    Restore the original view
mouse_move(event)

pan(*args)
    Activate the pan/zoom tool. pan with left button, zoom with right
press(event)
    Called whenever a mouse button is pressed.
press_pan(event)
    the press mouse button in pan/zoom mode callback
press_zoom(event)
    the press mouse button in zoom to rect mode callback
push_current()
    push the current view limits and position onto the stack
release(event)
    this will be called whenever mouse button is released
release_pan(event)
    the release mouse button callback in pan/zoom mode
release_zoom(event)
    the release mouse button callback in zoom to rect mode
remove_rubberband()
    Remove the rubberband
save_figure(*args)
    Save the current figure
set_cursor(cursor)
    Set the current cursor to one of the Cursors enums values
set_history_buttons()
    Enable or disable back/forward button
set_message(s)
    Display a message on toolbar or in status bar
update()
    Reset the axes stack
**zoom**(*args*)

Activate zoom to rect mode

---

**exception** matplotlib.backend_bases.NonGuiException

Bases: Exception

class matplotlib.backend_bases.PickEvent(name, canvas, mouseevent, artist, guiEvent=None, **kwargs)

Bases: matplotlib.backend_bases.Event

A pick event, fired when the user picks a location on the canvas sufficiently close to an artist.

Attrs: all the Event attributes plus

**mouseevent** the MouseEvent that generated the pick

**artist** the Artist picked

**other** extra class dependent attrs – e.g., a Line2D pick may define different extra attributes than a PatchCollection pick event

Example usage:

```python
ax.plot(np.rand(100), 'o', picker=5)  # 5 points tolerance

def on_pick(event):
    line = event.artist
    xdata, ydata = line.get_data()
    ind = event.ind
    print('on pick line:', np.array([xdata[ind], ydata[ind]]).T)

cid = fig.canvas.mpl_connect('pick_event', on_pick)
```

---

**class** matplotlib.backend_bases.RendererBase

Bases: object

An abstract base class to handle drawing/rendering operations.

The following methods must be implemented in the backend for full functionality (though just implementing `draw_path()` alone would give a highly capable backend):

- `draw_path()`
- `draw_image()`
- `draw_gouraud_triangle()`

The following methods *should* be implemented in the backend for optimization reasons:

- `draw_text()`
- `draw_markers()`
- `draw_path_collection()`
- `draw_quad_mesh()`

**close_group**(s)

Close a grouping element with label s Is only currently used by backend_svg

---

21.1. matplotlib.backend_bases 927
**draw_gouraud_triangle**(*gc, points, colors, transform*)

Draw a Gouraud-shaded triangle.

- *points* is a 3x2 array of (x, y) points for the triangle.
- *colors* is a 3x4 array of RGBA colors for each point of the triangle.
- *transform* is an affine transform to apply to the points.

**draw_gouraud_triangles**(*gc, triangles_array, colors_array, transform*)

Draws a series of Gouraud triangles.

- *points* is a Nx3x2 array of (x, y) points for the triangle.
- *colors* is a Nx3x4 array of RGBA colors for each point of the triangle.
- *transform* is an affine transform to apply to the points.

**draw_image**(*gc, x, y, im, transform=None*)

Draw an RGBA image.

- *gc* is a GraphicsContextBase instance with clipping information.
- *x* is the distance in physical units (i.e., dots or pixels) from the left hand side of the canvas.
- *y* is the distance in physical units (i.e., dots or pixels) from the bottom side of the canvas.
- *im* is an NxMx4 array of RGBA pixels (of dtype uint8).
- *transform* If and only if the concrete backend is written such that option_scale_image() returns True, an affine transformation may be passed to draw_image(). It takes the form of a Affine2DBase instance. The translation vector of the transformation is given in physical units (i.e., dots or pixels). Note that the transformation does not override x and y, and has to be applied before translating the result by x and y (this can be accomplished by adding x and y to the translation vector defined by transform).

**draw_markers**(*gc, marker_path, marker_trans, path, trans, rgbFace=None*)

Draws a marker at each of the vertices in path. This includes all vertices, including control points on curves. To avoid that behavior, those vertices should be removed before calling this function.

- *gc* is the GraphicsContextBase instance.
- *marker_trans* is an affine transform applied to the marker.
- *trans* is an affine transform applied to the path.

This provides a fallback implementation of draw_markers that makes multiple calls to draw_path(). Some backends may want to override this method in order to draw the marker only once and reuse it multiple times.

**draw_path**(*gc, path, transform, rgbFace=None*)

Draws a Path instance using the given affine transform.

**draw_path_collection**(*gc, master_transform, paths, all_transforms, offsets, offsetTrans, facecolors, edgecolors, linewidths, linestyles, antialiaseds, urls, offset_position*)

Draws a collection of paths selecting drawing properties from the lists facecolors, edgecolors,
linewidths, linestyles and antialiaseds. offsets is a list of offsets to apply to each of the paths. The offsets in offsets are first transformed by offsetTrans before being applied. offset_position may be either “screen” or “data” depending on the space that the offsets are in.

This provides a fallback implementation of draw_path_collection() that makes multiple calls to draw_path(). Some backends may want to override this in order to render each set of path data only once, and then reference that path multiple times with the different offsets, colors, styles etc. The generator methods _iter_collection_raw_paths() and _iter_collection() are provided to help with (and standardize) the implementation across backends. It is highly recommended to use those generators, so that changes to the behavior of draw_path_collection() can be made globally.

draw_quad_mesh(gc, master_transform, meshWidth, meshHeight, coordinates, offsets, offsetTrans, facecolors, antialiased, edgecolors)
This provides a fallback implementation of draw_quad_mesh() that generates paths and then calls draw_path_collection().

draw_text(gc, x, y, s, prop, angle, ismath='TeX!', mtext=None)

def draw_text(gc, x, y, s, prop, angle, ismath=False, mtext=None)
Draw the text instance

gc the GraphicsContextBase instance

x the x location of the text in display coords

y the y location of the text baseline in display coords

s the text string

prop a matplotlib.font_manager.FontProperties instance

angle the rotation angle in degrees

mtext a matplotlib.text.Text instance

backend implementers note
When you are trying to determine if you have gotten your bounding box right (which is what enables the text layout/alignment to work properly), it helps to change the line in text.py:

```python
if 0: bbox_artist(self, renderer)
```
to if 1, and then the actual bounding box will be plotted along with your text.

flipy()
Return true if y small numbers are top for renderer Is used for drawing text(matplotlib.text) and images(matplotlib.image) only

get_canvas_width_height()
return the canvas width and height in display coords

get_image_magnification()
Get the factor by which to magnify images passed to draw_image(). Allows a backend to have images at a different resolution to other artists.
Matplotlib, Release 2.0.2

get_texmanager()
return the matplotlib.texmanager.TexManager instance

get_text_width_height_descent(s, prop, ismath)
get the width and height, and the offset from the bottom to the baseline (descent), in display
cords of the string s with FontProperties prop

new_gc()
Return an instance of a GraphicsContextBase

open_group(s, gid=None)
Open a grouping element with label s. If gid is given, use gid as the id of the group. Is only
currently used by backend_svg.

option_image_nocomposite()
override this method for renderers that do not necessarily always want to rescale and composite
raster images. (like SVG, PDF, or PS)

option_scale_image()
override this method for renderers that support arbitrary affine transformations in
draw_image() (most vector backends).

points_to_pixels(points)
Convert points to display units

points a float or a numpy array of float

return points converted to pixels

You need to override this function (unless your backend doesn’t have a dpi, e.g., postscript or
svg). Some imaging systems assume some value for pixels per inch:

points to pixels = points * pixels_per_inch/72.0 * dpi/72.0

start_filter()
Used in AggRenderer. Switch to a temporary renderer for image filtering effects.

start_rasterizing()
Used in MixedModeRenderer. Switch to the raster renderer.

stop_filter(filter_func)
Used in AggRenderer. Switch back to the original renderer. The contents of the temporary
renderer is processed with the filter_func and is drawn on the original renderer as an image.

stop_rasterizing()
Used in MixedModeRenderer. Switch back to the vector renderer and draw the contents of the
raster renderer as an image on the vector renderer.

strip_math(s)

class matplotlib.backend_bases.ResizeEvent(name, canvas)
Bases: matplotlib.backend_bases.Event

An event triggered by a canvas resize
In addition to the \textit{Event} attributes, the following event attributes are defined:

\textit{width} width of the canvas in pixels

\textit{height} height of the canvas in pixels

\texttt{class matplotlib.backend_bases.ShowBase}
Bases: object

Simple base class to generate a show() callable in backends.

Subclass must override mainloop() method.

\texttt{mainloop()}

\texttt{class matplotlib.backend_bases.StatusbarBase(toolmanager)}
Bases: object

Base class for the statusbar

\texttt{set_message(s)}
Display a message on toolbar or in status bar

\texttt{Parameters s : str}
Message text

\texttt{class matplotlib.backend_bases.TimerBase(interval=None, callbacks=None)}
Bases: object

A base class for providing timer events, useful for things animations. Backends need to implement a few specific methods in order to use their own timing mechanisms so that the timer events are integrated into their event loops.

Mandatory functions that must be implemented:

- \texttt{.timer_start}: Contains backend-specific code for starting the timer

- \texttt{.timer_stop}: Contains backend-specific code for stopping the timer

Optional overrides:

- \texttt{.timer_set_single_shot}: Code for setting the timer to single shot operating mode, if supported by the timer object. If not, the \texttt{Timer} class itself will store the flag and the \texttt{on_timer} method should be overridden to support such behavior.

- \texttt{.timer_set_interval}: Code for setting the interval on the timer, if there is a method for doing so on the timer object.

- \texttt{.on_timer}: This is the internal function that any timer object should call, which will handle the task of running all callbacks that have been set.

Attributes:

- \texttt{interval}: The time between timer events in milliseconds. Default is 1000 ms.

- \texttt{single_shot}: Boolean flag indicating whether this timer should operate as single shot (run once and then stop). Defaults to \texttt{False}. 
callbacks: Stores list of (func, args) tuples that will be called upon timer events. This list can be manipulated directly, or the functions `add_callback` and `remove_callback` can be used.

```python
add_callback(func, *args, **kwargs)
```
Register func to be called by timer when the event fires. Any additional arguments provided will be passed to func.

```python
interval
```

```python
remove_callback(func, *args, **kwargs)
```
Remove func from list of callbacks. args and kwargs are optional and used to distinguish between copies of the same function registered to be called with different arguments.

single_shot

```python
start(interval=None)
```
Start the timer object. interval is optional and will be used to reset the timer interval first if provided.

```python
stop()
```
Stop the timer.

```python
class matplotlib.backend_bases.ToolContainerBase(toolmanager)
```
Bases: object
Base class for all tool containers, e.g. toolbars.

Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>toolmanager</td>
<td>(ToolManager object that holds the tools that) this ToolContainer wants to communicate with.</td>
</tr>
</tbody>
</table>

```python
add_tool(tool, group, position=-1)
```
Adds a tool to this container

Parameters tool : tool_like

The tool to add, see ToolManager.get_tool.

```python
group : str
```

The name of the group to add this tool to.

```python
position : int (optional)
```

The position within the group to place this tool. Defaults to end.

```python
add_toolitem(name, group, position, image, description, toggle)
```
Add a toolitem to the container

This method must get implemented per backend

The callback associated with the button click event, must be EXACTLY `self.
trigger_tool(name)`
Parameters name : string
    Name of the tool to add, this gets used as the tool’s ID and as the default label
    of the buttons

group : String
    Name of the group that this tool belongs to

position : Int
    Position of the tool within its group, if -1 it goes at the End

image_file : String
    Filename of the image for the button or None

description : String
    Description of the tool, used for the tooltips

toggle : Bool
    • True : The button is a toggle (change the pressed/unpressed state between con-
    secutive clicks)
    • False : The button is a normal button (returns to unpressed state after release)

remove_toolitem(name)
    Remove a toolitem from the ToolContainer
    This method must get implemented per backend
    Called when ToolManager emits a tool_removed_event

    Parameters name : string
        Name of the tool to remove

toggle_toolitem(name, toggled)
    Toggle the toolitem without firing event

    Parameters name : String
        Id of the tool to toggle

    toggled : bool
        Whether to set this tool as toggled or not.

trigger_tool(name)
    Trigger the tool

    Parameters name : String
        Name(id) of the tool triggered from within the container

matplotlib.backend_bases.get_registered_canvas_class(format)
    Return the registered default canvas for given file format. Handles deferred import of required back-
end.
matplotlib.backend_bases.key_press_handler(event, canvas, toolbar=None)

Implement the default mpl key bindings for the canvas and toolbar described at Navigation Keyboard Shortcuts

- **event** a KeyEvent instance
- **canvas** a FigureCanvasBase instance
- **toolbar** a NavigationToolbar2 instance

matplotlib.backend_bases.register_backend(format, backend, description=None)

Register a backend for saving to a given file format.

- **format** [str] File extension
- **backend** [module string or canvas class] Backend for handling file output
- **description** [str, optional] Description of the file type. Defaults to an empty string

## 21.2 matplotlib.backend_managers

**ToolManager**  Class that makes the bridge between user interaction (key press, toolbar clicks, ..) and the actions in response to the user inputs.

### class matplotlib.backend_managers.ToolEvent(name, sender, tool, data=None)

Bases: object

Event for tool manipulation (add/remove)

### class matplotlib.backend_managers.ToolManager(canvas)

Bases: object

Helper class that groups all the user interactions for a FigureManager

#### Attributes

<table>
<thead>
<tr>
<th>manager: FigureManager</th>
<th>keypresslock: widgets.LockDraw</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LockDraw object to know if the canvas key_press_event is locked</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>active_toggle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Currently toggled tools</td>
</tr>
</tbody>
</table>

**add_tool**(name, tool, *args, **kwargs)

Add tool to ToolManager

If successful adds a new event tool_trigger_name where name is the name of the tool, this event is fired everytime the tool is triggered.

Parameters

- **name** : str
Name of the tool, treated as the ID, has to be unique

tool : class_like, i.e. str or type
    Reference to find the class of the Tool to added.

See also:

`matplotlib.backend_tools.ToolBase` The base class for tools.

Notes

args and kwars get passed directly to the tools constructor.

get_tool(name, warn=True)
    Return the tool object, also accepts the actual tool for convenience
    
    Parameters name : str, ToolBase
        Name of the tool, or the tool itself
    
    warn : bool, optional
        If this method should give warnings.

get_tool_keymap(name)
    Get the keymap associated with the specified tool
    
    Parameters name : string
        Name of the Tool

    Returns list : list of keys associated with the Tool

message_event(message, sender=None)
    Emit a ToolManagerMessageEvent

remove_tool(name)
    Remove tool from ToolManager
    
    Parameters name : string
        Name of the Tool

toolmanager_connect(s, func)
    Connect event with string s to func.
    
    Parameters s : String
        Name of the event
        The following events are recognized
        
        • 'tool_message_event'
        • 'tool_removed_event'
        • 'tool_added_event'
For every tool added a new event is created

• 'tool_trigger_TOOLNAME' Where TOOLNAME is the id of the tool.

**func**: function

Function to be called with signature def func(event)

```python
toolmanager_disconnect(cid)

Disconnect callback id cid

Example usage:

cid = toolmanager.toolmanager_connect('tool_trigger_zoom',
on_press)
#...later
toolmanager.toolmanager_disconnect(cid)
```

**tools**

Return the tools controlled by `ToolManager`

**trigger_tool**(name, sender=None, canvasevent=None, data=None)

Trigger a tool and emit the tool_trigger_[name] event

**Parameters**

- name: string
  Name of the tool
- sender: object
  Object that wishes to trigger the tool
- canvasevent: Event
  Original Canvas event or None
- data: Object
  Extra data to pass to the tool when triggering

**update_keymap**(name, *keys)

Set the keymap to associate with the specified tool

**Parameters**

- name: string
  Name of the Tool
- keys: keys to associate with the Tool

**class** `matplotlib.backend_managers.ToolManagerMessageEvent` *(name, sender, message)*

Bases: object

Event carrying messages from toolmanager

Messages usually get displayed to the user by the toolbar

**class** `matplotlib.backend_managers.ToolTriggerEvent` *(name, sender, tool, canvasevent=None, data=None)*

Bases: `matplotlib.backend_managers.ToolEvent`
Event to inform that a tool has been triggered

## 21.3 matplotlib.backend_tools

Abstract base classes define the primitives for Tools. These tools are used by `matplotlib.backend_managers.ToolManager`

**ToolBase** Simple stateless tool

**ToolToggleBase** Tool that has two states, only one Toggle tool can be active at any given time for the same `matplotlib.backend_managers.ToolManager`

```python
class matplotlib.backend_tools.AxisScaleBase(*args, **kwargs)
    Bases: matplotlib.backend_tools.ToolToggleBase
    Base Tool to toggle between linear and logarithmic

    disable(event)

    enable(event)

    trigger(sender, event, data=None)
```

```python
class matplotlib.backend_tools.ConfigureSubplotsBase(toolmanager, name)
    Bases: matplotlib.backend_tools.ToolBase
    Base tool for the configuration of subplots

    description = ‘Configure subplots’

    image = ‘subplots.png’
```

```python
class matplotlib.backend_tools.Cursors
    Bases: object
    Simple namespace for cursor reference

    HAND = 0

    MOVE = 3

    POINTER = 1

    SELECT_REGION = 2
```
class matplotlib.backend_tools.RubberbandBase(toolmanager, name)
    Bases: matplotlib.backend_tools.ToolBase
    Draw and remove rubberband
    draw_rubberband(*data)
        Draw rubberband
        This method must get implemented per backend
    remove_rubberband()
        Remove rubberband
        This method should get implemented per backend
    trigger(sender, event, data)
        Call draw_rubberband or remove_rubberband based on data

class matplotlib.backend_tools.SaveFigureBase(toolmanager, name)
    Bases: matplotlib.backend_tools.ToolBase
    Base tool for figure saving
    default_keymap = ['s', 'ctrl+s']
    description = ‘Save the figure’
    image = ‘filesave.png’

class matplotlib.backend_tools.SetCursorBase(*args, **kwargs)
    Bases: matplotlib.backend_tools.ToolBase
    Change to the current cursor while inaxes
    This tool, keeps track of all ToolToggleBase derived tools, and calls set_cursor when a tool gets triggered
    set_cursor(cursor)
        Set the cursor
        This method has to be implemented per backend

class matplotlib.backend_tools.ToolBack(toolmanager, name)
    Bases: matplotlib.backend_tools.ViewsPositionsBase
    Move back up the view lim stack
    default_keymap = ['left', 'c', 'backspace']
    description = ‘Back to previous view’
    image = ‘back.png’
class matplotlib.backend_tools.ToolBase(toolmanager, name)

Bases: object

Base tool class

A base tool, only implements trigger method or not method at all. The tool is instantiated by matplotlib.backend_managers.ToolManager

Attributes

<table>
<thead>
<tr>
<th>attribute</th>
<th>type</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>toolmanager</td>
<td>matplotlib.backend_managers.ToolManager</td>
<td>ToolManager that controls this Tool</td>
</tr>
<tr>
<td>figure</td>
<td>FigureCanvas</td>
<td>Figure instance that is affected by this Tool</td>
</tr>
<tr>
<td>name</td>
<td>String</td>
<td>Used as Id of the tool, has to be unique among tools of the same ToolManager</td>
</tr>
<tr>
<td>default_keymap</td>
<td>None</td>
<td>Keymap to associate with this tool</td>
</tr>
<tr>
<td>description</td>
<td>None</td>
<td>Description of the Tool</td>
</tr>
<tr>
<td>destroy()</td>
<td>None</td>
<td>Destroy the tool</td>
</tr>
<tr>
<td>image</td>
<td>None</td>
<td>Filename of the image</td>
</tr>
<tr>
<td>name</td>
<td>Tool Id</td>
<td></td>
</tr>
<tr>
<td>trigger</td>
<td>(sender, event, data=None)</td>
<td>Called when this tool gets used</td>
</tr>
</tbody>
</table>

**Parameters**

- **event**: ‘Event’
  
  The Canvas event that caused this tool to be called

- **sender**: object
Object that requested the tool to be triggered

**data: object**

Extra data

class matplotlib.backend_tools.ToolCursorPosition(*args, **kwargs)
   Bases: matplotlib.backend_tools.ToolBase

   Send message with the current pointer position
   This tool runs in the background reporting the position of the cursor

   **send_message**(event)
       Call matplotlib.backend_managers.ToolManager.message_event

class matplotlib.backend_tools.ToolEnableAllNavigation(toolmanager, name)
   Bases: matplotlib.backend_tools.ToolBase

   Tool to enable all axes for toolmanager interaction

   **default_keymap** = ['a']

   **description** = ‘Enables all axes toolmanager’

   **trigger**(sender, event, data=None)

class matplotlib.backend_tools.ToolEnableNavigation(toolmanager, name)
   Bases: matplotlib.backend_tools.ToolBase

   Tool to enable a specific axes for toolmanager interaction

   **default_keymap** = (1, 2, 3, 4, 5, 6, 7, 8, 9)

   **description** = ‘Enables one axes toolmanager’

   **trigger**(sender, event, data=None)

class matplotlib.backend_tools.ToolForward(toolmanager, name)
   Bases: matplotlib.backend_tools.ViewsPositionsBase

   Move forward in the view lim stack

   **default_keymap** = ['right', 'v']

   **description** = ‘Forward to next view’

   **image** = ‘forward.png’
class matplotlib.backend_tools.ToolFullScreen(*args, **kwargs)
Bases: matplotlib.backend_tools.ToolToggleBase

Tool to toggle full screen

default_keymap = ['f', 'ctrl+f']

description = ‘Toogle Fullscreen mode’

disable(event)

enable(event)

class matplotlib.backend_tools.ToolGrid(*args, **kwargs)
Bases: matplotlib.backend_tools.ToolToggleBase

Tool to toggle the grid of the figure

default_keymap = ['g']

description = ‘Toogle Grid’

disable(event)

enable(event)

trigger(sender, event, data=None)

class matplotlib.backend_tools.ToolHome(toolmanager, name)
Bases: matplotlib.backend_tools.ViewsPositionsBase

Restore the original view lim

default_keymap = ['h', 'r', 'home']

description = ‘Reset original view’

image = ‘home.png’

class matplotlib.backend_tools.ToolPan(*args)
Bases: matplotlib.backend_tools.ZoomPanBase

Pan axes with left mouse, zoom with right

cursor = 3
default_keymap = ['p']

description = 'Pan axes with left mouse, zoom with right'

image = 'move.png'

radio_group = 'default'

class matplotlib.backend_tools.ToolQuit(toolmanager, name)
    Bases: matplotlib.backend_tools.ToolBase
    Tool to call the figure manager destroy method
    default_keymap = ['ctrl+w', 'cmd+w']

description = 'Quit the figure'

trigger(sender, event, data=None)

class matplotlib.backend_tools.ToolToggleBase(*args, **kwargs)
    Bases: matplotlib.backend_tools.ToolBase
    Toggleable tool
    Every time it is triggered, it switches between enable and disable
    cursor = None
        Cursor to use when the tool is active
    disable(event=None)
        Disable the toggle tool
        trigger call this method when toggled is True.
        This can happen in different circumstances
        • Click on the toolbar tool button
        • Call to matplotlib.backend_managers.ToolManager.trigger_tool
        • Another ToolToggleBase derived tool is triggered (from the same ToolManager)
    enable(event=None)
        Enable the toggle tool
        trigger calls this method when toggled is False
    radio_group = None
        Attribute to group ‘radio’ like tools (mutually exclusive)
        String that identifies the group or None if not belonging to a group
```python
toggled
State of the toggled tool

trigger(sender, event, data=None)
Calls enable or disable based on toggled value

class matplotlib.backend_tools.ToolViewsPositions(*args, **kwargs)
Bases: matplotlib.backend_tools.ToolBase
Auxiliary Tool to handle changes in views and positions
Runs in the background and should get used by all the tools that need to access the figure’s history of views and positions, e.g.

- ToolZoom
- ToolPan
- ToolHome
- ToolBack
- ToolForward

add_figure()
Add the current figure to the stack of views and positions

back()
Back one step in the stack of views and positions

clear(figure)
Reset the axes stack

forward()
Forward one step in the stack of views and positions

home()
Recall the first view and position from the stack

push_current()
push the current view limits and position onto the stack

refresh_locators()
Redraw the canvases, update the locators

update_view()
Update the viewlim and position from the view and position stack for each axes

class matplotlib.backend_tools.ToolXScale(*args, **kwargs)
Bases: matplotlib.backend_tools.AxisScaleBase
Tool to toggle between linear and logarithmic scales on the X axis

default_keymap = ['k', 'L']

description = ‘Toggle Scale X axis’
```
Matplotlib, Release 2.0.2

```python
set_scale(ax, scale)

class matplotlib.backend_tools.ToolYScale(*args, **kwargs)
    Bases: matplotlib.backend_tools.AxisScaleBase
    Tool to toggle between linear and logarithmic scales on the Y axis
    
    default_keymap = ['l']

    description = ‘Toogle Scale Y axis’

    set_scale(ax, scale)

class matplotlib.backend_tools.ToolZoom(*args)
    Bases: matplotlib.backend_tools.ZoomPanBase
    Zoom to rectangle

    cursor = 2

    default_keymap = ['o']

    description = ‘Zoom to rectangle’

    image = ‘zoom_to_rect.png’

    radio_group = ‘default’

class matplotlib.backend_tools.ViewsPositionsBase(toolmanager, name)
    Bases: matplotlib.backend_tools.ToolBase
    Base class for ToolHome, ToolBack and ToolForward

    trigger(sender, event, data=None)

class matplotlib.backend_tools.ZoomPanBase(*args)
    Bases: matplotlib.backend_tools.ToolToggleBase
    Base class for ToolZoom and ToolPan

    disable(event)
        Release the canvas and disconnect press/release events

    enable(event)
        Connect press/release events and lock the canvas

    scroll_zoom(event)
```
trigger(sender, event, data=None)


Add multiple tools to the container.

Parameters container: Container

backend_bases.ToolContainerBase object that will get the tools added

tools : list, optional

List in the form [[group1, [tool1, tool2 ...]], [group2, [...]]] Where the tools given by tool1, and tool2 will display in group1. See add_tool for details.


Add multiple tools to ToolManager

Parameters toolmanager: ToolManager
backend_managers.ToolManager object that will get the tools added

tools : {str: class_like}, optional

The tools to add in a {name: tool} dict, see add_tool for more info.

matplotlib.backend_tools.default_toolbar_tools = [['navigation', ['home', 'back', 'forward']], ['zoompan', ['pan', 'zoom', 'subplots']], ['io', ['save']]]

Default tools in the toolbar


Default tools

21.4 matplotlib.backends.backend_gtkagg

TODO We’ll add this later, importing the gtk backends requires an active X-session, which is not compatible with cron jobs.

21.5 matplotlib.backends.backend_qt4agg

Render to qt from agg

matplotlib.backends.backend_qt4agg.FigureCanvas alias of FigureCanvasQTAgg
class matplotlib.backends.backend_qt4agg.FigureCanvasQTAgg(figure)
   Bases: matplotlib.backends.backend_qt4agg.FigureCanvasQTAggBase, matplotlib.backends.backend_qt4.FigureCanvasQT, matplotlib.backends.backend_agg.FigureCanvasAgg

The canvas the figure renders into. Calls the draw and print fig methods, creates the renderers, etc...

   Public attribute
   figure - A Figure instance
class matplotlib.backends.backend_qt4agg.FigureCanvasQTAggBase(figure)
   Bases: matplotlib.backends.backend_qt5agg.FigureCanvasQTAggBase

matplotlib.backends.backend_qt4agg.new_figure_manager(num, *args, **kwargs)
   Create a new figure manager instance

matplotlib.backends.backend_qt4agg.new_figure_manager_given_figure(num, figure)
   Create a new figure manager instance for the given figure.

21.6 matplotlib.backends.backend_qt5agg

Render to qt from agg

matplotlib.backends.backend_qt5agg.FigureCanvas alias of FigureCanvasQTAgg
class matplotlib.backends.backend_qt5agg.FigureCanvasQTAgg(figure)
    Bases: matplotlib.backends.backend_qt5agg.FigureCanvasQTAggBase, matplotlib.backends.backend_qt5.FigureCanvasQT, matplotlib.backends.backend_agg.FigureCanvasAgg

The canvas the figure renders into. Calls the draw and print fig methods, creates the renderers, etc.

Modified to import from Qt5 backend for new-style mouse events.

Public attribute
    figure - A Figure instance

class matplotlib.backends.backend_qt5agg.FigureCanvasQTAggBase(figure)
    Bases: object

The canvas the figure renders into. Calls the draw and print fig methods, creates the renderers, etc...

Public attribute
    figure - A Figure instance

blit(bbox=None)
    Blit the region in bbox

draw()
    Draw the figure with Agg, and queue a request for a Qt draw.

drawRectangle(rect)

draw_idle()
    Queue redraw of the Agg buffer and request Qt paintEvent.

paintEvent(e)
    Copy the image from the Agg canvas to the qt.drawable. In Qt, all drawing should be done inside of here when a widget is shown onscreen.

print_figure(*args, **kwargs)

matplotlib.backends.backend_qt5agg.new_figure_manager(num, *args, **kwargs)
    Create a new figure manager instance

matplotlib.backends.backend_qt5agg.new_figure_manager_given_figure(num, figure)
    Create a new figure manager instance for the given figure.

21.7 matplotlib.backends.backend_wxagg

matplotlib.backends.backend_wxagg.FigureCanvas
    alias of FigureCanvasWxAgg

class matplotlib.backends.backend_wxagg.FigureCanvasWxAgg(parent, id, figure)
    Bases: matplotlib.backends.backend_agg.FigureCanvasAgg, matplotlib.backends.backend_wx.FigureCanvasWx
The FigureCanvas contains the figure and does event handling.

In the wxPython backend, it is derived from wxPanel, and (usually) lives inside a frame instantiated by a FigureManagerWx. The parent window probably implements a wxSizer to control the displayed control size - but we give a hint as to our preferred minimum size.

Initialise a FigureWx instance.

- Initialise the FigureCanvasBase and wxPanel parents.
- Set event handlers for: EVT_SIZE (Resize event) EVT_PAINT (Paint event)

```
blit(bbox=None)
```

Transfer the region of the agg buffer defined by bbox to the display. If bbox is None, the entire buffer is transferred.

```
draw(drawDC=None)
```

Render the figure using agg.

```
filetypes = {'ps': 'Postscript', 'eps': 'Encapsulated Postscript', 'pdf': 'Portable Document Format', 'pgf': 'PGF code for LaTeX', ...
```

```
print_figure(filename, *args, **kwargs)
```

21.8 matplotlib.backends.backend_pdf

A PDF matplotlib backend Author: Jouni K Seppänen <jks@iki.fi>

```
matplotlib.backends.backend_pdf.FigureCanvas
alias of FigureCanvasPdf
```

```
class matplotlib.backends.backend_pdf.FigureCanvasPdf(figure)
Bases: matplotlib.backend_bases.FigureCanvasBase
```

The canvas the figure renders into. Calls the draw and print fig methods, creates the renderers, etc...
Public attribute

figure - A Figure instance

class matplotlib.backends.backend_pdf.Name(name)
    Bases: object
    PDF name object.

class matplotlib.backends.backend_pdf.Operator(op)
    Bases: object
    PDF operator object.

class matplotlib.backends.backend_pdf.PdfFile(filename)
    Bases: object
    PDF file object.

alphaState(alpha)
    Return name of an ExtGState that sets alpha to the given value

embedTTF(filename, characters)
    Embed the TTF font from the named file into the document.

fontName(fontprop)
    Select a font based on fontprop and return a name suitable for Op.selectfont. If fontprop is a
    string, it will be interpreted as the filename (or dvi name) of the font.

imageObject(image)
    Return name of an image XObject representing the given image.

markerObject(path, trans, fill, stroke, lw, joinstyle, capstyle)
    Return name of a marker XObject representing the given path.

reserveObject(name='')
    Reserve an ID for an indirect object. The name is used for debugging in case we forget to print
    out the object with writeObject.

writeInfoDict()
    Write out the info dictionary, checking it for good form

writeTrailer()
    Write out the PDF trailer.

writeXref()
    Write out the xref table.

class matplotlib.backends.backend_pdf.PdfPages(filename, keep_empty=True)
    Bases: object
    A multi-page PDF file.
Notes

In reality `PdfPages` is a thin wrapper around `PdfFile`, in order to avoid confusion when using `savefig()` and forgetting the format argument.

Examples

```python
>>> import matplotlib.pyplot as plt
>>> # Initialize:
>>> with PdfPages('foo.pdf') as pdf:
...     # As many times as you like, create a figure fig and save it:
...     fig = plt.figure()
...     pdf.savefig(fig)
...     # When no figure is specified the current figure is saved
...     pdf.savefig()
```

Create a new PdfPages object.

**Parameters**

- **filename**: str
  - Plots using `PdfPages.savefig()` will be written to a file at this location. The file is opened at once and any older file with the same name is overwritten.

- **keep_empty**: bool, optional
  - If set to False, then empty pdf files will be deleted automatically when closed.

- **attach_note**(text, positionRect=[-100, -100, 0, 0])
  - Add a new text note to the page to be saved next. The optional positionRect specifies the position of the new note on the page. It is outside the page per default to make sure it is invisible on printouts.

- **close()**
  - Finalize this object, making the underlying file a complete PDF file.

- **get_pagecount()**
  - Returns the current number of pages in the multipage pdf file.

- **infodict()**
  - Return a modifiable information dictionary object (see PDF reference section 10.2.1 ‘Document Information Dictionary’).

- **savefig(figure=None, **kwargs)**
  - Saves a `Figure` to this file as a new page.
  - Any other keyword arguments are passed to `savefig()`.

**Parameters**

- **figure**: :class:`~matplotlib.figure.Figure` or int, optional
  - Specifies what figure is saved to file. If not specified, the active figure is saved. If a `Figure` instance is provided, this figure is saved. If an int is specified, the figure instance to save is looked up by number.
class matplotlib.backends.backend_pdf.Reference(id)
    Bases: object
    
    PDF reference object. Use PdfFile.reserveObject() to create References.

class matplotlib.backends.backend_pdf.Stream(id, len, file, extra=None, png=None)
    Bases: object
    
    PDF stream object.
    
    This has no pdfRepr method. Instead, call begin(), then output the contents of the stream by calling
    write(), and finally call end().

    id: object id of stream; len: an unused Reference object for the length of the stream, or None (to use
    a memory buffer); file: a PdfFile; extra: a dictionary of extra key-value pairs to include in the stream
    header; png: if the data is already png compressed, the decode parameters

    end()
        Finalize stream.

    write(data)
        Write some data on the stream.

class matplotlib.backends.backend_pdf.Verbam(x)
    Bases: object
    
    Store verbatim PDF command content for later inclusion in the stream.

matplotlib.backends.backend_pdf.fill(strings, linelen=75)
    Make one string from sequence of strings, with whitespace in between. The whitespace is chosen to
    form lines of at most linelen characters, if possible.

matplotlib.backends.backend_pdf.new_figure_manager(num, *args, **kwargs)
    Create a new figure manager instance

matplotlib.backends.backend_pdf.new_figure_manager_given_figure(num, figure)
    Create a new figure manager instance for the given figure.

matplotlib.backends.backend_pdf.pdfRepr(obj)
    Map Python objects to PDF syntax.

21.9 matplotlib.backends.backend_svg

matplotlib.backends.backend_svg.new_figure_manager_given_figure(num, figure)
    Create a new figure manager instance for the given figure.

matplotlib.backends.backend_svg.short_float_fmt(x)
    Create a short string representation of a float, which is %f formatting with trailing zeros and the
decimal point removed.
22.1 matplotlib.cbook

A collection of utility functions and classes. Originally, many (but not all) were from the Python Cookbook – hence the name cbook.

This module is safe to import from anywhere within matplotlib; it imports matplotlib only at runtime.

```python
class matplotlib.cbook.Bunch(**kwds)
    Bases: object

    Often we want to just collect a bunch of stuff together, naming each item of the bunch; a dictionary’s OK for that, but a small do-nothing class is even handier, and prettier to use. Whenever you want to group a few variables:

    >>> point = Bunch(datum=2, squared=4, coord=12)
    >>> point.datum

    By: Alex Martelli
    From: https://code.activestate.com/recipes/121294/
```

```python
class matplotlib.cbook.CallbackRegistry
    Bases: object

    Handle registering and disconnecting for a set of signals and callbacks:

    >>> from matplotlib.cbook import CallbackRegistry
    >>> callbacks = CallbackRegistry()

    >>> id_eat = callbacks.connect('eat', oneat)
    >>> id_drink = callbacks.connect('drink', ondrink)

    >>> callbacks.process('drink', 123)
    drink 123
```
In practice, one should always disconnect all callbacks when they are no longer needed to avoid dangling references (and thus memory leaks). However, real code in matplotlib rarely does so, and due to its design, it is rather difficult to place this kind of code. To get around this, and prevent this class of memory leaks, we instead store weak references to bound methods only, so when the destination object needs to die, the CallbackRegistry won’t keep it alive. The Python stdlib weakref module can not create weak references to bound methods directly, so we need to create a proxy object to handle weak references to bound methods (or regular free functions). This technique was shared by Peter Parente on his “Mindtrove” blog.

`connect(s, func)`

register `func` to be called when a signal `s` is generated `func` will be called

`disconnect(cid)`

disconnect the callback registered with callback id `cid`

`process(s, *args, **kwargs)`

process signal `s`. All of the functions registered to receive callbacks on `s` will be called with `*args` and `**kwargs`

```python
>>> from matplotlib.cbook import Grouper
>>> class Foo(object):
...     def __init__(self, s):
...         self.s = s
...     def __repr__(self):
...         return self.s
... >>>
>>> a, b, c, d, e, f = [Foo(x) for x in 'abcdef']
>>> grp = Grouper()
>>> grp.join(a, b)
>>> grp.join(b, c)
>>> sorted(map(tuple, grp))
[(<__main__.Foo object at 0x7f5d5e532b30>, <__main__.Foo object at 0x7f5d5e532b30>), (<__main__.Foo object at 0x7f5d5e532b30>, <__main__.Foo object at 0x7f5d5e532b40>), (<__main__.Foo object at 0x7f5d5e532b40>, <__main__.Foo object at 0x7f5d5e532b50>)]
```
((a, b, c), (d, e))

>>> grp.joined(a, b)
True

>>> grp.joined(a, c)
True

>>> grp.joined(a, d)
False

clean()

Clean dead weak references from the dictionary

get_siblings(a)

Returns all of the items joined with a, including itself.

join(a, *args)

Join given arguments into the same set. Accepts one or more arguments.

joined(a, b)

Returns True if a and b are members of the same set.

remove(a)

exception matplotlib.cbook.IgnoredKeywordWarning

Bases: UserWarning

A class for issuing warnings about keyword arguments that will be ignored by matplotlib

class matplotlib.cbook.Locked(path)

Bases: object

Context manager to handle locks.
Based on code from conda.

(c) 2012-2013 Continuum Analytics, Inc. / https://www.continuum.io/ All Rights Reserved
conda is distributed under the terms of the BSD 3-clause license. Consult LICENSE_CONDA or
https://opensource.org/licenses/BSD-3-Clause.

LOCKFN = ‘.matplotlib_lock’

exception TimeoutError

Bases: RuntimeError

exception matplotlib.cbook MatplotlibDeprecationWarning

Bases: UserWarning

A class for issuing deprecation warnings for Matplotlib users.

In light of the fact that Python builtin DeprecationWarnings are ignored by default as of Python 2.7
(see link below), this class was put in to allow for the signaling of deprecation, but via UserWarnings
which are not ignored by default.

https://docs.python.org/3.7.0/whatsnew/2.7.html#the-future-for-python-2-x
class matplotlib.cbook.MemoryMonitor(nmax=20000)
    Bases: object
    
    clear()

    plot(i0=0, isub=1, fig=None)

    report(segments=4)

    xy(i0=0, isub=1)

class matplotlib.cbook.Null(*args, **kwargs)
    Bases: object
    
    Null objects always and reliably “do nothing.”

class matplotlib.cbook.RingBuffer(size_max)
    Bases: object
    
    class that implements a not-yet-full buffer

    append(x)
        append an element at the end of the buffer

    get()
        Return a list of elements from the oldest to the newest.

class matplotlib.cbook.Sorter
    Bases: object
    
    Sort by attribute or item

    Example usage:

    sort = Sorter()
    list = [(1, 2), (4, 8), (0, 3)]
    dict = [{'a': 3, 'b': 4}, {'a': 5, 'b': 2}, {'a': 0, 'b': 0},
            {'a': 9, 'b': 9}]
    sort(list) # default sort
    sort(list, 1) # sort by index 1
    sort(dict, 'a') # sort a list of dicts by key 'a'

    byAttribute(data, attributename, inplace=1)

    byItem(data, itemindex=None, inplace=1)

    sort(data, itemindex=None, inplace=1)
class matplotlib.cbook.Stack(default=None)
    Bases: object

    Implement a stack where elements can be pushed on and you can move back and forth. But no pop. Should mimic home / back / forward in a browser

    back()
        move the position back and return the current element

    bubble(o)
        raise o to the top of the stack and return o. o must be in the stack

    clear()
        empty the stack

    empty()

    forward()
        move the position forward and return the current element

    home()
        push the first element onto the top of the stack

    push(o)
        push object onto stack at current position - all elements occurring later than the current position are discarded

    remove(o)
        remove element o from the stack

class matplotlib.cbook.Xlator
    Bases: dict

    All-in-one multiple-string-substitution class

    Example usage:

    text = "Larry Wall is the creator of Perl"
adict = {
    "Larry Wall" : "Guido van Rossum",
    "creator" : "Benevolent Dictator for Life",
    "Perl" : "Python",
}

    print(multiple_replace(adict, text))

    xlat = Xlator(adict)
    print(xlat.xlat(text))

    xlat(text)
        Translate text, returns the modified text.
**(func(row), [rows from all iterators matching func(row)])**

It is used by *matplotlib.mlab.recs_join()* to join record arrays.

```python
def allequal(seq):
    return True if all elements of seq compare equal. If seq is 0 or 1 length, return True
```

```python
def allpairs(x):
    return all possible pairs in sequence x

Condensed by Alex Martelli from this thread on c.l.python
```

```python
def alltrue(seq):
    return True if all elements of seq evaluate to True. If seq is empty, return False.
```

```python
def boxplot_stats(X, whis=1.5, bootstrap=None, labels=None, autorange=False):
    Returns list of dictionaries of statistics used to draw a series of box and whisker plots. The Returns section enumerates the required keys of the dictionary. Users can skip this function and pass a user-defined set of dictionaries to the new axes.bxp method instead of relying on MPL to do the calculations.

**Parameters**

- **X**: array-like
  Data that will be represented in the boxplots. Should have 2 or fewer dimensions.

- **whis**: float, string, or sequence (default = 1.5)
  As a float, determines the reach of the whiskers to the beyond the first and third quartiles. In other words, where IQR is the interquartile range (Q3-Q1), the upper whisker will extend to last datum less than Q3 + whis*IQR). Similarly, the lower whisker will extend to the first datum greater than Q1 - whis*IQR. Beyond the whiskers, data are considered outliers and are plotted as individual points. This can be set this to an ascending sequence of percentile (e.g., [5, 95]) to set the whiskers at specific percentiles of the data. Finally, whis can be the string 'range' to force the whiskers to the minimum and maximum of the data. In the edge case that the 25th and 75th percentiles are equivalent, whis can be automatically set to 'range' via the autorange option.

- **bootstrap**: int, optional
  Number of times the confidence intervals around the median should be bootstrapped (percentile method).

- **labels**: array-like, optional
  Labels for each dataset. Length must be compatible with dimensions of X.

- **autorange**: bool, optional (False)
  When True and the data are distributed such that the 25th and 75th percentiles are equal, whis is set to 'range' such that the whisker ends are at the minimum and maximum of the data.
**Returns bxpstats**: list of dict

A list of dictionaries containing the results for each column of data. Keys of each dictionary are the following:

<table>
<thead>
<tr>
<th>Key</th>
<th>Value Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>label</td>
<td>tick label for the boxplot</td>
</tr>
<tr>
<td>mean</td>
<td>arithmetic mean value</td>
</tr>
<tr>
<td>med</td>
<td>50th percentile</td>
</tr>
<tr>
<td>q1</td>
<td>first quartile (25th percentile)</td>
</tr>
<tr>
<td>q3</td>
<td>third quartile (75th percentile)</td>
</tr>
<tr>
<td>cilo</td>
<td>lower notch around the median</td>
</tr>
<tr>
<td>cihi</td>
<td>upper notch around the median</td>
</tr>
<tr>
<td>whislo</td>
<td>end of the lower whisker</td>
</tr>
<tr>
<td>whishi</td>
<td>end of the upper whisker</td>
</tr>
<tr>
<td>fliers</td>
<td>outliers</td>
</tr>
</tbody>
</table>

**Notes**

Non-bootstrapping approach to confidence interval uses Gaussian-based asymptotic approximation:

\[
\text{med} \pm 1.57 \times \frac{iqr}{\sqrt{N}}
\]  

(22.1)


**class matplotlib.cbook.converter**(missing=’Null’, missingval=None)

Bases: object

Base class for handling string -> python type with support for missing values

**is_missing**(s)

**matplotlib.cbook.dedent**(s)

Remove excess indentation from docstring s.

Discards any leading blank lines, then removes up to n whitespace characters from each line, where n is the number of leading whitespace characters in the first line. It differs from textwrap.dedent in its deletion of leading blank lines and its use of the first non-blank line to determine the indentation.

It is also faster in most cases.

**matplotlib.cbook.delete_masked_points**(args)

Find all masked and/or non-finite points in a set of arguments, and return the arguments with only the unmasked points remaining.

Arguments can be in any of 5 categories:

1. 1-D masked arrays
2. 1-D ndarrays
3. ndarrays with more than one dimension
4. other non-string iterables
5. anything else

The first argument must be in one of the first four categories; any argument with a length differing from that of the first argument (and hence anything in category 5) then will be passed through unchanged.

Masks are obtained from all arguments of the correct length in categories 1, 2, and 4; a point is bad if masked in a masked array or if it is a nan or inf. No attempt is made to extract a mask from categories 2, 3, and 4 if np.isfinite() does not yield a Boolean array.

All input arguments that are not passed unchanged are returned as ndarrays after removing the points or rows corresponding to masks in any of the arguments.

A vastly simpler version of this function was originally written as a helper for Axes.scatter().

**matplotlib.cbook.deprecated** *(since, message='', name='', alternative='', pending=False, obj_type='function')*

Decorator to mark a function as deprecated.

**Parameters**

*since*: str

The release at which this API became deprecated. This is required.

*message*: str, optional

Override the default deprecation message. The format specifier %s may be used for the name of the function, and %s may be used in the deprecation message to insert the name of an alternative to the deprecated function. %s may be used to insert a friendly name for the type of object being deprecated.

*name*: str, optional

The name of the deprecated function; if not provided the name is automatically determined from the passed in function, though this is useful in the case of renamed functions, where the new function is just assigned to the name of the deprecated function. For example:

```python
def new_function():
    ...
oldFunction = new_function
```

*alternative*: str, optional

An alternative function that the user may use in place of the deprecated function. The deprecation warning will tell the user about this alternative if provided.

*pending*: bool, optional

If True, uses a PendingDeprecationWarning instead of a DeprecationWarning.
Examples

Basic example:

```python
@deprecated('1.4.0')
def the_function_to_deprecate():
    pass
```

```python
matplotlib.cbook.dict_delall(d, keys)
delete all of the keys from the dict d
```

```python
matplotlib.cbook.exception_to_str(s=None)
```

```python
matplotlib.cbook.file_requires_unicode(x)
Returns True if the given writable file-like object requires Unicode to be written to it.
```

```python
matplotlib.cbook.finddir(o, match, case=False)
return all attributes of o which match string in match. if case is True require an exact case match.
```

```python
matplotlib.cbook.flatten(seq, scalarp=<function is_scalar_or_string>)
Returns a generator of flattened nested containers

For example:

```python
>>> from matplotlib.cbook import flatten
>>> l = (('John', ['Hunter']), (1, 23), [[[42, (5, 23), ]]])
>>> print(list(flatten(l)))
['John', 'Hunter', 1, 23, 42, 5, 23]
```

By: Composite of Holger Krekel and Luther Blissett From: https://code.activestate.com/recipes/121294/ and Recipe 1.12 in cookbook

```python
matplotlib.cbook.get_label(y, default_name)
```

```python
matplotlib.cbook.get_recursive_filelist(args)
Recurse all the files and dirs in args ignoring symbolic links and return the files as a list of strings
```

```python
matplotlib.cbook.get_sample_data(fname, asfileobj=True)
Return a sample data file. fname is a path relative to the mpl-data/sample_data directory. If asfileobj is True return a file object, otherwise just a file path.

Set the rc parameter examples.directory to the directory where we should look, if sample_data files are stored in a location different than default (which is ‘mpl-data/sample_data’ at the same level of ‘matplotlib’ Python module files).

If the filename ends in .gz, the file is implicitly ungzipped.
```

```python
matplotlib.cbook.get_split_ind(seq, N)
seq is a list of words. Return the index into seq such that:
```

```python
len(''.join(seq[:ind]))<=N
```
matplotlib.cbook.index_of(y)
A helper function to get the index of an input to plot against if x values are not explicitly given.
Tries to get y.index (works if this is a pd.Series), if that fails, return np.arange(y.shape[0]).
This will be extended in the future to deal with more types of labeled data.

**Parameters**

- **y**: scalar or array-like
  
  The proposed y-value

**Returns**

- **x, y**: ndarray
  
  The x and y values to plot.

matplotlib.cbook.is_hashable(obj)
Returns true if obj can be hashed

matplotlib.cbook.is_math_text(s)

matplotlib.cbook.is_numlike(obj)
return true if obj looks like a number

matplotlib.cbook.is_scalar(obj)
return true if obj is not string like and is not iterable

matplotlib.cbook.is_scalar_or_string(val)
Return whether the given object is a scalar or string like.

matplotlib.cbook.is_sequence_of_strings(obj)
Returns true if obj is iterable and contains strings

matplotlib.cbook.is_string_like(obj)
Return True if obj looks like a string

matplotlib.cbook.is_writable_file_like(obj)
return true if obj looks like a file object with a write method

matplotlib.cbook.issubclass_safe(x, klass)
return issubclass(x, klass) and return False on a TypeError

matplotlib.cbook.iterable(obj)
return true if obj is iterable

matplotlib.cbook.listFiles(root, patterns='*', recurse=1, return_folders=0)
Recursively list files
from Parmar and Martelli in the Python Cookbook

matplotlib.cbook.local_over_kwdict(local_var, kwargs, *keys)
Enforces the priority of a local variable over potentially conflicting argument(s) from a kwargs dict.
The following possible output values are considered in order of priority:

- local_var > kwargs[keys[0]] > ... > kwargs[keys[-1]]
The first of these whose value is not None will be returned. If all are None then None will be returned. Each key in keys will be removed from the kwargs dict in place.

**Parameters**

- `local_var`: any object
  
  The local variable (highest priority)

- `kwargs`: dict
  
  Dictionary of keyword arguments; modified in place

- `keys`: str(s)
  
  Name(s) of keyword arguments to process, in descending order of priority

**Returns**

- `out`: any object
  
  Either local_var or one of kwargs[key] for key in keys

**Raises** IgnoredKeywordWarning

For each key in keys that is removed from kwargs but not used as the output value

**class** matplotlib.cbook.maxdict(maxsize)

- `Bases`: dict
  
  A dictionary with a maximum size; this doesn’t override all the relevant methods to constrain the size, just setitem, so use with caution

**matplotlib.cbook.mkdirs(newdir, mode=511)**

- `make directory newdir recursively, and set mode. Equivalent to`

```
> mkdir -p NEWDIR
> chmod MODE NEWDIR
```

**matplotlib.cbook.mplDeprecation**

- `alias of MatplotlibDeprecationWarning`

**matplotlib.cbook.normalize_kwargs(kw, alias_mapping=None, required=(), forbidden=(), allowed=None)**

- `Helper function to normalize kwarg inputs`

  The order they are resolved are:

  1. aliasing
  2. required
  3. forbidden
  4. allowed

  This order means that only the canonical names need appear in allowed, forbidden, required

- **Parameters**
  
  - `alias_mapping`, dict, optional
    
    A mapping between a canonical name to a list of aliases, in order of precedence from lowest to highest.

    If the canonical value is not in the list it is assumed to have the highest priority.
required : iterable, optional
   A tuple of fields that must be in kwargs.

forbidden : iterable, optional
   A list of keys which may not be in kwargs

allowed : tuple, optional
   A tuple of allowed fields. If this not None, then raise if kw contains any keys
   not in the union of required and allowed. To allow only the required fields
   pass in () for allowed

Raises TypeError
   To match what python raises if invalid args/kwargs are passed to a callable.

matplotlib.cbook.onetrue(seq)
   Return True if one element of seq is True. It seq is empty, return False.

matplotlib.cbook.pieces(seq, num=2)
   Break up the seq into num tuples

matplotlib.cbook.popall(seq)
   empty a list

matplotlib.cbook.print_cycles(objects, outstream=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>, show_progress=False)

   objects A list of objects to find cycles in. It is often useful to pass in gc.garbage to find the cycles that
   are preventing some objects from being garbage collected.

   outstream The stream for output.

   show_progress If True, print the number of objects reached as they are found.

matplotlib.cbook pts_to_midstep(x, *args)
   Covert continuous line to pre-steps

   Given a set of N points convert to 2N -1 points which when connected linearly give a step function
   which changes values at the begining the intervals.

   Parameters x : array
      The x location of the steps

   y1, y2, ... : array
      Any number of y arrays to be turned into steps. All must be the same length
      as x

   Returns x, y1, y2, .. : array
      The x and y values converted to steps in the same order as the input. If the
      input is length N, each of these arrays will be length 2N + 1
Examples

```python
g >> x_s, y1_s, y2_s = pts_to_prestep(x, y1, y2)
```

`matplotlib.cbook.pts_to_poststep(x, *args)`
Convert continuous line to pre-steps

Given a set of N points convert to 2N -1 points which when connected linearly give a step function which changes values at the begining the intervals.

**Parameters**
- `x` : array
  The x location of the steps
- `y1, y2, ...` : array
  Any number of y arrays to be turned into steps. All must be the same length as `x`

**Returns**
- `x, y1, y2, ..` : array
  The x and y values converted to steps in the same order as the input. If the input is length N, each of these arrays will be length 2N + 1

Examples

```python
g >> x_s, y1_s, y2_s = pts_to_prestep(x, y1, y2)
```

`matplotlib.cbook.pts_to_prestep(x, *args)`
Convert continuous line to pre-steps

Given a set of N points convert to 2N -1 points which when connected linearly give a step function which changes values at the begining the intervals.

**Parameters**
- `x` : array
  The x location of the steps
- `y1, y2, ...` : array
  Any number of y arrays to be turned into steps. All must be the same length as `x`

**Returns**
- `x, y1, y2, ..` : array
  The x and y values converted to steps in the same order as the input. If the input is length N, each of these arrays will be length 2N + 1

Examples

```python
g >> x_s, y1_s, y2_s = pts_to_prestep(x, y1, y2)
```

`matplotlib.cbook.recursive_remove(path)`
matplotlib.cbook.report_memory(i=0)
    return the memory consumed by process

matplotlib.cbook.restrict_dict(d, keys)
    Return a dictionary that contains those keys that appear in both d and keys, with values from d.

matplotlib.cbook.reverse_dict(d)
    reverse the dictionary – may lose data if values are not unique!

matplotlib.cbook.safe_first_element(obj)

matplotlib.cbook.safe_masked_invalid(x, copy=False)

matplotlib.cbook.safezip(*args)
    make sure args are equal len before zipping

class matplotlib.cbook.silent_list(type, seq=None)
    Bases: list
    override repr when returning a list of matplotlib artists to prevent long, meaningless output. This is meant to be used for a homogeneous list of a given type

matplotlib.cbook.simple_linear_interpolation(a, steps)

matplotlib.cbook.soundex(name, len=4)
    soundex module conforming to Odell-Russell algorithm

matplotlib.cbook.strip_math(s)
    remove latex formatting from mathtext

matplotlib.cbook.to_filehandle(fname, flag='rU', return_opened=False)
    fname can be a filename or a file handle. Support for gzipped files is automatic, if the filename ends in .gz. flag is a read/write flag for file()

class matplotlib.cbook.todate(fmt='%Y-%m-%d', missing='Null', missingval=None)
    Bases: matplotlib.cbook.converter
    convert to a date or None
    use a time.strptime() format string for conversion

class matplotlib.cbook.todatetime(fmt='%Y-%m-%d', missing='Null', missingval=None)
    Bases: matplotlib.cbook.converter
    convert to a datetime or None
    use a time.strptime() format string for conversion

class matplotlib.cbook.tofloat(missing='Null', missingval=None)
    Bases: matplotlib.cbook.converter
    convert to a float or None

class matplotlib.cbook.toint(missing='Null', missingval=None)
    Bases: matplotlib.cbook.converter

convert to an int or None

```python
class matplotlib.cbook.tostr(missing='Null', missingval='')
    Bases: matplotlib.cbook.converter

convert to string or None
```

```python
matplotlib.cbook.unicode_safe(s)
```

```python
matplotlib.cbook.unique(x)
    Return a list of unique elements of x
```

```python
matplotlib.cbook.unmasked_index_ranges(mask, compressed=True)
    Find index ranges where mask is False.

    mask will be flattened if it is not already 1-D.

    Returns Nx2 numpy.ndarray with each row the start and stop indices for slices of the compressed
    numpy.ndarray corresponding to each of N uninterrupted runs of unmasked values. If optional
    argument compressed is False, it returns the start and stop indices into the original numpy.ndarray,
    not the compressed numpy.ndarray. Returns None if there are no unmasked values.

    Example:

    ```python
    y = ma.array(np.arange(5), mask = [0,0,1,0,0])
    ii = unmasked_index_ranges(ma.getmaskarray(y))
    # returns array [[0,2], [2,4]]
    y.compressed()[ii[1,0]:ii[1,1]]
    # returns array [3,4]
    ii = unmasked_index_ranges(ma.getmaskarray(y), compressed=False)
    # returns array [[0, 2], [3, 5]]
    y.filled()[ii[1,0]:ii[1,1]]
    # returns array [3,4]
    ```
```

Prior to the transforms refactoring, this was used to support masked arrays in Line2D.

```python
matplotlib.cbook.violin_stats(X, method, points=100)
    Returns a list of dictionaries of data which can be used to draw a series of violin plots. See the
    Returns section below to view the required keys of the dictionary. Users can skip this function and
    pass a user-defined set of dictionaries to the axes.vplot method instead of using MPL to do the
    calculations.
```

**Parameters**

- **X**: array-like
  
  Sample data that will be used to produce the gaussian kernel density estimates. Must have 2 or fewer dimensions.

- **method**: callable
  
  The method used to calculate the kernel density estimate for each column of data. When called via method(v, coords), it should return a vector of the values of the KDE evaluated at the values specified in coords.
points : scalar, default = 100

Defines the number of points to evaluate each of the gaussian kernel density estimates at.

Returns A list of dictionaries containing the results for each column of data.

The dictionaries contain at least the following:

• coords: A list of scalars containing the coordinates this particular kernel density estimate was evaluated at.
• vals: A list of scalars containing the values of the kernel density estimate at each of the coordinates given in coords.
• mean: The mean value for this column of data.
• median: The median value for this column of data.
• min: The minimum value for this column of data.
• max: The maximum value for this column of data.

matplotlib.cbook.warn_deprecated(since, message='', name='', alternative='', pending=False, obj_type='attribute')

Used to display deprecation warning in a standard way.

Parameters since : str

The release at which this API became deprecated.

message : str, optional

Override the default deprecation message. The format specifier %(func)s may be used for the name of the function, and %(alternative)s may be used in the deprecation message to insert the name of an alternative to the deprecated function. %(obj_type) may be used to insert a friendly name for the type of object being deprecated.

name : str, optional

The name of the deprecated function; if not provided the name is automatically determined from the passed in function, though this is useful in the case of renamed functions, where the new function is just assigned to the name of the deprecated function. For example:

```python
def new_function():
    ...
oldFunction = new_function
```

alternative : str, optional

An alternative function that the user may use in place of the deprecated function. The deprecation warning will tell the user about this alternative if provided.

pending : bool, optional
If True, uses a PendingDeprecationWarning instead of a DeprecationWarning.

**obj_type** : str, optional

The object type being deprecated.

**Examples**

Basic example:

```python
# To warn of the deprecation of "matplotlib.name_of_module"
warn_deprecated('1.4.0', name='matplotlib.name_of_module',
                obj_type='module')
```

`matplotlib.cbook.wrap(prefix, text, cols)`

wrap text with `prefix` at length `cols`
23.1 *matplotlib.cm*

This module provides a large set of colormaps, functions for registering new colormaps and for getting a colormap by name, and a mixin class for adding color mapping functionality.

```python
class matplotlib.cm.ScalarMappable(n=norm=None, cmap=None):
    Bases: object

    This is a mixin class to support scalar data to RGBA mapping. The ScalarMappable makes use of data normalization before returning RGBA colors from the given colormap.

    Parameters:
    norm : matplotlib.colors.Normalize instance
        The normalizing object which scales data, typically into the interval [0, 1]. If None, norm defaults to a colors.Normalize object which initializes its scaling based on the first data processed.

    cmap : str or Colormap instance
        The colormap used to map normalized data values to RGBA colors.

    add_checker(checker)
        Add an entry to a dictionary of boolean flags that are set to True when the mappable is changed.

    autoscale()
        Autoscale the scalar limits on the norm instance using the current array.

    autoscale_None()
        Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None.

    changed()
        Call this whenever the mappable is changed to notify all the callback listeners to the 'changed' signal.

    check_update(checker)
        If mappable has changed since the last check, return True; else return False.

    cmap = None
        The Colormap instance of this ScalarMappable.
```
colorbar = None
   The last colorbar associated with this ScalarMappable. May be None.

get_array()
   Return the array

get_clim()
   return the min, max of the color limits for image scaling

get_cmap()
   return the colormap

norm = None
   The Normalization instance of this ScalarMappable.

set_array(A)
   Set the image array from numpy array A

set_clim(vmin=None, vmax=None)
   set the norm limits for image scaling; if vmin is a length2 sequence, interpret it as (vmin, vmax) which is used to support setp

   ACCEPTS: a length 2 sequence of floats

set_cmap(cmap)
   set the colormap for luminance data

   ACCEPTS: a colormap or registered colormap name

set_norm(norm)
   set the normalization instance

to_rgba(x, alpha=None, bytes=False, norm=True)
   Return a normalized rgba array corresponding to x.

   In the normal case, x is a 1-D or 2-D sequence of scalars, and the corresponding ndarray of rgba values will be returned, based on the norm and colormap set for this ScalarMappable.

   There is one special case, for handling images that are already rgb or rgba, such as might have been read from an image file. If x is an ndarray with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an rgb or rgba array, and no mapping will be done. If the last dimension is 3, the alpha kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the alpha kwarg is ignored; it does not replace the pre-existing alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

   In either case, if bytes is False (default), the rgba array will be floats in the 0-1 range; if it is True, the returned rgba array will be uint8 in the 0 to 255 range.

   If norm is False, no normalization of the input data is performed, and it is assumed to already be in the range (0-1).

   Note: this method assumes the input is well-behaved; it does not check for anomalies such as x being a masked rgba array, or being an integer type other than uint8, or being a floating point rgba array with values outside the 0-1 range.
matplotlib.cm.get_cmap(name=None, lut=None)

Get a colormap instance, defaulting to rc values if name is None.

Colormaps added with register_cmap() take precedence over built-in colormaps.

If name is a matplotlib.colors.Colormap instance, it will be returned.

If lut is not None it must be an integer giving the number of entries desired in the lookup table, and name must be a standard mpl colormap name.

matplotlib.cm.register_cmap(name=None, cmap=None, data=None, lut=None)

Add a colormap to the set recognized by get_cmap().

It can be used in two ways:

```
register_cmap(name='swirly', cmap=swirly_cmap)
register_cmap(name='choppy', data=choppydata, lut=128)
```

In the first case, cmap must be a matplotlib.colors.Colormap instance. The name is optional; if absent, the name will be the name attribute of the cmap.

In the second case, the three arguments are passed to the LinearSegmentedColormap initializer, and the resulting colormap is registered.

matplotlib.cm.revcmap(data)

Can only handle specification data in dictionary format.
24.1 matplotlib.collections

Classes for the efficient drawing of large collections of objects that share most properties, e.g., a large number of line segments or polygons.

The classes are not meant to be as flexible as their single element counterparts (e.g., you may not be able to select all line styles) but they are meant to be fast for common use cases (e.g., a large set of solid line segments).

class matplotlib.collections.AsteriskPolygonCollection(numsides, rotation=0, sizes=(1,), **kwargs)

Bases: matplotlib.collections.RegularPolyCollection

Draw a collection of regular asterisks with \texttt{numsides} points.

\texttt{numsides} the number of sides of the polygon

\texttt{rotation} the rotation of the polygon in radians

\texttt{sizes} gives the area of the circle circumscribing the regular polygon in points^2

Valid Collection keyword arguments:
Matplotlib, Release 2.0.2

- **edgecolors**: None
- **facecolors**: None
- **linewidths**: None
- **antialiaseds**: None
- **offsets**: None
- **transOffset**: transforms.IdentityTransform()
- **norm**: None (optional for `matplotlib.cm.ScalarMappable`)
- **cmap**: None (optional for `matplotlib.cm.ScalarMappable`)

*offsets* and *transOffset* are used to translate the patch after rendering (default no offsets)

If any of *edgecolors*, *facecolors*, *linewidths*, *antialiaseds* are None, they default to their `matplotlib.rcParams` patch setting, in sequence form.

Example: see examples/dynamic_collection.py for complete example:

```python
offsets = np.random.rand(20,2)
facecolors = [cm.jet(x) for x in np.random.rand(20)]
black = (0,0,0,1)

collection = RegularPolyCollection(
    numsides=5, # a pentagon
    rotation=0, sizes=(50,),
    facecolors = facecolors,
    edgecolors = (black,),
    linewidths = (1,),
    offsets = offsets,
    transOffset = ax.transData,
)
```

**add_callback** *(func)*

Adds a callback function that will be called whenever one of the Artist’s properties changes.

Returns an *id* that is useful for removing the callback with *remove_callback()* later.

**add_checker** *(checker)*

Add an entry to a dictionary of boolean flags that are set to True when the mappable is changed.

**aname** = ‘Artist’

**autoscale** ()

Autoscale the scalar limits on the norm instance using the current array

**autoscale_None** ()

Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

**axes**

The *Axes* instance the artist resides in, or *None*. 
changed()  
Call this whenever the mappable is changed to notify all the callbacks listeners to the  
‘changed’ signal
check_update(checker)  
If mappable has changed since the last check, return True; else return False
contains(mouseevent)  
Test whether the mouse event occurred in the collection.  
Returns True | False, dict(ind=itemlist), where every item in itemlist contains the event.
convert_xunits(x)  
For artists in an axes, if the xaxis has units support, convert x using xaxis unit type
convert_yunits(y)  
For artists in an axes, if the yaxis has units support, convert y using yaxis unit type
draw(artist, renderer, *args, **kwargs)
findobj(match=None, include_self=True)  
Find artist objects.  
Recursively find all Artist instances contained in self.  
match can be
• None: return all objects contained in artist.
• function with signature boolean = match(artist) used to filter matches
• class instance: e.g., Line2D. Only return artists of class type.
If include_self is True (default), include self in the list to be checked for a match.
format_cursor_data(data)  
Return cursor data string formatted.
get_agg_filter()  
return filter function to be used for agg filter
get_alpha()  
Return the alpha value used for blending - not supported on all backends
get_animated()  
Return the artist’s animated state
get_array()  
Return the array
get_axes()  
Return the Axes instance the artist resides in, or None.  
This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.
get_children()  
Return a list of the child Artist’s this :class:`Artist` contains.
get_clim()
return the min, max of the color limits for image scaling

g_get_clip_box()
Return artist clipbox

get_clip_on()
Return whether artist uses clipping

get_clip_path()
Return artist clip path

get_cmap()
return the colormap

get_contains()
Return the _contains test used by the artist, or None for default.

get_cursor_data(event)
Get the cursor data for a given event.

get_dashes()


g_get_datalim(transData)

get_edgecolor()

get_edgecolors()

get_facecolor()

get_facecolors()

g_get_figure()
Return the Figure instance the artist belongs to.

get_fill()
return whether fill is set

g_get_gid()
Returns the group id

get_hatch()
Return the current hatching pattern

get_label()
Get the label used for this artist in the legend.

get_linestyle()
get_linestyles()

get_linewidth()

get_linewidths()

get_numsides()

get_offset_position()
    Returns how offsets are applied for the collection. If offset_position is ‘screen’, the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is ‘data’, the offset is applied before the master transform, i.e., the offsets are in data coordinates.

get_offset_transform()

get_offsets()
    Return the offsets for the collection.

get_path_effects()

get_paths()

get_picker()
    Return the picker object used by this artist

get_pickradius()

get_rasterized()
    return True if the artist is to be rasterized

get_rotation()

get_sizes()
    Returns the sizes of the elements in the collection. The value represents the ‘area’ of the element.

        Returns sizes : array
            The ‘area’ of each element.

get_sketch_params()
    Returns the sketch parameters for the artist.

        Returns sketch_params : tuple or None
            A 3-tuple with the following elements:
            • scale: The amplitude of the wiggle perpendicular to the source line.
• length: The length of the wiggle along the line.
• randomness: The scale factor by which the length is shrunken or expanded.

May return None if no sketch parameters were set.

get_snap()
Returns the snap setting which may be:
• True: snap vertices to the nearest pixel center
• False: leave vertices as-is
• None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

get_transform()
Return the Transform instance used by this artist.

get_transformed_clip_path_and_affine()
Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

get_transforms()

get_url()
Returns the url

get_urls()

get_visible()
Return the artist’s visibility

get_window_extent(renderer)

get_zorder()
Return the Artist’s zorder.

have_units()
Return True if units are set on the x or y axes

hitlist(event)
List the children of the artist which contain the mouse event event.

is_figure_set()
Returns True if the artist is assigned to a Figure.

is_transform_set()
Returns True if Artist has a transform explicitly set.

mouseover
pchanged()
Fire an event when property changed, calling all of the registered callbacks.

pick(mouseevent)
Process pick event

each child artist will fire a pick event if mouseevent is over the artist and the artist has picker set

pickable()
Return True if Artist is pickable.

properties()
return a dictionary mapping property name -> value for all Artist props

remove()
Remove the artist from the figure if possible. The effect will not be visible until the figure is redrawn, e.g., with matplotlib.axes.Axes.draw_idle(). Call matplotlib.axes.Axes.relim() to update the axes limits if desired.

Note: relim() will not see collections even if the collection was added to axes with autolim = True.

Note: there is no support for removing the artist’s legend entry.

remove_callback(oid)
Remove a callback based on its id.

See also:

add_callback() For adding callbacks

set(**kwargs)
A property batch setter. Pass kwargs to set properties.

set_agg_filter(filter_func)
set agg_filter function.

set_alpha(alpha)
Set the alpha transparency of the collection. alpha must be a float or None.

ACCEPTS: float or None

set_animated(b)
Set the artist’s animation state.

ACCEPTS: [True | False]

set_antialiased(aa)
Set the antialiasing state for rendering.

ACCEPTS: Boolean or sequence of booleans

set_antialiaseds(aa)
alias for set_antialiased

set_array(A)
Set the image array from numpy array A
**set_axes**(*axes*)
Set the *Axes* instance in which the artist resides, if any.

This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

ACCEPTS: an *Axes* instance

**set_clim**(*vmin=None, vmax=None*)
set the norm limits for image scaling; if vmin is a length2 sequence, interpret it as (vmin, vmax) which is used to support setp

ACCEPTS: a length 2 sequence of floats

**set_clip_box**(*clipbox*)
Set the artist’s clip *Bbox*.

ACCEPTS: a *matplotlib.transforms.Bbox* instance

**set_clip_on**(*b*)
Set whether artist uses clipping.

When False artists will be visible out side of the axes which can lead to unexpected results.

ACCEPTS: [True | False]

**set_clip_path**(*path, transform=None*)
Set the artist’s clip path, which may be:

• a *Patch* (or subclass) instance
  
  • a *Path* instance, in which case an optional *Transform* instance may be provided, which will be applied to the path before using it for clipping.

• *None*, to remove the clipping path

For efficiency, if the path happens to be an axis-aligned rectangle, this method will set the clipping box to the corresponding rectangle and set the clipping path to *None*.

ACCEPTS: [ (*Path, Transform)* | *Patch* | *None* ]

**set_cmap**(*cmap*)
set the colormap for luminance data

ACCEPTS: a colormap or registered colormap name

**set_color**(*c*)
Set both the edgecolor and the facecolor.

ACCEPTS: matplotlib color arg or sequence of rgba tuples

See also:

**set_facecolor()**, **set_edgecolor()** For setting the edge or face color individually.

**set_contains**(*picker*)
Replace the contains test used by this artist. The new picker should be a callable function which determines whether the artist is hit by the mouse event:
hit, props = picker(artist, mouseevent)

If the mouse event is over the artist, return hit = True and props is a dictionary of properties you want returned with the contains test.

ACCEPTS: a callable function

**set_dashes**(*ls*)

alias for set_linestyle

**set_edgecolor**(*c*)

Set the edgecolor(s) of the collection. c can be a matplotlib color spec (all patches have same color), or a sequence of specs; if it is a sequence the patches will cycle through the sequence.

If c is ‘face’, the edge color will always be the same as the face color. If it is ‘none’, the patch boundary will not be drawn.

ACCEPTS: matplotlib color spec or sequence of specs

**set_edgecolors**(*c*)

alias for set_edgecolor

**set_facecolor**(*c*)

Set the facecolor(s) of the collection. c can be a matplotlib color spec (all patches have same color), or a sequence of specs; if it is a sequence the patches will cycle through the sequence.

If c is ‘none’, the patch will not be filled.

ACCEPTS: matplotlib color spec or sequence of specs

**set_facecolors**(*c*)

alias for set_facecolor

**set_figure**(*fig*)

Set the Figure instance the artist belongs to.

ACCEPTS: a matplotlib.figure.Figure instance

**set_gid**(*gid*)

Sets the (group) id for the artist

ACCEPTS: an id string

**set_hatch**(*hatch*)

Set the hatching pattern

hatch can be one of:

/ - diagonal hatching
\ - back diagonal
| - vertical
- - horizontal
+ - crossed
x - crossed diagonal
o - small circle
O - large circle
Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

ACCEPTS: [‘/’ | ‘\’ | ‘|’ | ‘-’ | ‘+’ | ‘x’ | ‘o’ | ‘O’ | ‘.’ | ‘*’ ]

set_label(s)
Set the label to s for auto legend.

ACCEPTS: string or anything printable with ‘%s’ conversion.

set_linestyle(ls)
Set the linestyle(s) for the collection.

<table>
<thead>
<tr>
<th>linestyle</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘-’ or ‘solid’</td>
<td>solid line</td>
</tr>
<tr>
<td>‘--’ or ‘dashed’</td>
<td>dashed line</td>
</tr>
<tr>
<td>‘-.’ or ‘dashdot’</td>
<td>dash-dotted line</td>
</tr>
<tr>
<td>‘:’ or ‘dotted’</td>
<td>dotted line</td>
</tr>
</tbody>
</table>

Alternatively a dash tuple of the following form can be provided:

(offset, onoffseq),

where onoffseq is an even length tuple of on and off ink in points.

ACCEPTS: [‘solid’ | ‘dashed’, ‘dashdot’, ‘dotted’ | (offset, on-off-dash-seq) | ‘-’ | ‘--’ | ‘-.’ | ‘:’ | ‘None’ | ‘ ‘ | ‘’ ]

Parameters ls : {‘-’, ‘--’, ‘-.’, ‘:’} and more see description

The line style.

set_linestyles(ls)
alias for set_linestyle

set_linewidth(lw)
Set the linewidth(s) for the collection. lw can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence

ACCEPTS: float or sequence of floats

set_linewidths(lw)
alias for set_linewidth

set_lw(lw)
alias for set_linewidth
**set_norm**(norm)

set the normalization instance

**set_offset_position**(offset_position)

Set how offsets are applied. If offset_position is ‘screen’ (default) the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is ‘data’, the offset is applied before the master transform, i.e., the offsets are in data coordinates.

**set_offsets**(offsets)

Set the offsets for the collection. offsets can be a scalar or a sequence.

ACCEPTS: float or sequence of floats

**set_path_effects**(path_effects)

set path_effects, which should be a list of instances of matplotlib.path_effect._Base class or its derivatives.

**set_paths()**

**set_picker**(picker)

Set the epsilon for picking used by this artist

picker can be one of the following:

- *None*: picking is disabled for this artist (default)
- A boolean: if True then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist
- A float: if picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if it’s data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event
- A function: if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

  hit, props = picker(artist, mouseevent)

  to determine the hit test. if the mouse event is over the artist, return hit=True and props is a dictionary of properties you want added to the PickEvent attributes.

ACCEPTS: [None|float|boolean|callable]

**set_pickradius**(pr)

**set_rasterized**(rasterized)

Force rasterized (bitmap) drawing in vector backend output.

Defaults to None, which implies the backend’s default behavior

ACCEPTS: [True | False | None]
**set_sizes**(*sizes, dpi=72.0*)
Set the sizes of each member of the collection.

**Parameters**

`sizes` : ndarray or None
The size to set for each element of the collection. The value is the ‘area’ of the element.

`dpi` : float
The dpi of the canvas. Defaults to 72.0.

**set_sketch_params**(*scale=None, length=None, randomness=None*)
Sets the sketch parameters.

**Parameters**

`scale` : float, optional
The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is None, or not provided, no sketch filter will be provided.

`length` : float, optional
The length of the wiggle along the line, in pixels (default 128.0)

`randomness` : float, optional
The scale factor by which the length is shrunken or expanded (default 16.0)

**set_snap**(*snap*)
Sets the snap setting which may be:

- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

**set_transform**(*t*)
Set the *Transform* instance used by this artist.

* ACCEPTS: *Transform* instance

**set_url**(*url*)
Sets the url for the artist

* ACCEPTS: a url string

**set_urls**(*urls*)

**set_visible**(*b*)
Set the artist’s visibility.

* ACCEPTS: [True | False]
**set_zorder**(level)
Set the zorder for the artist. Artists with lower zorder values are drawn first.

ACCEPTS: any number

**stale**
If the artist is 'stale' and needs to be re-drawn for the output to match the internal state of the artist.

**sticky_edges**
\(x\) and \(y\) sticky edge lists.

When performing autoscaling, if a data limit coincides with a value in the corresponding sticky_edges list, then no margin will be added—the view limit “sticks” to the edge. A typical usecase is histograms, where one usually expects no margin on the bottom edge (0) of the histogram.

This attribute cannot be assigned to; however, the \(x\) and \(y\) lists can be modified in place as needed.

**Examples**

```python
>>> artist.sticky_edges.x[:] = (xmin, xmax)
>>> artist.sticky_edges.y[:] = (ymin, ymax)
```

**to_rgba**(x, alpha=None, bytes=False, norm=True)
Return a normalized rgba array corresponding to \(x\).

In the normal case, \(x\) is a 1-D or 2-D sequence of scalars, and the corresponding ndarray of rgba values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already rgb or rgba, such as might have been read from an image file. If \(x\) is an ndarray with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an rgb or rgba array, and no mapping will be done. If the last dimension is 3, the alpha kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the alpha kwarg is ignored; it does not replace the pre-existing alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if bytes is False (default), the rgba array will be floats in the 0-1 range; if it is True, the returned rgba array will be uint8 in the 0 to 255 range.

If norm is False, no normalization of the input data is performed, and it is assumed to already be in the range (0-1).

Note: this method assumes the input is well-behaved; it does not check for anomalies such as \(x\) being a masked rgba array, or being an integer type other than uint8, or being a floating point rgba array with values outside the 0-1 range.

**update**(props)
Update the properties of this Artist from the dictionary prop.

**update_from**(other)
copy properties from other to self
update_scalarmappable()
If the scalar mappable array is not none, update colors from scalar data

zorder = 0

class matplotlib.collections.BrokenBarHCollection(xranges, yrange, **kwargs)
Bases: matplotlib.collections.PolyCollection
A collection of horizontal bars spanning yrange with a sequence of xranges.

xranges sequence of (xmin, xwidth)

yrange ymin, ywidth
Valid Collection keyword arguments:
• edgecolors: None
• facecolors: None
• linewidths: None
• antialiaseds: None
• offsets: None
• transOffset: transforms.IdentityTransform()
• norm: None (optional for matplotlib.cm.ScalarMappable)
• cmap: None (optional for matplotlib.cm.ScalarMappable)

offsets and transOffset are used to translate the patch after rendering (default no offsets)

If any of edgecolors, facecolors, linewidths, antialiaseds are None, they default to their matplotlib.rcParams patch setting, in sequence form.

add_callback(func)
Adds a callback function that will be called whenever one of the Artist's properties changes.
Returns an id that is useful for removing the callback with remove_callback() later.

add_checker(checker)
Add an entry to a dictionary of boolean flags that are set to True when the mappable is changed.

aname = 'Artist'

autoscale()
Autoscale the scalar limits on the norm instance using the current array

autoscale_None()
Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

axes
The Axes instance the artist resides in, or None.
changed()
Call this whenever the mappable is changed to notify all the callback listeners to the
'changed' signal

check_update(checker)
If mappable has changed since the last check, return True; else return False

contains(mouseevent)
Test whether the mouse event occurred in the collection.
Returns True | False, dict(ind=itemlist), where every item in itemlist contains the event.

convert_xunits(x)
For artists in an axes, if the xaxis has units support, convert x using xaxis unit type

convert_yunits(y)
For artists in an axes, if the yaxis has units support, convert y using yaxis unit type

draw(artist, renderer, *args, **kwargs)

findobj(match=None, include_self=True)
Find artist objects.
Recursively find all Artist instances contained in self.

match can be
• None: return all objects contained in artist.
• function with signature boolean = match(artist) used to filter matches
• class instance: e.g., Line2D. Only return artists of class type.
If include_self is True (default), include self in the list to be checked for a match.

format_cursor_data(data)
Return cursor data string formatted.

get_agg_filter()  
return filter function to be used for agg filter

get_alpha()  
Return the alpha value used for blending - not supported on all backends

get_animated()  
Return the artist’s animated state

get_array()  
Return the array

get_axes()  
Return the Axes instance the artist resides in, or None.
This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

get_children()  
Return a list of the child Artist’s this :class:`Artist` contains.
get_clim()  
return the min, max of the color limits for image scaling

get_clip_box()  
Return artist clipbox

get_clip_on()  
Return whether artist uses clipping

get_clip_path()  
Return artist clip path

get_cmap()  
return the colormap

get_contains()  
Return the _contains test used by the artist, or None for default.

get_cursor_data(event)  
Get the cursor data for a given event.

get_dashes()  

get_datalim(transData)  

get_edgecolor()  

get_edgecolors()  

get_facecolor()  

get_facecolors()  

get_figure()  
Return the Figure instance the artist belongs to.

get_fill()  
return whether fill is set

get_gid()  
Returns the group id

get_hatch()  
Return the current hatching pattern

get_label()  
Get the label used for this artist in the legend.

get_linestyle()  

get_linestyles

get_linewidth

get_linewidths

get_offset_position

Returns how offsets are applied for the collection. If offset_position is ‘screen’, the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is ‘data’, the offset is applied before the master transform, i.e., the offsets are in data coordinates.

get_offset_transform

get_offsets

Return the offsets for the collection.

get_path_effects

get_paths

get_picker

Return the picker object used by this artist

get_pickradius

get_rasterized

return True if the artist is to be rasterized

get_sizes

Returns the sizes of the elements in the collection. The value represents the ‘area’ of the element.

Returns sizes : array

The ‘area’ of each element.

get_sketch_params

Returns the sketch parameters for the artist.

Returns sketch_params : tuple or None

A 3-tuple with the following elements:

• scale: The amplitude of the wiggle perpendicular to the source line.
• length: The length of the wiggle along the line.
• randomness: The scale factor by which the length is shrunken or expanded.

May return None if no sketch parameters were set.
get_snap()
Returns the snap setting which may be:

- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

get_transform()
Return the Transform instance used by this artist.

get_transformed_clip_path_and_affine()
Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

get_transforms()

get_url()
Returns the url

get_urls()

get_visible()
Return the artist’s visibility

get_window_extent(renderer)

get_zorder()
Return the Artist’s zorder.

have_units()
Return True if units are set on the x or y axes

hitlist(event)
List the children of the artist which contain the mouse event event.

is_figure_set()
Returns True if the artist is assigned to a Figure.

is_transform_set()
Returns True if Artist has a transform explicitly set.

mouseover

pchanged()
Fire an event when property changed, calling all of the registered callbacks.

pick(mouseevent)
Process pick event
each child artist will fire a pick event if mouseevent is over the artist and the artist has picker set

**pickable()**
Return True if Artist is pickable.

**properties()**
return a dictionary mapping property name -> value for all Artist props

**remove()**
Remove the artist from the figure if possible. The effect will not be visible until the figure is redrawn, e.g., with matplotlib.axes.Axes.draw_idle(). Call matplotlib.axes.Axes.relim() to update the axes limits if desired.

Note: relim() will not see collections even if the collection was added to axes with autolim = True.

Note: there is no support for removing the artist’s legend entry.

**remove_callback(oid)**
Remove a callback based on its id.

See also:

**add_callback()** For adding callbacks

**set(**kwargs**)
A property batch setter. Pass kwargs to set properties.

**set_agg_filter(filter_func)**
set agg_filter function.

**set_alpha(alpha)**
Set the alpha transparencies of the collection. alpha must be a float or None.

ACCEPTS: float or None

**set_animated(b)**
Set the artist’s animation state.

ACCEPTS: [True | False]

**set_antialiased(aa)**
Set the antialiasing state for rendering.

ACCEPTS: Boolean or sequence of booleans

**set_antialiaseds(aa)**
alias for set_antialiased

**set_array(A)**
Set the image array from numpy array A

**set_axes(axes)**
Set the Axes instance in which the artist resides, if any.

This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.
set_clim(vmin=None, vmax=None)
set the norm limits for image scaling; if vmin is a length2 sequence, interpret it as (vmin, vmax) which is used to support setp
ACCEPSTS: a length 2 sequence of floats

set_clip_box(clipbox)
Set the artist’s clip Bbox.
ACCEPSTS: a matplotlib.transforms.Bbox instance

set_clip_on(b)
Set whether artist uses clipping.
When False artists will be visible out side of the axes which can lead to unexpected results.
ACCEPSTS: [True | False]

set_clip_path(path, transform=None)
Set the artist’s clip path, which may be:
• a Patch (or subclass) instance
  • a Path instance, in which case an optional Transform instance may be provided, which
    will be applied to the path before using it for clipping.
  • None, to remove the clipping path
For efficiency, if the path happens to be an axis-aligned rectangle, this method will set the clipping box to the corresponding rectangle and set the clipping path to None.
ACCEPSTS: [(Path, Transform)| Patch | None ]

set_cmap(cmap)
set the colormap for luminance data
ACCEPSTS: a colormap or registered colormap name

set_color(c)
Set both the edgecolor and the facecolor.
ACCEPSTS: matplotlib color arg or sequence of rgba tuples

See also:

set_facecolor(), set_edgecolor() For setting the edge or face color individually.

set_contains(picker)
Replace the contains test used by this artist. The new picker should be a callable function which determines whether the artist is hit by the mouse event:

hit, props = picker(artist, mouseevent)

If the mouse event is over the artist, return hit = True and props is a dictionary of properties you want returned with the contains test.
ACCEP'TS: a callable function

`set_dashes(ls)`
alias for `set_linestyle`

`set_edgecolor(c)`
Set the edg color(s) of the collection. `c` can be a matplotlib color spec (all patches have same color), or a sequence of specs; if it is a sequence the patches will cycle through the sequence.

If `c` is ‘face’, the edge color will always be the same as the face color. If it is ‘none’, the patch boundary will not be drawn.

ACCEP'T'S: matplotlib color spec or sequence of specs

`set_edgecolors(c)`
alias for `set_edgecolor`

`set_facecolor(c)`
Set the facecolor(s) of the collection. `c` can be a matplotlib color spec (all patches have same color), or a sequence of specs; if it is a sequence the patches will cycle through the sequence.

If `c` is ‘none’, the patch will not be filled.

ACCEP'T'S: matplotlib color spec or sequence of specs

`set_facecolors(c)`
alias for `set_facecolor`

`set_figure(fig)`
Set the `Figure` instance the artist belongs to.

ACCEP'T'S: a `matplotlib.figure.Figure` instance

`set_gid(gid)`
Sets the (group) id for the artist

ACCEP'T'S: an id string

`set_hatch(hatch)`
Set the hatching pattern

`hatch` can be one of:

```
/ - diagonal hatching
\ - back diagonal
| - vertical
- - horizontal
+ - crossed
x - crossed diagonal
o - small circle
O - large circle
. - dots
* - stars
```

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.
Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

ACCEPTS: [‘\‘’ | ‘\‘’ | ‘-‘ | ‘\‘’ | ‘-‘ | ‘-‘ | ‘\‘’ | ‘-‘ | ‘\‘’ | ‘-‘ | ‘\‘’]

**set_label(s)**

Set the label to `s` for auto legend.

ACCEPTS: string or anything printable with ‘%s’ conversion.

**set_linestyle(ls)**

Set the linestyle(s) for the collection.

<table>
<thead>
<tr>
<th>linestyle</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘-‘ or ‘solid’</td>
<td>solid line</td>
</tr>
<tr>
<td>‘-\‘’ or ‘dashed’</td>
<td>dashed line</td>
</tr>
<tr>
<td>‘-\‘’ or ‘dashdot’</td>
<td>dash-dotted line</td>
</tr>
<tr>
<td>‘:‘ or ‘dotted’</td>
<td>dotted line</td>
</tr>
</tbody>
</table>

Alternatively a dash tuple of the following form can be provided:

```
(offset, onoffseq),
```

where onoffseq is an even length tuple of on and off ink in points.

**Parameters** `ls` : {‘-‘, ‘-\‘’, ‘\‘’, ‘:\‘’} and more see description

The line style.

**set_linestyles(ls)**

alias for `set_linestyle`

**set_linewidth(lw)**

Set the linewidth(s) for the collection. `lw` can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence.

ACCEPTS: float or sequence of floats

**set_linewidths(lw)**

alias for `set_linewidth`

**set_lw(lw)**

alias for `set_linewidth`

**set_norm(norm)**

set the normalization instance

**set_offset_position(offset_position)**

Set how offsets are applied. If `offset_position` is ‘screen’ (default) the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If
offset_position is ‘data’, the offset is applied before the master transform, i.e., the offsets are in data coordinates.

**set_offsets**(*offsets*)

Set the offsets for the collection. *offsets* can be a scalar or a sequence.

ACCEPTS: float or sequence of floats

**set_path_effects**(*path_effects*)

Set path_effects, which should be a list of instances of matplotlib.path_effects.PathEffect._Base class or its derivatives.

**set_paths**(*verts, closed=True*)

This allows one to delay initialization of the vertices.

**set_picker**(*picker*)

Set the epsilon for picking used by this artist

*picker* can be one of the following:

- *None*: picking is disabled for this artist (default)
- A boolean: if True then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist
- A float: if picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if it’s data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event
- A function: if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```python
hit, props = picker(artist, mouseevent)
```

to determine the hit test. if the mouse event is over the artist, return *hit=True* and *props* is a dictionary of properties you want added to the PickEvent attributes.

ACCEPTS: [None|float|boolean|callable]

**set_pickradius**(*pr*)

**set_rasterized**(*rasterized*)

Force rasterized (bitmap) drawing in vector backend output.

Defaults to None, which implies the backend’s default behavior

ACCEPTS: [True | False | None]

**set_sizes**(*sizes, dpi=72.0*)

Set the sizes of each member of the collection.

**Parameters**

*sizes*: ndarray or None

The size to set for each element of the collection. The value is the ‘area’ of the element.
**dpi**: float

The dpi of the canvas. Defaults to 72.0.

**set_sketch_params**(scale=None, length=None, randomness=None)

Sets the sketch parameters.

**Parameters**

- **scale**: float, optional
  
  The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is None, or not provided, no sketch filter will be provided.

- **length**: float, optional
  
  The length of the wiggle along the line, in pixels (default 128.0)

- **randomness**: float, optional
  
  The scale factor by which the length is shrunken or expanded (default 16.0)

**set_snap**(snap)

Sets the snap setting which may be:

- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

**set_transform**(t)

Set the *Transform* instance used by this artist.

**ACCEPTS**: *Transform* instance

**set_url**(url)

Sets the url for the artist

**ACCEPTS**: a url string

**set_urls**(urls)

**set_verts**(verts, closed=True)

This allows one to delay initialization of the vertices.

**set_verts_and_codes**(verts, codes)

This allows one to initialize vertices with path codes.

**set_visible**(b)

Set the artist’s visibility.

**ACCEPTS**: [True | False]

**set_zorder**(level)

Set the zorder for the artist. Artists with lower zorder values are drawn first.

**ACCEPTS**: any number
**static span_where** *(x, ymin, ymax, where, **kwargs)*

Create a BrokenBarHCollection to plot horizontal bars from over the regions in `x` where `where` is True. The bars range on the y-axis from `ymin` to `ymax`.

A BrokenBarHCollection is returned. `kwargs` are passed on to the collection.

**stale**

If the artist is ‘stale’ and needs to be re-drawn for the output to match the internal state of the artist.

**sticky_edges**

`x` and `y` sticky edge lists.

When performing autoscaling, if a data limit coincides with a value in the corresponding `sticky_edges` list, then no margin will be added—the view limit “sticks” to the edge. A typical usecase is histograms, where one usually expects no margin on the bottom edge (0) of the histogram.

This attribute cannot be assigned to; however, the `x` and `y` lists can be modified in place as needed.

**Examples**

```python
>>> artist.sticky_edges.x[:] = (xmin, xmax)
>>> artist.sticky_edges.y[:] = (ymin, ymax)
```

**to_rgba** *(x, alpha=None, bytes=False, norm=True)*

Return a normalized rgba array corresponding to `x`.

In the normal case, `x` is a 1-D or 2-D sequence of scalars, and the corresponding ndarray of rgba values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already rgb or rgba, such as might have been read from an image file. If `x` is an ndarray with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an rgb or rgba array, and no mapping will be done. If the last dimension is 3, the `alpha` kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the `alpha` kwarg is ignored; it does not replace the pre-existing alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if `bytes` is `False` (default), the rgba array will be floats in the 0-1 range; if it is `True`, the returned rgba array will be uint8 in the 0 to 255 range.

If norm is `False`, no normalization of the input data is performed, and it is assumed to already be in the range (0-1).

Note: this method assumes the input is well-behaved; it does not check for anomalies such as `x` being a masked rgba array, or being an integer type other than uint8, or being a floating point rgba array with values outside the 0-1 range.

**update** *(props)*

Update the properties of this Artist from the dictionary `prop`.
update_from(other)
    copy properties from other to self

update_scalarmappable()
    If the scalar mappable array is not none, update colors from scalar data

zorder = 0

class matplotlib.collections.CircleCollection(sizes, **kwargs)
    Bases: matplotlib.collections._CollectionWithSizes
    A collection of circles, drawn using splines.

sizes  Gives the area of the circle in points^2

Valid Collection keyword arguments:
    • edgecolors: None
    • facecolors: None
    • linewidths: None
    • antialiaseds: None
    • offsets: None
    • transOffset: transforms.IdentityTransform()
    • norm: None (optional for matplotlib.cm.ScalarMappable)
    • cmap: None (optional for matplotlib.cm.ScalarMappable)

offsets and transOffset are used to translate the patch after rendering (default no offsets)

If any of edgecolors, facecolors, linewidths, antialiaseds are None, they default to their matplotlib.
rcParams patch setting, in sequence form.

add_callback(func)
    Adds a callback function that will be called whenever one of the Artist’s properties changes.

    Returns an id that is useful for removing the callback with remove_callback() later.

add_checker(checker)
    Add an entry to a dictionary of boolean flags that are set to True when the mappable is changed.

aname = ‘Artist’

autoscale()
    Autoscale the scalar limits on the norm instance using the current array

autoscale_None()
    Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

axes
    The Axes instance the artist resides in, or None.
changed()
   Call this whenever the mappable is changed to notify all the callback listeners to the
   ‘changed’ signal

check_update(checker)
   If mappable has changed since the last check, return True; else return False

contains(mouseevent)
   Test whether the mouse event occurred in the collection.
   Returns True | False, dict(ind=itemlist), where every item in itemlist contains the event.

convert_xunits(x)
   For artists in an axes, if the xaxis has units support, convert x using xaxis unit type

convert_yunits(y)
   For artists in an axes, if the yaxis has units support, convert y using yaxis unit type

draw(artist, renderer, *args, **kwargs)

findobj(match=None, include_self=True)
   Find artist objects.
   Recursively find all Artist instances contained in self.

   match can be
   • None: return all objects contained in artist.
   • function with signature boolean = match(artist) used to filter matches
   • class instance: e.g., Line2D. Only return artists of class type.

   If include_self is True (default), include self in the list to be checked for a match.

format_cursor_data(data)
   Return cursor data string formatted.

get_agg_filter()
   return filter function to be used for agg filter

get_alpha()
   Return the alpha value used for blending - not supported on all backends

get_animated()
   Return the artist’s animated state

get_array()
   Return the array

get_axes()
   Return the Axes instance the artist resides in, or None.

   This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

get_children()
   Return a list of the child Artist’s this :class:`Artist` contains.
get_clim()  
    return the min, max of the color limits for image scaling

get_clip_box()  
    Return artist clipbox

get_clip_on()  
    Return whether artist uses clipping

get_clip_path()  
    Return artist clip path

get_cmap()  
    return the colormap

get_contains()  
    Return the _contains test used by the artist, or None for default.

get_cursor_data(event)  
    Get the cursor data for a given event.

get_dashes()

get_datalim(transData)

get_edgecolor()  

get_edgecolors()

get_facecolor()  

get_facecolors()

get_figure()  
    Return the Figure instance the artist belongs to.

get_fill()  
    return whether fill is set

get_gid()  
    Returns the group id

get_hatch()  
    Return the current hatching pattern

get_label()  
    Get the label used for this artist in the legend.

get_linestyle()
get_linestyles()

get_linewidth()

get_linewidths()

get_offset_position()
   Returns how offsets are applied for the collection. If offset_position is ‘screen’, the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is ‘data’, the offset is applied before the master transform, i.e., the offsets are in data coordinates.

get_offset_transform()

get_offsets()
   Return the offsets for the collection.

get_path_effects()

get_paths()

get_picker()
   Return the picker object used by this artist

get_pickradius()

get_rasterized()
   return True if the artist is to be rasterized

get_sizes()
   Returns the sizes of the elements in the collection. The value represents the ‘area’ of the element.

   Returns sizes : array
      The ‘area’ of each element.

get_sketch_params()
   Returns the sketch parameters for the artist.

   Returns sketch_params : tuple or None
      A 3-tuple with the following elements:
      • scale: The amplitude of the wiggle perpendicular to the source line.
      • length: The length of the wiggle along the line.
      • randomness: The scale factor by which the length is shrunken or expanded.
      May return None if no sketch parameters were set.
get_snap()  
Returns the snap setting which may be:

- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

get_transform()  
Return the Transform instance used by this artist.

get_transformed_clip_path_and_affine()  
Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

get_transforms()  

get_url()  
Returns the url

get_urls()  

get_visible()  
Return the artist’s visibility

get_window_extent(renderer)  

get_zorder()  
Return the Artist’s zorder.

have_units()  
Return True if units are set on the x or y axes

hitlist(event)  
List the children of the artist which contain the mouse event event.

is_figure_set()  
Returns True if the artist is assigned to a Figure.

is_transform_set()  
Returns True if Artist has a transform explicitly set.

mouseover

pchanged()  
Fire an event when property changed, calling all of the registered callbacks.

pick(mouseevent)  
Process pick event
each child artist will fire a pick event if *mouseevent* is over the artist and the artist has picker set

**pickable()**

Return *True* if *Artist* is pickable.

**properties()**

return a dictionary mapping property name -\> value for all *Artist* props

**remove()**

Remove the artist from the figure if possible. The effect will not be visible until the figure is redrawn, e.g., with *matplotlib.axes.Axes.draw_idle()*. Call *matplotlib.axes.Axes.relim()* to update the axes limits if desired.

Note: *relim()* will not see collections even if the collection was added to axes with *autolim = True*.

Note: there is no support for removing the artist’s legend entry.

**remove_callback(oid)**

Remove a callback based on its *id*.

See also:

**add_callback()** For adding callbacks

**set(****kwargs**

A property batch setter. Pass *kwargs* to set properties.

**set_agg_filter(filter_func)**

set agg_filter function.

**set_alpha(alpha)**

Set the alpha transparencies of the collection. *alpha* must be a float or *None*.

ACCEPTS: float or None

**set_animated(b)**

Set the artist’s animation state.

ACCEPTS: [True | False]

**set_antialiased(aa)**

Set the antialiasing state for rendering.

ACCEPTS: Boolean or sequence of booleans

**set_antialiaseds(aa)**

alias for *set_antialiased*

**set_array(A)**

Set the image array from numpy array *A*

**set_axes(axes)**

Set the *Axes* instance in which the artist resides, if any.

This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.
set_clim(vmin=None, vmax=None)
set the norm limits for image scaling; if vmin is a length2 sequence, interpret it as (vmin, vmax) which is used to support setp
ACCEPST: a length 2 sequence of floats

set_clip_box(clipbox)
Set the artist’s clip Bbox.
ACCEPST: a matplotlib.transforms.Bbox instance

set_clip_on(b)
Set whether artist uses clipping.
When False artists will be visible out side of the axes which can lead to unexpected results.
ACCEPST: [True | False]

set_clip_path(path, transform=None)
Set the artist’s clip path, which may be:
• a Patch (or subclass) instance
• a Path instance, in which case an optional Transform instance may be provided, which will be applied to the path before using it for clipping.
• None, to remove the clipping path
For efficiency, if the path happens to be an axis-aligned rectangle, this method will set the clipping box to the corresponding rectangle and set the clipping path to None.
ACCEPST: [ (Path, Transform) | Patch | None ]

set_cmap(cmap)
set the colormap for luminance data
ACCEPST: a colormap or registered colormap name

set_color(c)
Set both the edgecolor and the facecolor.
ACCEPST: matplotlib color arg or sequence of rgba tuples
See also:

set_facecolor(), set_edgecolor() For setting the edge or face color individually.

set_contains(picker)
Replace the contains test used by this artist. The new picker should be a callable function which determines whether the artist is hit by the mouse event:

hit, props = picker(artist, mouseevent)

If the mouse event is over the artist, return hit = True and props is a dictionary of properties you want returned with the contains test.
ACCEPtS: a callable function

```
set_dashes(ls)
```
alias for set_linestyle

```
set_edgecolor(c)
```
Set the edgecolor(s) of the collection. `c` can be a matplotlib color spec (all patches have same color), or a sequence of specs; if it is a sequence the patches will cycle through the sequence.

If `c` is ‘face’, the edge color will always be the same as the face color. If it is ‘none’, the patch boundary will not be drawn.

ACCEPtS: matplotlib color spec or sequence of specs

```
set_edgecolors(c)
```
alias for set_edgecolor

```
set_facecolor(c)
```
Set the facecolor(s) of the collection. `c` can be a matplotlib color spec (all patches have same color), or a sequence of specs; if it is a sequence the patches will cycle through the sequence.

If `c` is ‘none’, the patch will not be filled.

ACCEPtS: matplotlib color spec or sequence of specs

```
set_facecolors(c)
```
alias for set_facecolor

```
set_figure(fig)
```
Set the `Figure` instance the artist belongs to.

ACCEPtS: a `matplotlib.figure.Figure` instance

```
set_gid(gid)
```
Sets the (group) id for the artist

ACCEPtS: an id string

```
set_hatch(hatch)
```
Set the hatching pattern

`hatch` can be one of:

```
/ - diagonal hatching
\ - back diagonal
| - vertical
- - horizontal
+ - crossed
x - crossed diagonal
o - small circle
O - large circle.
. - dots
* - stars
```

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.
Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

**ACCEPTS:** ['/' | '\'] | '|' | '+' | 'x' | 'o' | 'O' | '.' | '*' ]

**set_label(s)**
Set the label to s for auto legend.

**ACCEPTS:** string or anything printable with ‘%s’ conversion.

**set_linestyle(ls)**
Set the linestyle(s) for the collection.

<table>
<thead>
<tr>
<th>linestyle</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'-' or 'solid'</td>
<td>solid line</td>
</tr>
<tr>
<td>'--' or 'dashed'</td>
<td>dashed line</td>
</tr>
<tr>
<td>'-.' or 'dashdot'</td>
<td>dash-dotted line</td>
</tr>
<tr>
<td>':' or 'dotted'</td>
<td>dotted line</td>
</tr>
</tbody>
</table>

Alternatively a dash tuple of the following form can be provided:

```
(offset, onoffseq),
```

where onoffseq is an even length tuple of on and off ink in points.

**ACCEPTS:** ['solid' | 'dashed' | 'dashdot' | 'dotted' | (offset, on-off-dash-seq) | '-' | '--' | '-.' | ':' | 'None' | ' ' | '']

**Parameters ls :** {'-', '--', '-.', ':'} and more see description

The line style.

**set_linestyles(ls)**
alias for set_linestyle

**set_linewidth(lw)**
Set the linewidth(s) for the collection. lw can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence

**ACCEPTS:** float or sequence of floats

**set_linewidths(lw)**
alias for set_linewidth

**set_lw(lw)**
alias for set_linewidth

**set_norm(norm)**
set the normalization instance

**set_offset_position(offset_position)**
Set how offsets are applied. If offset_position is ‘screen’ (default) the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If
offset_position is ‘data’, the offset is applied before the master transform, i.e., the offsets are in data coordinates.

**set_offsets**(*offsets*)
Set the offsets for the collection. *offsets* can be a scalar or a sequence.

**set_path_effects**(*path_effects*)
set path_effects, which should be a list of instances of matplotlib.path.effect._Base class or its derivatives.

**set_paths**()

**set_picker**(*picker*)
Set the epsilon for picking used by this artist

*picker* can be one of the following:

- *None*: picking is disabled for this artist (default)
- A boolean: if *True* then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist
- A float: if picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if it’s data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event
- A function: if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

to determine the hit test. if the mouse event is over the artist, return *hit=True* and props is a dictionary of properties you want added to the PickEvent attributes.

**set_pickradius**(*pr*)

**set_rasterized**(*rasterized*)
Force rasterized (bitmap) drawing in vector backend output.

Defaults to *None*, which implies the backend’s default behavior

**set_sizes**(*sizes*, *dpi=72.0*)
Set the sizes of each member of the collection.

**Parameters**

*sizes* : ndarray or None
The size to set for each element of the collection. The value is the ‘area’ of the element.
**dpi** : float

The dpi of the canvas. Defaults to 72.0.

**set_sketch_params** *(scale=None, length=None, randomness=None)*

Sets the sketch parameters.

**Parameters**

**scale** : float, optional

The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is `None`, or not provided, no sketch filter will be provided.

**length** : float, optional

The length of the wiggle along the line, in pixels (default 128.0)

**randomness** : float, optional

The scale factor by which the length is shrunken or expanded (default 16.0)

**set_snap** *(snap)*

Sets the snap setting which may be:

- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

**set_transform** *(t)*

Set the `Transform` instance used by this artist.

**ACCEPTS**: `Transform` instance

**set_url** *(url)*

Sets the url for the artist

**ACCEPTS**: a url string

**set_urls** *(urls)*

**set_visible** *(b)*

Set the artist’s visibility.

**ACCEPTS**: [True \| False]

**set_zorder** *(level)*

Set the zorder for the artist. Artists with lower zorder values are drawn first.

**ACCEPTS**: any number

**stale**

If the artist is ‘stale’ and needs to be re-drawn for the output to match the internal state of the artist.
**sticky_edges**

x and y sticky edge lists.

When performing autoscaling, if a data limit coincides with a value in the corresponding sticky_edges list, then no margin will be added–the view limit “sticks” to the edge. A typical usecase is histograms, where one usually expects no margin on the bottom edge (0) of the histogram.

This attribute cannot be assigned to; however, the x and y lists can be modified in place as needed.

**Examples**

```python
>>> artist.sticky_edges.x[:] = (xmin, xmax)
>>> artist.sticky_edges.y[:] = (ymin, ymax)
```

**to_rgba**(x, alpha=None, bytes=False, norm=True)

Return a normalized rgba array corresponding to x.

In the normal case, x is a 1-D or 2-D sequence of scalars, and the corresponding ndarray of rgba values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already rgb or rgba, such as might have been read from an image file. If x is an ndarray with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an rgb or rgba array, and no mapping will be done. If the last dimension is 3, the alpha kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the alpha kwarg is ignored; it does not replace the pre-existing alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if bytes is False (default), the rgba array will be floats in the 0-1 range; if it is True, the returned rgba array will be uint8 in the 0 to 255 range.

If norm is False, no normalization of the input data is performed, and it is assumed to already be in the range (0-1).

Note: this method assumes the input is well-behaved; it does not check for anomalies such as x being a masked rgba array, or being an integer type other than uint8, or being a floating point rgba array with values outside the 0-1 range.

**update**(props)

Update the properties of this Artist from the dictionary prop.

**update_from**(other)

copy properties from other to self

**update_scalarmappable**()

If the scalar mappable array is not none, update colors from scalar data

zorder = 0
class matplotlib.collections.Collection(
    edgecolors=None, facecolors=None, linewidths=None, linestyles='solid', antialiaseds=None, offsets=None, transOffset=None, norm=None, cmap=None, pickradius=5.0, hatch=None, urls=None, offset_position='screen', zorder=1, **kwargs)

Bases: matplotlib.artist.Artist, matplotlib.cm.ScalarMappable

Base class for Collections. Must be subclassed to be usable.

All properties in a collection must be sequences or scalars; if scalars, they will be converted to sequences. The property of the ith element of the collection is:

\[
\text{prop}[i \mod \text{len(props)}]
\]

Keyword arguments and default values:

- **edgecolors**: None
- **facecolors**: None
- **linewidths**: None
- **antialiaseds**: None
- **offsets**: None
- **transOffset**: transforms.IdentityTransform()
- **offset_position**: ‘screen’ (default) or ‘data’
- **norm**: None (optional for `matplotlib.cm.ScalarMappable`)
- **cmap**: None (optional for `matplotlib.cm.ScalarMappable`)
- **hatch**: None
- **zorder**: 1

`offsets` and `transOffset` are used to translate the patch after rendering (default no offsets). If offset_position is ‘screen’ (default) the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is ‘data’, the offset is applied before the master transform, i.e., the offsets are in data coordinates.

If any of `edgecolors`, `facecolors`, `linewidths`, `antialiaseds` are None, they default to their `matplotlib.rcParams` patch setting, in sequence form.

The use of `ScalarMappable` is optional. If the `ScalarMappable` matrix `_A` is not None (i.e., a call to `set_array` has been made), at draw time a call to scalar mappable will be made to set the face colors.

Create a Collection

%(Collection)s

`add_callback(func)`

Adds a callback function that will be called whenever one of the `Artist`'s properties changes.

Returns an `id` that is useful for removing the callback with `remove_callback()` later.
add_checker(checker)
    Add an entry to a dictionary of boolean flags that are set to True when the mappable is changed.

aname = ‘Artist’

autoscale()
    Autoscale the scalar limits on the norm instance using the current array

autoscale_None()
    Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

axes
    The Axes instance the artist resides in, or None.

changed()
    Call this whenever the mappable is changed to notify all the callback listeners to the ‘changed’ signal

check_update(checker)
    If mappable has changed since the last check, return True; else return False

contains(mouseevent)
    Test whether the mouse event occurred in the collection.
    Returns True | False, dict(ind=itemlist), where every item in itemlist contains the event.

convert_xunits(x)
    For artists in an axes, if the xaxis has units support, convert x using xaxis unit type

convert_yunits(y)
    For artists in an axes, if the yaxis has units support, convert y using yaxis unit type

draw(artist, renderer, *args, **kwargs)

findobj(match=None, include_self=True)
    Find artist objects.
    Recursively find all Artist instances contained in self.

    match can be
    • None: return all objects contained in artist.
    • function with signature boolean = match(artist) used to filter matches
    • class instance: e.g., Line2D. Only return artists of class type.

    If include_self is True (default), include self in the list to be checked for a match.

format_cursor_data(data)
    Return cursor data string formatted.

get_agg_filter()
    return filter function to be used for agg filter
get_alpha()
Return the alpha value used for blending - not supported on all backends

get_animated()
Return the artist’s animated state

get_array()
Return the array

get_axes()
Return the Axes instance the artist resides in, or None.
This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

get_children()
Return a list of the child Artist's this :class:`Artist` contains.

get_clim()
return the min, max of the color limits for image scaling

get_clip_box()
Return artist clipbox

get_clip_on()
Return whether artist uses clipping

get_clip_path()
Return artist clip path

get_cmap()
return the colormap

get_contains()
Return the _contains test used by the artist, or None for default.

get_cursor_data(event)
Get the cursor data for a given event.

get_dashes()

get_datalim(transData)

get_edgecolor()

get_edgecolors()

get_facecolor()

get_facecolors()
get_figure()
Return the Figure instance the artist belongs to.

get_fill()
return whether fill is set

get_gid()
Returns the group id

get_hatch()
Return the current hatching pattern

get_label()
Get the label used for this artist in the legend.

get_linestyle()

get_linestyles()

get_linewidth()

get_linewidths()

get_offset_position()
Returns how offsets are applied for the collection. If offset_position is ‘screen’, the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is ‘data’, the offset is applied before the master transform, i.e., the offsets are in data coordinates.

get_offset_transform()

get_offsets()
Return the offsets for the collection.

get_path_effects()

get_paths()

get_picker()
Return the picker object used by this artist

get_pickradius()

get_rasterized()
return True if the artist is to be rasterized

get_sketch_params()
Returns the sketch parameters for the artist.
**Returns sketch_params**: tuple or None

A 3-tuple with the following elements:

- **scale**: The amplitude of the wiggle perpendicular to the source line.
- **length**: The length of the wiggle along the line.
- **randomness**: The scale factor by which the length is shrunken or expanded.

May return None if no sketch parameters were set.

**get_snap**

Returns the snap setting which may be:

- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

**get_transform**

Return the Transform instance used by this artist.

**get_transformed_clip_path_and_affine**

Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

**get_transforms**

**get_url**

Returns the url

**get_urls**

**get_visible**

Return the artist’s visibility

**get_window_extent**(renderer)

**get_zorder**

Return the Artist’s zorder.

**have_units**

Return True if units are set on the x or y axes

**hitlist**(event)

List the children of the artist which contain the mouse event event.

**is_figure_set**

Returns True if the artist is assigned to a Figure.
is_transform_set()
    Returns True if Artist has a transform explicitly set.

mouseover

pchanged()
    Fire an event when property changed, calling all of the registered callbacks.

pick(mouseevent)
    Process pick event
    each child artist will fire a pick event if mouseevent is over the artist and the artist has picker set

pickable()
    Return True if Artist is pickable.

properties()
    return a dictionary mapping property name -> value for all Artist props

remove()
    Remove the artist from the figure if possible. The effect will not be visible until the figure is
    redrawn, e.g., with matplotlib.axes.Axes.draw_idle(). Call matplotlib.axes.Axes.relim() to update the axes limits if desired.
    Note: relim() will not see collections even if the collection was added to axes with autolim = True.
    Note: there is no support for removing the artist’s legend entry.

remove_callback(oid)
    Remove a callback based on its id.

    See also:

    add_callback() For adding callbacks

set(**kwargs)
    A property batch setter. Pass kwargs to set properties.

set_agg_filter(filter_func)
    set agg_filter function.

set_alpha(alpha)
    Set the alpha transparencies of the collection. alpha must be a float or None.
    ACCEPTS: float or None

set_animated(b)
    Set the artist’s animation state.
    ACCEPTS: [True | False]

set_antialiased(aa)
    Set the antialiasing state for rendering.
    ACCEPTS: Boolean or sequence of booleans
set_antialiaseds(aa)
    alias for set_antialiased

set_array(A)
    Set the image array from numpy array A

set_axes(axes)
    Set the Axes instance in which the artist resides, if any.
    This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.
    ACCEPTS: an Axes instance

set_clim(vmin=None, vmax=None)
    set the norm limits for image scaling; if vmin is a length2 sequence, interpret it as (vmin, vmax) which is used to support setp
    ACCEPTS: a length 2 sequence of floats

set_clip_box(clipbox)
    Set the artist’s clip Bbox.
    ACCEPTS: a matplotlib.transforms.Bbox instance

set_clip_on(b)
    Set whether artist uses clipping.
    When False artists will be visible out side of the axes which can lead to unexpected results.
    ACCEPTS: [True | False]

set_clip_path(path, transform=None)
    Set the artist’s clip path, which may be:
        ● a Patch (or subclass) instance
        ● a Path instance, in which case an optional Transform instance may be provided, which will be applied to the path before using it for clipping.
        ● None, to remove the clipping path
    For efficiency, if the path happens to be an axis-aligned rectangle, this method will set the clipping box to the corresponding rectangle and set the clipping path to None.
    ACCEPTS: [ (Path, Transform) | Patch | None ]

set_cmap(cmap)
    set the colormap for luminance data
    ACCEPTS: a colormap or registered colormap name

set_color(c)
    Set both the edgecolor and the facecolor.
    ACCEPTS: matplotlib color arg or sequence of rgba tuples
    See also:

    set_facecolor(), set_edgecolor() For setting the edge or face color individually.
**set_contains** *(picker)*
Replace the contains test used by this artist. The new picker should be a callable function which determines whether the artist is hit by the mouse event:

```python
hit, props = picker(artist, mouseevent)
```

If the mouse event is over the artist, return *hit* = *True* and *props* is a dictionary of properties you want returned with the contains test.

**ACCEPTS:** a callable function

**set_dashes** *(ls)*
alias for *set_linestyle*

**set_edgecolor** *(c)*
Set the edgecolor(s) of the collection. *c* can be a matplotlib color spec (all patches have same color), or a sequence of specs; if it is a sequence the patches will cycle through the sequence.

If *c* is ‘face’, the edge color will always be the same as the face color. If it is ‘none’, the patch boundary will not be drawn.

**ACCEPTS:** matplotlib color spec or sequence of specs

**set_edgecolors** *(c)*
alias for *set_edgecolor*

**set_facecolor** *(c)*
Set the facecolor(s) of the collection. *c* can be a matplotlib color spec (all patches have same color), or a sequence of specs; if it is a sequence the patches will cycle through the sequence.

If *c* is ‘none’, the patch will not be filled.

**ACCEPTS:** matplotlib color spec or sequence of specs

**set_facecolors** *(c)*
alias for *set_facecolor*

**set_figure** *(fig)*
Set the *Figure* instance the artist belongs to.

**ACCEPTS:** a *matplotlib.figure.Figure* instance

**set_gid** *(gid)*
Sets the (group) id for the artist

**ACCEPTS:** an id string

**set_hatch** *(hatch)*
Set the hatching pattern

*hatch* can be one of:

```
/  - diagonal hatching
\  - back diagonal
|  - vertical
-  - horizontal
```
Letters can be combined, in which case all the specified hatchings are done. If same letter
repeats, it increases the density of hatching of that pattern.

Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Unlike other properties such as linewidth and colors, hatching can only be specified for the
collection as a whole, not separately for each member.

**ACCEPTS:** [ ‘/’ | ‘\’ | ‘|’ | ‘-.’ | ‘+’ | ‘x’ | ‘o’ | ‘O’ | ‘.’ | ‘*’ ]

**set_label**(s)

Set the label to s for auto legend.

**ACCEPTS:** string or anything printable with ‘%s’ conversion.

**set_linestyle**(ls)

Set the linestyle(s) for the collection.

<table>
<thead>
<tr>
<th>linestyle</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘-‘ or ‘solid’</td>
<td>solid line</td>
</tr>
<tr>
<td>‘--‘ or ‘dashed’</td>
<td>dashed line</td>
</tr>
<tr>
<td>‘.-‘ or ‘dashdot‘</td>
<td>dash-dotted line</td>
</tr>
<tr>
<td>‘:’ or ‘dotted‘</td>
<td>dotted line</td>
</tr>
</tbody>
</table>

Alternatively a dash tuple of the following form can be provided:

```
(offset, onoffseq),
```

where onoffseq is an even length tuple of on and off ink in points.

**ACCEPTS:** [‘solid’ | ‘dashed’, ‘dashdot’, ‘dotted’ | (offset, on-off-dash-seq) | ‘-‘ | ‘--‘ | ‘.-‘ | ‘:’ | ‘None’ | ’ ’ | ’ ’ ]

**Parameters ls :** { ‘-‘, ‘--‘, ‘.-‘, ‘:‘ } and more see description

The line style.

**set_linestyles**(ls)

alias for set_linestyle

**set_linewidth**(lw)

Set the linewidth(s) for the collection. lw can be a scalar or a sequence; if it is a sequence the
patches will cycle through the sequence

**ACCEPTS:** float or sequence of floats

**set_linewidths**(lw)

alias for set_linewidth
**set_lw**(lw)

alias for set_linewidth

**set_norm**(norm)

set the normalization instance

**set_offset_position**(offset_position)

Set how offsets are applied. If offset_position is ‘screen’ (default) the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is ‘data’, the offset is applied before the master transform, i.e., the offsets are in data coordinates.

**set_offsets**(offsets)

Set the offsets for the collection. offsets can be a scalar or a sequence.

ACCEPTS: float or sequence of floats

**set_path_effects**(path_effects)

set path_effects, which should be a list of instances of matplotlib.pathEffects._Base class or its derivatives.

**set_paths()**

**set_picker**(picker)

Set the epsilon for picking used by this artist

picker can be one of the following:

- *None*: picking is disabled for this artist (default)
- A boolean: if True then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist
- A float: if picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if it’s data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event
- A function: if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```python
hit, props = picker(artist, mouseevent)
```

to determine the hit test. if the mouse event is over the artist, return hit=True and props is a dictionary of properties you want added to the PickEvent attributes.

ACCEPTS: [None|float|boolean|callable]

**set_pickradius**(pr)

**set_rasterized**(rasterized)

Force rasterized (bitmap) drawing in vector backend output.

Defaults to None, which implies the backend’s default behavior
**set_sketch_params**(
  *scale=None, length=None, randomness=None*)

Sets the sketch parameters.

**Parameters**

* scale: float, optional
  The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is `None`, or not provided, no sketch filter will be provided.

* length: float, optional
  The length of the wiggle along the line, in pixels (default 128.0)

* randomness: float, optional
  The scale factor by which the length is shrunken or expanded (default 16.0)

**set_snap**(*snap*)

Sets the snap setting which may be:

- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

**set_transform**(*t*)

Set the `Transform` instance used by this artist.

ACCEPETS: `Transform` instance

**set_url**(*url*)

Sets the url for the artist

ACCEPETS: a url string

**set_urls**(*urls*)

**set_visible**(*b*)

Set the artist’s visibility.

ACCEPETS: [True | False]

**set_zorder**(*level*)

Set the zorder for the artist. Artists with lower zorder values are drawn first.

ACCEPETS: any number

**stale**

If the artist is ‘stale’ and needs to be re-drawn for the output to match the internal state of the artist.

**sticky_edges**

x and y sticky edge lists.
When performing autoscaling, if a data limit coincides with a value in the corresponding sticky_edges list, then no margin will be added—the view limit “sticks” to the edge. A typical usecase is histograms, where one usually expects no margin on the bottom edge (0) of the histogram.

This attribute cannot be assigned to; however, the x and y lists can be modified in place as needed.

**Examples**

```python
>>> artist.sticky_edges.x[:] = (xmin, xmax)
>>> artist.sticky_edges.y[:] = (ymin, ymax)
```

to_rgba(x, alpha=None, bytes=False, norm=True)

Return a normalized rgba array corresponding to x.

In the normal case, x is a 1-D or 2-D sequence of scalars, and the corresponding ndarray of rgba values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already rgb or rgba, such as might have been read from an image file. If x is an ndarray with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an rgb or rgba array, and no mapping will be done. If the last dimension is 3, the alpha kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the alpha kwarg is ignored; it does not replace the pre-existing alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if bytes is False (default), the rgba array will be floats in the 0-1 range; if it is True, the returned rgba array will be uint8 in the 0 to 255 range.

If norm is False, no normalization of the input data is performed, and it is assumed to already be in the range (0-1).

Note: this method assumes the input is well-behaved; it does not check for anomalies such as x being a masked rgba array, or being an integer type other than uint8, or being a floating point rgba array with values outside the 0-1 range.

**update**

Update the properties of this Artist from the dictionary prop.

**update_from**

copy properties from other to self

**update_scalarmappable**

If the scalar mappable array is not none, update colors from scalar data

zorder = 0

class matplotlib.collections.EllipseCollection(widths, heights, angles, units='points', **kwargs)

Bases: matplotlib.collections.Collection

A collection of ellipses, drawn using splines.
**widths:** sequence lengths of first axes (e.g., major axis lengths)

**heights:** sequence lengths of second axes

**angles:** sequence angles of first axes, degrees CCW from the X-axis

**units:** ['points' | 'inches' | 'dots' | 'width' | 'height' | 'x' | 'y' | 'xy']

units in which majors and minors are given; ‘width’ and ‘height’ refer to the dimensions of the axes, while ‘x’ and ‘y’ refer to the offsets data units. ‘xy’ differs from all others in that the angle as plotted varies with the aspect ratio, and equals the specified angle only when the aspect ratio is unity. Hence it behaves the same as the *Ellipse* with axes.transData as its transform.

Additional kwargs inherited from the base *Collection*:

Valid Collection keyword arguments:

- *edgecolors*: None
- *facecolors*: None
- *linewidths*: None
- *antialiaseds*: None
- *offsets*: None
- *transOffset*: transforms.IdentityTransform()
- *norm*: None (optional for *matplotlib.cm.ScalarMappable*)
- *cmap*: None (optional for *matplotlib.cm.ScalarMappable*)

*offsets* and *transOffset* are used to translate the patch after rendering (default no offsets)

If any of *edgecolors*, *facecolors*, *linewidths*, *antialiaseds* are None, they default to their *matplotlib.rcParams* patch setting, in sequence form.

**add_callback(func)**

Adds a callback function that will be called whenever one of the *Artist*’s properties changes.

Returns an id that is useful for removing the callback with *remove_callback()* later.

**add_checker(checker)**

Add an entry to a dictionary of boolean flags that are set to True when the mappable is changed.

**aname = ‘Artist’**

**autoscale()**

Autoscale the scalar limits on the norm instance using the current array

**autoscale_None()**

Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

**axes**

The *Axes* instance the artist resides in, or *None*.
changed()
    Call this whenever the mappable is changed to notify all the callback listeners to the 'changed' signal

check_update(checker)
    If mappable has changed since the last check, return True; else return False

contains(mouseevent)
    Test whether the mouse event occurred in the collection.
    Returns True | False, dict(ind=itemlist), where every item in itemlist contains the event.

convert_xunits(x)
    For artists in an axes, if the xaxis has units support, convert x using xaxis unit type

convert_yunits(y)
    For artists in an axes, if the yaxis has units support, convert y using yaxis unit type

draw(artist, renderer, *args, **kwargs)

findobj(match=None, include_self=True)
    Find artist objects.
    Recursively find all Artist instances contained in self.
    match can be
        • None: return all objects contained in artist.
        • function with signature boolean = match(artist) used to filter matches
        • class instance: e.g., Line2D. Only return artists of class type.
    If include_self is True (default), include self in the list to be checked for a match.

format_cursor_data(data)
    Return cursor data string formatted.

get_agg_filter()
    return filter function to be used for agg filter

get_alpha()
    Return the alpha value used for blending - not supported on all backends

get_animated()
    Return the artist’s animated state

get_array()
    Return the array

get_axes()
    Return the Axes instance the artist resides in, or None.
    This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

get_children()
    Return a list of the child Artist's this :class:`Artist` contains.
get_clim()
    return the min, max of the color limits for image scaling

get_clip_box()
    Return artist clipbox

get_clip_on()
    Return whether artist uses clipping

get_clip_path()
    Return artist clip path

get_cmap()
    return the colormap

get_contains()
    Return the _contains test used by the artist, or None for default.

get_cursor_data(event)
    Get the cursor data for a given event.

get_dashes()

get_datalim(transData)

get_edgecolor()

get_edgecolors()

get_facecolor()

get_facecolors()

get_figure()
    Return the Figure instance the artist belongs to.

get_fill()
    return whether fill is set

get_gid()
    Returns the group id

get_hatch()
    Return the current hatching pattern

get_label()
    Get the label used for this artist in the legend.

get_linestyle()
get_linestyles()
get_linewidth()
get_linewidths()

get_offset_position()
Returns how offsets are applied for the collection. If offset_position is ‘screen’, the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is ‘data’, the offset is applied before the master transform, i.e., the offsets are in data coordinates.

get_offset_transform()

get_offsets()
Return the offsets for the collection.

get_path_effects()

get_paths()

get_picker()
Return the picker object used by this artist

get_pickradius()

get_rasterized()
return True if the artist is to be rasterized

get_sketch_params()
Returns the sketch parameters for the artist.

Returns sketch_params : tuple or None
A 3-tuple with the following elements:
• scale: The amplitude of the wiggle perpendicular to the source line.
• length: The length of the wiggle along the line.
• randomness: The scale factor by which the length is shrunken or expanded.
May return None if no sketch parameters were set.

get_snap()
Returns the snap setting which may be:
• True: snap vertices to the nearest pixel center
• False: leave vertices as-is
get_transform()  
Return the Transform instance used by this artist.

get_transformed_clip_path_and_affine()  
Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

get_transforms()  

get_url()  
Returns the url

get_urls()  

get_visible()  
Return the artist's visibility

get_window_extent(renderer)  

get_zorder()  
Return the Artist's zorder.

have_units()  
Return True if units are set on the x or y axes

hitlist(event)  
List the children of the artist which contain the mouse event event.

is_figure_set()  
Returns True if the artist is assigned to a Figure.

is_transform_set()  
Returns True if Artist has a transform explicitly set.

mouseover

pchanged()  
Fire an event when property changed, calling all of the registered callbacks.

pick(mouseevent)  
Process pick event
    each child artist will fire a pick event if mouseevent is over the artist and the artist has picker set

pickable()  
Return True if Artist is pickable.
properties()
    return a dictionary mapping property name -> value for all Artist props

remove()
    Remove the artist from the figure if possible. The effect will not be visible until the figure is
    redrawn, e.g., with matplotlib.axes.Axes.draw_idle(). Call matplotlib.axes.Axes.relim() to update the axes limits if desired.

    Note: relim() will not see collections even if the collection was added to axes with autolim =
    True.

    Note: there is no support for removing the artist’s legend entry.

remove_callback(oid)
    Remove a callback based on its id.

    See also:

        add_callback() For adding callbacks

set(**kwargs)
    A property batch setter. Pass kwargs to set properties.

set_agg_filter(filter_func)
    set agg_filter function.

set_alpha(alpha)
    Set the alpha transparencies of the collection. alpha must be a float or None.

    ACCEPTS: float or None

set_animated(b)
    Set the artist’s animation state.

    ACCEPTS: [True | False]

set_antialiased(aa)
    Set the antialiasing state for rendering.

    ACCEPTS: Boolean or sequence of booleans

set_antialiaseds(aa)
    alias for set_antialiased

set_array(A)
    Set the image array from numpy array A

set_axes(axes)
    Set the Axes instance in which the artist resides, if any.

    This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

    ACCEPTS: an Axes instance

set_clim(vmin=None, vmax=None)
    set the norm limits for image scaling; if vmin is a length2 sequence, interpret it as (vmin,
    vmax) which is used to support setp
set_clip_box(clipbox)
Set the artist’s clip Bbox.

set_clip_on(b)
Set whether artist uses clipping.
When False artists will be visible out side of the axes which can lead to unexpected results.

set_clip_path(path, transform=None)
Set the artist’s clip path, which may be:

- a Patch (or subclass) instance
- a Path instance, in which case an optional Transform instance may be provided, which will be applied to the path before using it for clipping.
- None, to remove the clipping path

For efficiency, if the path happens to be an axis-aligned rectangle, this method will set the clipping box to the corresponding rectangle and set the clipping path to None.

set_cmap(cmap)
set the colormap for luminance data

set_color(c)
Set both the edgecolor and the facecolor.

set_contains(picker)
Replace the contains test used by this artist. The new picker should be a callable function which determines whether the artist is hit by the mouse event:

```python
hit, props = picker(artist, mouseevent)
```

If the mouse event is over the artist, return hit = True and props is a dictionary of properties you want returned with the contains test.

set_dashes(ls)
alias for set_linestyle
**set.edgecolor()**
Set the edgecolor(s) of the collection. $c$ can be a matplotlib color spec (all patches have same color), or a sequence of specs; if it is a sequence the patches will cycle through the sequence.

If $c$ is ‘face’, the edge color will always be the same as the face color. If it is ‘none’, the patch boundary will not be drawn.

ACCEPTS: matplotlib color spec or sequence of specs

**set.edgecolors()**
alias for set.edgecolor

**set.facecolor()**
Set the facecolor(s) of the collection. $c$ can be a matplotlib color spec (all patches have same color), or a sequence of specs; if it is a sequence the patches will cycle through the sequence.

If $c$ is ‘none’, the patch will not be filled.

ACCEPTS: matplotlib color spec or sequence of specs

**set.facecolors()**
alias for set.facecolor

**set.figure()**
Set the Figure instance the artist belongs to.

ACCEPTS: a matplotlib.figure.Figure instance

**set_gid()**
Sets the (group) id for the artist

ACCEPTS: an id string

**set.hatch()**
Set the hatching pattern

$hatch$ can be one of:

```
/  - diagonal hatching
\  - back diagonal
|  - vertical
-  - horizontal
+  - crossed
x  - crossed diagonal
o  - small circle
0  - large circle
.  - dots
*  - stars
```

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.
ACCEPES: [ ‘/’ | ‘\’ | ‘|’ | ‘-’ | ‘+’ | ‘x’ | ‘o’ | ‘O’ | ‘.’ | ‘*’ ]

set_label(s)
Set the label to \( s \) for auto legend.
ACCEPES: string or anything printable with ‘\%s’ conversion.

set_linestyle(ls)
Set the linestyle(s) for the collection.

<table>
<thead>
<tr>
<th>linestyle</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘-‘ or ‘solid’</td>
<td>solid line</td>
</tr>
<tr>
<td>‘-.-’ or ‘dashed’</td>
<td>dashed line</td>
</tr>
<tr>
<td>‘-.’ or ‘dashdot’</td>
<td>dash-dotted line</td>
</tr>
<tr>
<td>‘:’ or ‘dotted’</td>
<td>dotted line</td>
</tr>
</tbody>
</table>

Alternatively a dash tuple of the following form can be provided:

\[(offset, onoffseq),\]

where onoffseq is an even length tuple of on and off ink in points.

ACCEPES: [‘solid’ | ‘dashed’, ‘dashdot’, ‘dotted’ | (offset, on-off-dash-seq) | ‘-‘ | ‘-.-’ | ‘-.’ | ‘:’ | ‘None’ | ‘‘ ‘‘ ‘‘ ‘‘]

Parameters ls : { ‘-‘, ‘-.-’, ‘-.’, ‘:’ } and more see description

The line style.

set_linestyles(ls)
alias for set_linestyle

set_linewidth(lw)
Set the linewidth(s) for the collection. \( lw \) can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence
ACCEPES: float or sequence of floats

set_linewidths(lw)
alias for set_linewidth

set_lw(lw)
alias for set_linewidth

set_norm(norm)
set the normalization instance

set_offset_position(offset_position)
Set how offsets are applied. If offset_position is ‘screen’ (default) the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is ‘data’, the offset is applied before the master transform, i.e., the offsets are in data coordinates.

set_offsets(offsets)
Set the offsets for the collection. offsets can be a scalar or a sequence.
ACCEPTS: float or sequence of floats

```
set_path_effects(path_effects)
```

Set path_effects, which should be a list of instances of matplotlib.pathEffects.PathEffects class or its derivatives.

```
set_paths()
```

```
set_picker(picker)
```

Set the epsilon for picking used by this artist

`picker` can be one of the following:

- `None`: picking is disabled for this artist (default)
- A boolean: if `True` then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist
- A float: if picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if it’s data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event
- A function: if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

to determine the hit test. if the mouse event is over the artist, return `hit=True` and props is a dictionary of properties you want added to the PickEvent attributes.

ACCEPTS: [None|float|boolean|callable]

```
set_pickradius(pr)
```

```
set_rasterized(rasterized)
```

Force rasterized (bitmap) drawing in vector backend output.

Defaults to `None`, which implies the backend’s default behavior

ACCEPTS: [True | False | None]

```
set_sketch_params(scale=None, length=None, randomness=None)
```

Sets the sketch parameters.

**Parameters**

- `scale` : float, optional
  The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is `None`, or not provided, no sketch filter will be provided.

- `length` : float, optional
  The length of the wiggle along the line, in pixels (default 128.0)

- `randomness` : float, optional
The scale factor by which the length is shrunken or expanded (default 16.0)

**set_snap**(snap)
Sets the snap setting which may be:

- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

**set_transform**(t)
Set the Transform instance used by this artist.

ACCEPTS: Transform instance

**set_url**(url)
Sets the url for the artist

ACCEPTS: a url string

**set_urls**(urls)

**set_visible**(b)
Set the artist’s visiblity.

ACCEPTS: [True | False]

**set_zorder**(level)
Set the zorder for the artist. Artists with lower zorder values are drawn first.

ACCEPTS: any number

**stale**
If the artist is ‘stale’ and needs to be re-drawn for the output to match the internal state of the artist.

**sticky_edges**

x and y sticky edge lists.

When performing autoscaling, if a data limit coincides with a value in the corresponding sticky_edges list, then no margin will be added—the view limit “sticks” to the edge. A typical usecase is histograms, where one usually expects no margin on the bottom edge (0) of the histogram.

This attribute cannot be assigned to; however, the x and y lists can be modified in place as needed.
Examples

```python
>>> artist.sticky_edges.x[:] = (xmin, xmax)
>>> artist.sticky_edges.y[:] = (ymin, ymax)
```

to_rgba(x, alpha=None, bytes=False, norm=True)

Return a normalized rgba array corresponding to x.

In the normal case, x is a 1-D or 2-D sequence of scalars, and the corresponding ndarray of rgba values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already rgb or rgba, such as might have been read from an image file. If x is an ndarray with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an rgb or rgba array, and no mapping will be done. If the last dimension is 3, the alpha kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the alpha kwarg is ignored; it does not replace the pre-existing alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if bytes is False (default), the rgba array will be floats in the 0-1 range; if it is True, the returned rgba array will be uint8 in the 0 to 255 range.

If norm is False, no normalization of the input data is performed, and it is assumed to already be in the range (0-1).

Note: this method assumes the input is well-behaved; it does not check for anomalies such as x being a masked rgba array, or being an integer type other than uint8, or being a floating point rgba array with values outside the 0-1 range.

update(props)

Update the properties of this Artist from the dictionary prop.

update_from(other)

copy properties from other to self

update_scalarmappable()

If the scalar mappable array is not None, update colors from scalar data

zorder = 0

class matplotlib.collections.EventCollection(positions, orientation=None, lineoffset=0, linelength=1, linewidth=None, color=None, linestyle='solid', antialiased=None, **kwargs)

Bases: matplotlib.collections.LineCollection

A collection of discrete events.

An event is a 1-dimensional value, usually the position of something along an axis, such as time or length. Events do not have an amplitude. They are displayed as v

positions a sequence of numerical values or a 1D numpy array. Can be None

orientation [ ‘horizontal’ | ‘vertical’ | None ] defaults to ‘horizontal’ if not specified or None
lineoffset a single numerical value, corresponding to the offset of the center of the markers from the origin

linelength a single numerical value, corresponding to the total height of the marker (i.e. the marker stretches from lineoffset+linelength/2 to lineoffset-linelength/2). Defaults to 1

linewidth a single numerical value

color must be a sequence of RGBA tuples (e.g., arbitrary color strings, etc, not allowed).

linestyle [ ‘solid’ | ‘dashed’ | ‘dashdot’ | ‘dotted’ ]

antialiased 1 or 2

If linewidth, color, or antialiased is None, they default to their rcParams setting, in sequence form.

norm None (optional for matplotlib.cm.ScalarMappable)

cmap None (optional for matplotlib.cm.ScalarMappable)

pickradius is the tolerance for mouse clicks picking a line. The default is 5 pt.

The use of ScalarMappable is optional. If the ScalarMappable array _A is not None (i.e., a call to set_array() has been made), at draw time a call to scalar mappable will be made to set the colors.

Example:

```
add_callback(func)
```

Adds a callback function that will be called whenever one of the Artist’s properties changes.
Returns an id that is useful for removing the callback with \texttt{remove\_callback()} later.

\textbf{add\_checker}(checker)

Add an entry to a dictionary of boolean flags that are set to True when the mappable is changed.

\textbf{add\_positions}(position)

\begin{itemize}
\item add one or more events at the specified positions
\end{itemize}

\textbf{aname = ‘Artist’}

\textbf{append\_positions}(position)

\begin{itemize}
\item add one or more events at the specified positions
\end{itemize}

\textbf{autoscale}()

Autoscale the scalar limits on the norm instance using the current array

\textbf{autoscale\_None}()

Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

\textbf{axes}

The \textit{Axes} instance the artist resides in, or \textit{None}.

\textbf{changed}()

Call this whenever the mappable is changed to notify all the callback listeners to the ‘changed’ signal

\textbf{check\_update}(checker)

If mappable has changed since the last check, return True; else return False

\textbf{contains}(mouseevent)

Test whether the mouse event occurred in the collection.

\begin{itemize}
\item Returns True | False, dict(ind=itemlist), where every item in itemlist contains the event.
\end{itemize}

\textbf{convert\_xunits}(x)

For artists in an axes, if the xaxis has units support, convert \(x\) using xaxis unit type

\textbf{convert\_yunits}(y)

For artists in an axes, if the yaxis has units support, convert \(y\) using yaxis unit type

\textbf{draw}(artist, renderer, *args, **kwargs)

\textbf{extend\_positions}(position)

\begin{itemize}
\item add one or more events at the specified positions
\end{itemize}

\textbf{findobj}(match=\texttt{None}, include\_self=\texttt{\textit{True}})

Find artist objects.

\begin{itemize}
\item Recursively find all Artist instances contained in self.
\item match can be
\begin{itemize}
\item None: return all objects contained in artist.
\item function with signature boolean = match(artist) used to filter matches
\end{itemize}
\end{itemize}
class instance: e.g., Line2D. Only return artists of class type.

If include_self is True (default), include self in the list to be checked for a match.

```
format_cursor_data(data)
    Return cursor data string formatted.
```

```
get_agg_filter()
    return filter function to be used for agg filter
```

```
get_alpha()
    Return the alpha value used for blending - not supported on all backends
```

```
get_animated()
    Return the artist’s animated state
```

```
get_array()
    Return the array
```

```
get_axes()
    Return the Axes instance the artist resides in, or None.
    This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.
```

```
get_children()
    Return a list of the child Artist’s this :class:`Artist` contains.
```

```
get_clim()
    return the min, max of the color limits for image scaling
```

```
get_clip_box()
    Return artist clipbox
```

```
get_clip_on()
    Return whether artist uses clipping
```

```
get_clip_path()
    Return artist clip path
```

```
get_cmap()
    return the colormap
```

```
get_color()
    get the color of the lines used to mark each event
```

```
get_colors()
```

```
get_contains()
    Return the _contains test used by the artist, or None for default.
```

```
get_cursor_data(event)
    Get the cursor data for a given event.
```

```
get_dashes()
```
get_datalim(transData)

get_edgecolor()

get_edgecolors()

get_facecolor()

get_facecolors()

get_figure()
    Return the Figure instance the artist belongs to.

get_fill()
    return whether fill is set

get_gid()
    Returns the group id

get_hatch()
    Return the current hatching pattern

get_label()
    Get the label used for this artist in the legend.

get_linelength()
    get the length of the lines used to mark each event

get_lineoffset()
    get the offset of the lines used to mark each event

get_linestyle()
    get the style of the lines used to mark each event [‘solid’ | ‘dashed’ | ‘dashdot’ | ‘dotted’]

get_linestyles()

get_linewidth()
    get the width of the lines used to mark each event

get_linewidths()

get_offset_position()
    Returns how offsets are applied for the collection. If offset_position is ‘screen’, the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is ‘data’, the offset is applied before the master transform, i.e., the offsets are in data coordinates.

get_offset_transform()
get_offsets()  
Return the offsets for the collection.

get_orientation()  
get the orientation of the event line, may be: ['horizontal' | 'vertical']

get_path_effects()  

get_paths()  

get_picker()  
Return the picker object used by this artist

get_pickradius()  

get_positions()  
return an array containing the floating-point values of the positions

get_rasterized()  
return True if the artist is to be rasterized

get_segments()  

get_sketch_params()  
Returns the sketch parameters for the artist.

    Returns sketch_params : tuple or None

    A 3-tuple with the following elements:

    • scale: The amplitude of the wiggle perpendicular to the source line.
    • length: The length of the wiggle along the line.
    • randomness: The scale factor by which the length is shrunken or expanded.

    May return None if no sketch parameters were set.

get_snap()  
Returns the snap setting which may be:

    • True: snap vertices to the nearest pixel center
    • False: leave vertices as-is
    • None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

    Only supported by the Agg and MacOSX backends.

get_transform()  
Return the Transform instance used by this artist.
get_transformed_clip_path_and_affine()  
Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

get_transforms()

get_url()  
Returns the url

get_urls()

get_visible()  
Return the artist’s visibility

get_window_extent(renderer)

get_zorder()  
Return the Artist's zorder.

have_units()  
Return True if units are set on the x or y axes

hitlist(event)  
List the children of the artist which contain the mouse event event.

is_figure_set()  
Returns True if the artist is assigned to a Figure.

is_horizontal()  
True if the eventcollection is horizontal, False if vertical

is_transform_set()  
Returns True if Artist has a transform explicitly set.

mouseover

pchanged()  
Fire an event when property changed, calling all of the registered callbacks.

pick(mouseevent)  
Process pick event  
each child artist will fire a pick event if mouseevent is over the artist and the artist has picker set

pickable()  
Return True if Artist is pickable.

properties()  
return a dictionary mapping property name -> value for all Artist props

remove()  
Remove the artist from the figure if possible. The effect will not be visible until the figure is
redrawn, e.g., with `matplotlib.axes.Axes.draw_idle()`. Call `matplotlib.axes.Axes.relim()` to update the axes limits if desired.

Note: `relim()` will not see collections even if the collection was added to axes with `autolim = True`.

Note: there is no support for removing the artist’s legend entry.

**remove_callback**(oid)

Remove a callback based on its id.

See also:

**add_callback()** For adding callbacks

**set(****kwargs**)**

A property batch setter. Pass `kwargs` to set properties.

**set_agg_filter**(filter_func)

Set agg_filter function.

**set_alpha**(alpha)

Set the alpha transparencies of the collection. `alpha` must be a float or `None`

ACCEPTS: float or None

**set_animated**(b)

Set the artist’s animation state.

ACCEPTS: [True | False]

**set_antialiased**(aa)

Set the antialiasing state for rendering.

ACCEPTS: Boolean or sequence of booleans

**set_antialiaseds**(aa)

alias for `set_antialiased`

**set_array**(A)

Set the image array from numpy array `A`

**set_axes**(axes)

Set the `Axes` instance in which the artist resides, if any.

This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

ACCEPTS: an `Axes` instance

**set_clim**(vmin=None, vmax=None)

Set the norm limits for image scaling; if `vmin` is a length2 sequence, interpret it as `(vmin, vmax)` which is used to support setp

ACCEPTS: a length 2 sequence of floats

**set_clip_box**(clipbox)

Set the artist’s clip `Bbox`. 

Chapter 24. collections


ACCEP'TS: a `matplotlib.transforms.Bbox` instance

**set_clip_on***(b)**

Set whether artist uses clipping.

When False artists will be visible out side of the axes which can lead to unexpected results.

ACCEP'TS: [True | False]

**set_clip_path**(path, transform=None)

Set the artist’s clip path, which may be:

- a `Patch` (or subclass) instance
- a `Path` instance, in which case an optional `Transform` instance may be provided, which will be applied to the path before using it for clipping.
- `None`, to remove the clipping path

For efficiency, if the path happens to be an axis-aligned rectangle, this method will set the clipping box to the corresponding rectangle and set the clipping path to `None`.

ACCEP'TS: `[ (Path, Transform) | Patch | None ]`

**set_cmap**(cmap)

set the colormap for luminance data

ACCEP'TS: a colormap or registered colormap name

**set_color***(c)**

Set the color(s) of the line collection. `c` can be a matplotlib color arg (all patches have same color), or a sequence or rgba tuples; if it is a sequence the patches will cycle through the sequence.

ACCEP'TS: matplotlib color arg or sequence of rgba tuples

**set_contains**(picker)

Replace the contains test used by this artist. The new picker should be a callable function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

If the mouse event is over the artist, return `hit = True` and `props` is a dictionary of properties you want returned with the contains test.

ACCEP'TS: a callable function

**set_dashes**(ls)

alias for set_linestyle

**set_edgecolor***(c)**

Set the edgecolor(s) of the collection. `c` can be a matplotlib color spec (all patches have same color), or a sequence of specs; if it is a sequence the patches will cycle through the sequence.

If `c` is ‘face’, the edge color will always be the same as the face color. If it is ‘none’, the patch boundary will not be drawn.
ACCEPTEs: matplotlib color spec or sequence of specs

**set_edgecolors**

```
(c)
```

alias for set_edgecolor

**set_facecolor**

```
(c)
```

Set the facecolor(s) of the collection. `c` can be a matplotlib color spec (all patches have same color), or a sequence of specs; if it is a sequence the patches will cycle through the sequence.

If `c` is ‘none’, the patch will not be filled.

ACCEPTEs: matplotlib color spec or sequence of specs

**set_facecolors**

```
(c)
```

alias for set_facecolor

**set_figure**

```
(fig)
```

Set the `Figure` instance the artist belongs to.

ACCEPTEs: a `matplotlib.figure.Figure` instance

**set_gid**

```
(gid)
```

Sets the (group) id for the artist

ACCEPTEs: an id string

**set_hatch**

```
(hatch)
```

Set the hatching pattern

`hatch` can be one of:

```
/  - diagonal hatching
\  - back diagonal
|  - vertical
-  - horizontal
+  - crossed
x  - crossed diagonal
o  - small circle
O  - large circle
.  - dots
*  - stars
```

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

ACCEPTEs: `[ / | \ | | - - | + | x | o | O | . | * ]`

**set_label**

```
(s)
```

Set the label to `s` for auto legend.

ACCEPTEs: string or anything printable with ‘%s’ conversion.
**set_linelength**(linelength)
set the length of the lines used to mark each event

**set_lineoffset**(lineoffset)
set the offset of the lines used to mark each event

**set_linestyle**(ls)
Set the linestyle(s) for the collection.

<table>
<thead>
<tr>
<th>linestyle</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>' - ' or 'solid'</td>
<td>solid line</td>
</tr>
<tr>
<td>' -- ' or 'dashed'</td>
<td>dashed line</td>
</tr>
<tr>
<td>' -. ' or 'dashdot'</td>
<td>dash-dotted line</td>
</tr>
<tr>
<td>':' or 'dotted'</td>
<td>dotted line</td>
</tr>
</tbody>
</table>

Alternatively a dash tuple of the following form can be provided:

```
(offset, onoffseq),
```

where `onoffseq` is an even length tuple of on and off ink in points.

**ACCEPTS:** ['solid' | ‘dashed’, ‘dashdot’, ‘dotted’ | (offset, on-off-dash-seq) | ' - ' | ' -- ' | ' -. ' | ':' | 'None' | ' ' | ' ']

**Parameters** *ls* : { ‘-', ‘-.’, ‘--’, ‘:’} and more see description

The line style.

**set_linestyles**(ls)
alias for `set_linestyle`

**set_linewidth**(lw)
Set the linewidth(s) for the collection. `lw` can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence

**ACCEPTS:** float or sequence of floats

**set_linewidths**(lw)
alias for `set_linewidth`

**set_lw**(lw)
alias for `set_linewidth`

**set_norm**(norm)
set the normalization instance

**set_offset_position**(offset_position)
Set how offsets are applied. If `offset_position` is ‘screen’ (default) the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If `offset_position` is ‘data’, the offset is applied before the master transform, i.e., the offsets are in data coordinates.

**set_offsets**(offsets)
Set the offsets for the collection. `offsets` can be a scalar or a sequence.
set_orientation(orientation=None)
    set the orientation of the event line [‘horizontal’ | ‘vertical’ | None] defaults to ‘horizontal’ if not specified or None

set_path_effects(path_effects)
    set path_effects, which should be a list of instances of matplotlib.pathEffects._Base class or its derivatives.

set_paths(segments)

set_picker(picker)
    Set the epsilon for picking used by this artist

    picker can be one of the following:
    • None: picking is disabled for this artist (default)
    • A boolean: if True then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist
    • A float: if picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if it’s data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event
    • A function: if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

    ```
    hit, props = picker(artist, mouseevent)
    ```

    to determine the hit test. if the mouse event is over the artist, return hit=True and props is a dictionary of properties you want added to the PickEvent attributes.

    ACCEPTS: [None | float | boolean | callable]

set_pickradius(pr)

set_positions(positions)
    set the positions of the events to the specified value

set_rasterized(rasterized)
    Force rasterized (bitmap) drawing in vector backend output.
    Defaults to None, which implies the backend’s default behavior

    ACCEPTS: [True | False | None]

set_segments(segments)

set_sketch_params(scale=None, length=None, randomness=None)
    Sets the sketch parameters.
**Parameters**

**scale**: float, optional
The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is `None`, or not provided, no sketch filter will be provided.

**length**: float, optional
The length of the wiggle along the line, in pixels (default 128.0)

**randomness**: float, optional
The scale factor by which the length is shrunken or expanded (default 16.0)

**set_snap**(snap)
Sets the snap setting which may be:

* True: snap vertices to the nearest pixel center
* False: leave vertices as-is
* None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

**set_transform**(t)
Set the `Transform` instance used by this artist.

ACCEPSTS: `Transform` instance

**set_url**(url)
Sets the url for the artist

ACCEPSTS: a url string

**set_urls**(urls)

**set_verts**(segments)

**set_visible**(b)
Set the artist’s visibility.

ACCEPSTS: [True | False]

**set_zorder**(level)
Set the zorder for the artist. Artists with lower zorder values are drawn first.

ACCEPSTS: any number

**stale**
If the artist is ‘stale’ and needs to be re-drawn for the output to match the internal state of the artist.

**sticky_edges**
* x and y sticky edge lists.*
When performing autoscaling, if a data limit coincides with a value in the corresponding sticky_edges list, then no margin will be added—the view limit “sticks” to the edge. A typical usecase is histograms, where one usually expects no margin on the bottom edge (0) of the histogram.

This attribute cannot be assigned to; however, the x and y lists can be modified in place as needed.

**Examples**

```python
>>> artist.sticky_edges.x[:2] = (xmin, xmax)
>>> artist.sticky_edges.y[:2] = (ymin, ymax)
```

**switch_orientation()**
switch the orientation of the event line, either from vertical to horizontal or vice versus

**to_rgba(x, alpha=None, bytes=False, norm=True)**
Return a normalized rgba array corresponding to x.

In the normal case, x is a 1-D or 2-D sequence of scalars, and the corresponding ndarray of rgba values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already rgb or rgba, such as might have been read from an image file. If x is an ndarray with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an rgb or rgba array, and no mapping will be done. If the last dimension is 3, the alpha kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the alpha kwarg is ignored; it does not replace the pre-existing alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if bytes is False (default), the rgba array will be floats in the 0-1 range; if it is True, the returned rgba array will be uint8 in the 0 to 255 range.

If norm is False, no normalization of the input data is performed, and it is assumed to already be in the range (0-1).

Note: this method assumes the input is well-behaved; it does not check for anomalies such as x being a masked rgba array, or being an integer type other than uint8, or being a floating point rgba array with values outside the 0-1 range.

**update(props)**
Update the properties of this Artist from the dictionary prop.

**update_from(other)**
copy properties from other to self

**update_scalarmappable()**
If the scalar mappable array is not none, update colors from scalar data

**zorder = 0**
class matplotlib.collections.LineCollection(segments, lineweights=None, colors=None, antialiaseds=None, linestyles='solid', offsets=None, transOffset=None, norm=None, cmap=None, pickradius=5, zorder=2, facecolors='none', **kwargs)

Bases: matplotlib.collections.Collection

All parameters must be sequences or scalars; if scalars, they will be converted to sequences. The property of the ith line segment is:

prop[i % len(props)]

e.g., the properties cycle if the len of props is less than the number of segments.

segments a sequence of (line0, line1, line2), where:

linen = (x0, y0), (x1, y1), ... (xm, ym)

or the equivalent numpy array with two columns. Each line can be a different length.

colors must be a sequence of RGBA tuples (e.g., arbitrary color strings, etc, not allowed).

antialiaseds must be a sequence of ones or zeros

linestyles ['solid' | 'dashed' | 'dashdot' | 'dotted'] a string or dash tuple. The dash tuple is:

(offset, onoffseq),

where onoffseq is an even length tuple of on and off ink in points.

If lineweights, colors, or antialiaseds is None, they default to their rcParams setting, in sequence form.

If offsets and transOffset are not None, then offsets are transformed by transOffset and applied after the segments have been transformed to display coordinates.

If offsets is not None but transOffset is None, then the offsets are added to the segments before any transformation. In this case, a single offset can be specified as:

offsets=(xo,yo)

and this value will be added cumulatively to each successive segment, so as to produce a set of successively offset curves.

norm None (optional for matplotlib.cm.ScalarMappable)

cmap None (optional for matplotlib.cm.ScalarMappable)

pickradius is the tolerance for mouse clicks picking a line. The default is 5 pt.

zorder The zorder of the LineCollection. Default is 2

facecolors The facecolors of the LineCollection. Default is ‘none’ Setting to a value other than ‘none’ will lead to a filled polygon being drawn between points on each line.

The use of ScalarMappable is optional. If the ScalarMappable array _A is not None (i.e., a call to set_array() has been made), at draw time a call to scalar mappable will be made to set the colors.
add_callback(func)

Adds a callback function that will be called whenever one of the Artist's properties changes.

Returns an id that is useful for removing the callback with remove_callback() later.

add_checker(checker)

Add an entry to a dictionary of boolean flags that are set to True when the mappable is changed.

aname = ‘Artist’

autoscale()

Autoscale the scalar limits on the norm instance using the current array

autoscale_None()

Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

axes

The Axes instance the artist resides in, or None.

changed()

Call this whenever the mappable is changed to notify all the callbackSM listeners to the ‘changed’ signal

check_update(checker)

If mappable has changed since the last check, return True; else return False

contains(mouseevent)

Test whether the mouse event occurred in the collection.

Returns True | False, dict(ind=itemlist), where every item in itemlist contains the event.

convert_xunits(x)

For artists in an axes, if the xaxis has units support, convert x using xaxis unit type

convert_yunits(y)

For artists in an axes, if the yaxis has units support, convert y using yaxis unit type

draw(artist, renderer, *args, **kwargs)

findobj(match=match, include_self=True)

Find artist objects.

Recursively find all Artist instances contained in self.

match can be

- None: return all objects contained in artist.
- function with signature boolean = match(artist) used to filter matches
- class instance: e.g., Line2D. Only return artists of class type.

If include_self is True (default), include self in the list to be checked for a match.
**format_cursor_data** (*data*)
Return cursor data string formatted.

**get_agg_filter**()
return filter function to be used for agg filter

**get_alpha**()
Return the alpha value used for blending - not supported on all backends

**get_animated**()
Return the artist’s animated state

**get_array**()
Return the array

**get_axes**()
Return the *Axes* instance the artist resides in, or *None*.
This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

**get_children**()
Return a list of the child *Artist`'s this :class:`Artist` contains.

**get_clim**()
return the min, max of the color limits for image scaling

**get_clip_box**()
Return artist clipbox

**get_clip_on**()
Return whether artist uses clipping

**get_clip_path**()
Return artist clip path

**get_cmap**()
return the colormap

**get_color**()

**get_colors**()

**get_contains**()
Return the _contains test used by the artist, or *None* for default.

**get_cursor_data**(*event*)
Get the cursor data for a given event.

**get_dashes**()

**get_datalim**(*transData*)
get_edgecolor()

get_edgecolors()

get_facecolor()

get_facecolors()

get_figure()
    Return the *Figure* instance the artist belongs to.

get_fill()
    return whether fill is set

get_gid()
    Returns the group id

get_hatch()
    Return the current hatching pattern

get_label()
    Get the label used for this artist in the legend.

get_linestyle()

get_linestyles()

get_linewidth()

get_linewidths()

get_offset_position()
    Returns how offsets are applied for the collection. If *offset_position* is ‘screen’, the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If *offset_position* is ‘data’, the offset is applied before the master transform, i.e., the offsets are in data coordinates.

get_offset_transform()

get_offsets()
    Return the offsets for the collection.

get_path_effects()

get_paths()
get_picker()  
   Return the picker object used by this artist

get_pickradius()

get_rasterized()  
   return True if the artist is to be rasterized

get_segments()

get_sketch_params()  
   Returns the sketch parameters for the artist.

   Returns sketch_params : tuple or None
   A 3-tuple with the following elements:
   • scale: The amplitude of the wiggle perpendicular to the source line.
   • length: The length of the wiggle along the line.
   • randomness: The scale factor by which the length is shrunken or expanded.
   May return None if no sketch parameters were set.

get_snap()  
   Returns the snap setting which may be:
   • True: snap vertices to the nearest pixel center
   • False: leave vertices as-is
   • None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center
   Only supported by the Agg and MacOSX backends.

get_transform()  
   Return the Transform instance used by this artist.

get_transformed_clip_path_and_affine()  
   Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

get_transforms()

get_url()  
   Returns the url

get_urls()

get_visible()  
   Return the artist’s visibility
get_window_extent(renderer)

get_zorder()
Return the Artist's zorder.

have_units()
Return True if units are set on the x or y axes.

hitlist(event)
List the children of the artist which contain the mouse event event.

is_figure_set()
Returns True if the artist is assigned to a Figure.

is_transform_set()
Returns True if Artist has a transform explicitly set.

mouseover

pchanged()
Fire an event when property changed, calling all of the registered callbacks.

pick(mouseevent)
Process pick event

pickable()
Return True if Artist is pickable.

properties()
return a dictionary mapping property name -> value for all Artist props.

remove()
Remove the artist from the figure if possible. The effect will not be visible until the figure is redrawn, e.g., with matplotlib.axes.Axes.draw_idle(). Call matplotlib.axes.Axes.relim() to update the axes limits if desired.

Note: relim() will not see collections even if the collection was added to axes with autolim = True.

Note: there is no support for removing the artist’s legend entry.

remove_callback(oid)
Remove a callback based on its id.

See also:

add_callback() For adding callbacks

set(**kwargs)
A property batch setter. Pass kwargs to set properties.
**set_agg_filter**(*filter_func*)
- `set_agg_filter` function.

**set_alpha**(*alpha*)
- Set the alpha transparencies of the collection. *alpha* must be a float or *None*.

 ACCEPTS: float or *None*

**set_animated**(*b*)
- Set the artist’s animation state.

 ACCEPTS: [True | False]

**set_antialiased**(*aa*)
- Set the antialiasing state for rendering.

 ACCEPTS: Boolean or sequence of booleans

**set_antialiaseds**(*aa*)
- alias for `set_antialiased`

**set_array**(*A*)
- Set the image array from numpy array *A*

**set_axes**(*axes*)
- Set the `Axes` instance in which the artist resides, if any.

 This has been deprecated in mpl 1.5, please use the `axes` property. Will be removed in 1.7 or 2.0.

 ACCEPTS: an `Axes` instance

**set_clim**(*vmin=None, vmax=None*)
- set the norm limits for image scaling; if *vmin* is a length2 sequence, interpret it as *(vmin, vmax)* which is used to support `setp`

 ACCEPTS: a length 2 sequence of floats

**set_clip_box**(*clipbox*)
- Set the artist’s clip *Bbox*.

 ACCEPTS: a `matplotlib.transforms.Bbox` instance

**set_clip_on**(*b*)
- Set whether artist uses clipping.

 When False artists will be visible out side of the axes which can lead to unexpected results.

 ACCEPTS: [True | False]

**set_clip_path**(*path, transform=None*)
- Set the artist’s clip path, which may be:
  - a *Patch* (or subclass) instance
  - a *Path* instance, in which case an optional *Transform* instance may be provided, which will be applied to the path before using it for clipping.
  - *None*, to remove the clipping path
For efficiency, if the path happens to be an axis-aligned rectangle, this method will set the clipping box to the corresponding rectangle and set the clipping path to None.

ACCEPTS: [ (Path, Transform) | Patch | None ]

set_cmap(cmap)
set the colormap for luminance data

ACCEPTS: a colormap or registered colormap name

set_color(c)
Set the color(s) of the line collection. c can be a matplotlib color arg (all patches have same color), or a sequence or rgba tuples; if it is a sequence the patches will cycle through the sequence.

ACCEPTS: matplotlib color arg or sequence of rgba tuples

set_contains(picker)
Replace the contains test used by this artist. The new picker should be a callable function which determines whether the artist is hit by the mouse event:

```python
hit, props = picker(artist, mouseevent)
```

If the mouse event is over the artist, return hit = True and props is a dictionary of properties you want returned with the contains test.

ACCEPTS: a callable function

set_dashes(ls)
alias for set_linestyle

set_edgecolor(c)
Set the edgecolor(s) of the collection. c can be a matplotlib color spec (all patches have same color), or a sequence of specs; if it is a sequence the patches will cycle through the sequence.

If c is ‘face’, the edge color will always be the same as the face color. If it is ‘none’, the patch boundary will not be drawn.

ACCEPTS: matplotlib color spec or sequence of specs

set_edgecolors(c)
alias for set_edgecolor

set_facecolor(c)
Set the facecolor(s) of the collection. c can be a matplotlib color spec (all patches have same color), or a sequence of specs; if it is a sequence the patches will cycle through the sequence.

If c is ‘none’, the patch will not be filled.

ACCEPTS: matplotlib color spec or sequence of specs

set_facecolors(c)
alias for set_facecolor

set_figure(fig)
Set the Figure instance the artist belongs to.
`set_gid(gid)`
Sets the (group) id for the artist

`set_hatch(hatch)`
Set the hatching pattern

`hatch` can be one of:

```
/ - diagonal hatching
\ - back diagonal
| - vertical
- - horizontal
+ - crossed diagonal
x - crossed diagonal
o - small circle
O - large circle
. - dots
* - stars
```

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

`set_label(s)`
Set the label to `s` for auto legend.

`set_linestyle(ls)`
Set the linestyle(s) for the collection.

```
<table>
<thead>
<tr>
<th>linestyle</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'-' or 'solid'</td>
<td>solid line</td>
</tr>
<tr>
<td>'--' or 'dashed'</td>
<td>dashed line</td>
</tr>
<tr>
<td>'-.' or 'dashdot'</td>
<td>dash-dotted line</td>
</tr>
<tr>
<td>':' or 'dotted'</td>
<td>dotted line</td>
</tr>
</tbody>
</table>
```

Alternatively a dash tuple of the following form can be provided:

```
(offset, onoffseq),
```

where `onoffseq` is an even length tuple of on and off ink in points.

`ACCEPTS: ['solid', 'dashed', 'dashdot', 'dotted'] [(offset, on-off-dash-seq) | '-' | '--' | '-.' | ':' | 'None' | ' ' | '']`
Parameters `ls` : { ‘-‘, ‘–’ , ‘-.’ , ‘:’ } and more see description
The line style.

```
set_linestyles(ls)
alias for set_linestyle
```

```
set_linewidth(lw)
Set the linewidth(s) for the collection. `lw` can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence
ACCEPTS: float or sequence of floats
```

```
set_linewidths(lw)
alías for set_linewidth
```

```
set_lw(lw)
alías for set_linewidth
```

```
set_norm(norm)
set the normalization instance
```

```
set_offset_position(offset_position)
Set how offsets are applied. If `offset_position` is ‘screen’ (default) the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If `offset_position` is ‘data’, the offset is applied before the master transform, i.e., the offsets are in data coordinates.
```

```
set_offsets(offsets)
Set the offsets for the collection. `offsets` can be a scalar or a sequence.
ACCEPTS: float or sequence of floats
```

```
set_path_effects(path_effects)
set path_effects, which should be a list of instances of matplotlib.path._Base class or its derivatives.
```

```
set_paths(segments)
```

```
set_picker(picker)
Set the epsilon for picking used by this artist

`picker` can be one of the following:

• `None`: picking is disabled for this artist (default)

• A boolean: if `True` then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist

• A float: if picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire an event if it’s data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event
A function: if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

to determine the hit test. If the mouse event is over the artist, return \(hit=True\) and props is a dictionary of properties you want added to the PickEvent attributes.

**ACCEPTS**: [None|float|boolean|callable]

**set_pickradius**(pr)

**set_rasterized**(rasterized)

Force rasterized (bitmap) drawing in vector backend output.

Defaults to None, which implies the backend’s default behavior

**ACCEPTS**: [True | False | None]

**set_segments**(segments)

**set_sketch_params**(scale=None, length=None, randomness=None)

Sets the sketch parameters.

**Parameters**

- **scale** : float, optional
  
  The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is None, or not provided, no sketch filter will be provided.
  
- **length** : float, optional
  
  The length of the wiggle along the line, in pixels (default 128.0)
  
- **randomness** : float, optional
  
  The scale factor by which the length is shrunken or expanded (default 16.0)

**set_snap**(snap)

Sets the snap setting which may be:

- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

**set_transform**(t)

Set the Transform instance used by this artist.

**ACCEPTS**: Transform instance

**set_url**(url)

Sets the url for the artist
set_urls(urls)

set_verts(segments)

set_visible(b)
    Set the artist’s visibility.
    ACCEPTS: [True | False]

set_zorder(level)
    Set the zorder for the artist. Artists with lower zorder values are drawn first.
    ACCEPTS: any number

stale
    If the artist is ‘stale’ and needs to be re-drawn for the output to match the internal state of the artist.

sticky_edges
    x and y sticky edge lists.
    When performing autoscaling, if a data limit coincides with a value in the corresponding sticky_edges list, then no margin will be added—the view limit “sticks” to the edge. A typical use case is histograms, where one usually expects no margin on the bottom edge (0) of the histogram.
    This attribute cannot be assigned to; however, the x and y lists can be modified in place as needed.

Examples

>>> artist.sticky_edges.x[:] = (xmin, xmax)
>>> artist.sticky_edges.y[:] = (ymin, ymax)

to_rgba(x, alpha=0.0, bytes=False, norm=True)
    Return a normalized rgba array corresponding to x.
    In the normal case, x is a 1-D or 2-D sequence of scalars, and the corresponding ndarray of rgba values will be returned, based on the norm and colormap set for this ScalarMappable.
    There is one special case, for handling images that are already rgb or rgba, such as might have been read from an image file. If x is an ndarray with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an rgb or rgba array, and no mapping will be done. If the last dimension is 3, the alpha kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the alpha kwarg is ignored; it does not replace the pre-existing alpha. A ValueError will be raised if the third dimension is other than 3 or 4.
    In either case, if bytes is False (default), the rgba array will be floats in the 0-1 range; if it is True, the returned rgba array will be uint8 in the 0 to 255 range.
If norm is False, no normalization of the input data is performed, and it is assumed to already be in the range (0-1).

Note: this method assumes the input is well-behaved; it does not check for anomalies such as \( x \) being a masked rgba array, or being an integer type other than uint8, or being a floating point rgba array with values outside the 0-1 range.

**update** *(props)*

Update the properties of this *Artist* from the dictionary *prop*.

**update_from** *(other)*

Copy properties from other to self

**update_scalarmappable()**

If the scalar mappable array is not none, update colors from scalar data

\[ zorder = 0 \]

class **matplotlib.collections.PatchCollection** *(patches, match_original=False, **kwargs)*

Bases: **matplotlib.collections.Collection**

A generic collection of patches.

This makes it easier to assign a color map to a heterogeneous collection of patches.

This also may improve plotting speed, since PatchCollection will draw faster than a large number of patches.

*patches* a sequence of Patch objects. This list may include a heterogeneous assortment of different patch types.

*match_original* If True, use the colors and linewdiths of the original patches. If False, new colors may be assigned by providing the standard collection arguments, facecolor, edgecolor, linewidths, norm or cmap.

If any of *edgecolors*, *facecolors*, *linewidths*, *antialiaseds* are None, they default to their *matplotlib.pyplot.rcParams* patch setting, in sequence form.

The use of *ScalarMappable* is optional. If the *ScalarMappable* matrix _A_ is not None (i.e., a call to *set_array* has been made), at draw time a call to scalar mappable will be made to set the face colors.

**add_callback** *(func)*

Adds a callback function that will be called whenever one of the *Artist*’s properties changes.

Returns an *id* that is useful for removing the callback with *remove_callback()* later.

**add_checker** *(checker)*

Add an entry to a dictionary of boolean flags that are set to True when the mappable is changed.

aname = ‘Artist’

**autoscale()**

Autoscale the scalar limits on the norm instance using the current array
autoscale_None()
    Autoscale the scalar limits on the norm instance using the current array, changing only limits
    that are None.

axes
    The Axes instance the artist resides in, or None.

changed()
    Call this whenever the mappable is changed to notify all the callback listeners to the
    ‘changed’ signal.

check_update(checker)
    If mappable has changed since the last check, return True; else return False.

contains(mouseevent)
    Test whether the mouse event occurred in the collection.
    Returns True | False, dict(ind=itemlist), where every item in itemlist contains the event.

convert_xunits(x)
    For artists in an axes, if the xaxis has units support, convert x using xaxis unit type.

convert_yunits(y)
    For artists in an axes, if the yaxis has units support, convert y using yaxis unit type.

draw(artist, renderer, *args, **kwargs)

findobj(match=None, include_self=True)
    Find artist objects.
    Recursively find all Artist instances contained in self.

        match can be

        • None: return all objects contained in artist.
        • function with signature boolean = match(artist) used to filter matches
        • class instance: e.g., Line2D. Only return artists of class type.

    If include_self is True (default), include self in the list to be checked for a match.

format_cursor_data(data)
    Return cursor data string formatted.

get_agg_filter()
    return filter function to be used for agg filter.

get_alpha()
    Return the alpha value used for blending - not supported on all backends.

get_animated()
    Return the artist’s animated state.

get_array()
    Return the array.
get_axes()
Return the Axes instance the artist resides in, or None.

This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

get_axes()
Return a list of the child Artist’s this :class:`Artist` contains.

get_clim()
return the min, max of the color limits for image scaling

get_clip_box()
Return artist clipbox

get_clip_on()
Return whether artist uses clipping

get_clip_path()
Return artist clip path

get_cmap()
return the colormap

get_contains()
Return the _contains test used by the artist, or None for default.

get_cursor_data(event)
Get the cursor data for a given event.

get_dashes()

get_datalim(transData)

get_edgecolor()

get_edgecolors()

get_facecolor()

get_facecolors()

get_figure()
Return the Figure instance the artist belongs to.

get_fill()
return whether fill is set

get_gid()
Returns the group id
get_hatch()
Return the current hatching pattern

get_label()
Get the label used for this artist in the legend.

get_linestyle()

get_linestyles()

get_linewidth()

get_linewidths()

get_offset_position()
Returns how offsets are applied for the collection. If offset_position is ‘screen’, the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is ‘data’, the offset is applied before the master transform, i.e., the offsets are in data coordinates.

get_offset_transform()

get_offsets()
Return the offsets for the collection.

get_path_effects()

get_paths()

get_picker()
Return the picker object used by this artist

get_pickradius()

get_rasterized()
return True if the artist is to be rasterized

get_sketch_params()
Returns the sketch parameters for the artist.

Returns sketch_params : tuple or None
A 3-tuple with the following elements:
• scale: The amplitude of the wiggle perpendicular to the source line.
• length: The length of the wiggle along the line.
• randomness: The scale factor by which the length is shrunken or expanded.
May return None if no sketch parameters were set.

get_snap()
Returns the snap setting which may be:

• True: snap vertices to the nearest pixel center
• False: leave vertices as-is
• None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

get_transform()
Return the Transform instance used by this artist.

get_transformed_clip_path_and_affine()
Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

get_transforms()

get_url()
Returns the url

get_urls()

get_visible()
Return the artist’s visibility

get_window_extent(renderer)

get_zorder()
Return the Artist’s zorder.

have_units()
Return True if units are set on the x or y axes

hitlist(event)
List the children of the artist which contain the mouse event event.

is_figure_set()
Returns True if the artist is assigned to a Figure.

is_transform_set()
Returns True if Artist has a transform explicitly set.

mouseover

pchanged()
Fire an event when property changed, calling all of the registered callbacks.
**pick**(mouseevent)
Process pick event

each child artist will fire a pick event if mouseevent is over the artist and the artist has picker set

**pickable**
Return True if Artist is pickable.

**properties**
return a dictionary mapping property name -> value for all Artist props

**remove**
Remove the artist from the figure if possible. The effect will not be visible until the figure is redrawn, e.g., with matplotlib.axes.Axes.draw_idle(). Call matplotlib.axes.Axes.relim() to update the axes limits if desired.

Note: relim() will not see collections even if the collection was added to axes with autolim = True.

Note: there is no support for removing the artist’s legend entry.

**remove_callback**(oid)
Remove a callback based on its id.

See also:

**add_callback**() For adding callbacks

**set**(**kwargs**)
A property batch setter. Pass kwargs to set properties.

**set_agg_filter**(filter_func)
set agg_filter fuction.

**set_alpha**(alpha)
Set the alpha transparencies of the collection. alpha must be a float or None.

ACCEPTS: float or None

**set_animated**(b)
Set the artist’s animation state.

ACCEPTS: [True | False]

**set_antialiased**(aa)
Set the antialiasing state for rendering.

ACCEPTS: Boolean or sequence of booleans

**set_antialiaseds**(aa)
alias for set_antialiased

**set_array**(A)
Set the image array from numpy array A
**set_axes(ax)**

Set the **Axes** instance in which the artist resides, if any.

This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

ACCEPTS: an **Axes** instance

**set_clim(vmin=None, vmax=None)**

set the norm limits for image scaling; if vmin is a length 2 sequence, interpret it as (vmin, vmax) which is used to support setp

ACCEPTS: a length 2 sequence of floats

**set_clip_box(clipbox)**

Set the artist’s clip **bbox**.

ACCEPTS: a **matplotlib.transforms.Bbox** instance

**set_clip_on(b)**

Set whether artist uses clipping.

When False artists will be visible out side of the axes which can lead to unexpected results.

ACCEPTS: [True | False]

**set_clip_path(path, transform=None)**

Set the artist’s clip path, which may be:

- a **Patch** (or subclass) instance
- a **Path** instance, in which case an optional **Transform** instance may be provided, which will be applied to the path before using it for clipping.
- **None**, to remove the clipping path

For efficiency, if the path happens to be an axis-aligned rectangle, this method will set the clipping box to the corresponding rectangle and set the clipping path to **None**.

ACCEPTS: [ (Path, Transform) | Patch | None ]

**set_cmap(cmap)**

set the colormap for luminance data

ACCEPTS: a colormap or registered colormap name

**set_color(c)**

Set both the edgecolor and the facecolor.

ACCEPTS: matplotlib color arg or sequence of rgba tuples

See also:

- **set_facecolor()**, **set_edgecolor()** For setting the edge or face color individually.

**set_contains(picker)**

Replace the contains test used by this artist. The new picker should be a callable function which determines whether the artist is hit by the mouse event:
hit, props = picker(artist, mouseevent)

If the mouse event is over the artist, return hit = True and props is a dictionary of properties you want returned with the contains test.

ACCEPTS: a callable function

**set_dashes**(ls)
alias for set_linestyle

**set_edgecolor**(c)
Set the edgecolor(s) of the collection. c can be a matplotlib color spec (all patches have same color), or a sequence of specs; if it is a sequence the patches will cycle through the sequence.

If c is ‘face’, the edge color will always be the same as the face color. If it is ‘none’, the patch boundary will not be drawn.

ACCEPTS: matplotlib color spec or sequence of specs

**set_edgecolors**(c)
alias for set_edgecolor

**set_facecolor**(c)
Set the facecolor(s) of the collection. c can be a matplotlib color spec (all patches have same color), or a sequence of specs; if it is a sequence the patches will cycle through the sequence.

If c is ‘none’, the patch will not be filled.

ACCEPTS: matplotlib color spec or sequence of specs

**set_facecolors**(c)
alias for set_facecolor

**set_figure**(fig)
Set the Figure instance the artist belongs to.

ACCEPTS: a matplotlib.figure.Figure instance

**set_gid**(gid)
Sets the (group) id for the artist

ACCEPTS: an id string

**set_hatch**(hatch)
Set the hatching pattern

hatch can be one of:

/ - diagonal hatching
\ - back diagonal
| - vertical
- - horizontal
+ - crossed
x - crossed diagonal
o - small circle
O - large circle
Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

**ACCEPTS:** [ ‘/’ | ‘\’ | ‘|’ | ‘-.’ | ‘+’ | ‘x’ | ‘o’ | ‘O’ | ‘.’ | ‘*’ ]

`set_label(s)`
Set the label to `s` for auto legend.

**ACCEPTS:** string or anything printable with ‘%s’ conversion.

`set_linestyle(ls)`
Set the linestyle(s) for the collection.

<table>
<thead>
<tr>
<th>linestyle</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'-' or 'solid'</td>
<td>solid line</td>
</tr>
<tr>
<td>'--' or 'dashed'</td>
<td>dashed line</td>
</tr>
<tr>
<td>'-.' or 'dashdot'</td>
<td>dash-dotted line</td>
</tr>
<tr>
<td>':' or 'dotted'</td>
<td>dotted line</td>
</tr>
</tbody>
</table>

Alternatively a dash tuple of the following form can be provided:

```python
[offset, onoffseq],
```

where `onoffseq` is an even length tuple of on and off ink in points.

**ACCEPTS:** [‘solid’ | ‘dashed’, ‘dashdot’, ‘dotted’ | (offset, on-off-dash-seq) | ‘-’ | ‘--’ | ‘-.’ | ‘:’ | ‘None’ | ‘ ’ | ‘’ | ‘’]

**Parameters ls :** { ‘-‘, ‘-‘, ‘-‘’, ‘:’} and more see description

The line style.

`set_linestyles(ls)`
alias for `set_linestyle`

`set_linewidth(lw)`
Set the linewidth(s) for the collection. `lw` can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence

**ACCEPTS:** float or sequence of floats

`set_linewidths(lw)`
alias for `set_linewidth`

`set_lw(lw)`
alias for `set_linewidth`
**set_norm** *(norm)*
set the normalization instance

**set_offset_position** *(offset_position)*
Set how offsets are applied. If *offset_position* is ‘screen’ (default) the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If *offset_position* is ‘data’, the offset is applied before the master transform, i.e., the offsets are in data coordinates.

**set_offsets** *(offsets)*
Set the offsets for the collection. *offsets* can be a scalar or a sequence.

ACCEPTS: float or sequence of floats

**set_path_effects** *(path_effects)*
set path_effects, which should be a list of instances of matplotlib.path._Base class or its derivatives.

**set_paths** *(patches)*

**set_picker** *(picker)*
Set the epsilon for picking used by this artist

picker can be one of the following:

- *None*: picking is disabled for this artist (default)
- A boolean: if *True* then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist
- A float: if picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if it’s data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event
- A function: if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

  ```python
  hit, props = picker(artist, mouseevent)
  ```

  to determine the hit test. if the mouse event is over the artist, return *hit=True* and *props* is a dictionary of properties you want added to the PickEvent attributes.

ACCEPTS: [None|float|boolean|callable]

**set_pickradius** *(pr)*

**set_rasterized** *(rasterized)*
Force rasterized (bitmap) drawing in vector backend output.

Defaults to None, which implies the backend’s default behavior

ACCEPTS: [True | False | None]
**set_sketch_params**(*scale=None, length=None, randomness=None*)

Sets the sketch parameters.

**Parameters scale** : float, optional

The amplitude of the wiggle perpendicular to the source line, in pixels. If `scale` is `None`, or not provided, no sketch filter will be provided.

**length** : float, optional

The length of the wiggle along the line, in pixels (default 128.0)

**randomness** : float, optional

The scale factor by which the length is shrunken or expanded (default 16.0)

**set_snap**(*snap*)

Sets the snap setting which may be:

- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

**set_transform**(*t*)

Set the `Transform` instance used by this artist.

ACCEPTS: `Transform` instance

**set_url**(*url*)

Sets the url for the artist

ACCEPTS: a url string

**set_urls**(*urls*)

**set_visible**(*b*)

Set the artist’s visibility.

ACCEPTS: [True | False]

**set_zorder**(*level*)

Set the zorder for the artist. Artists with lower zorder values are drawn first.

ACCEPTS: any number

**stale**

If the artist is ‘stale’ and needs to be re-drawn for the output to match the internal state of the artist.

**sticky_edges**

x and y sticky edge lists.
When performing autoscaling, if a data limit coincides with a value in the corresponding sticky_edges list, then no margin will be added—the view limit “sticks” to the edge. A typical use case is histograms, where one usually expects no margin on the bottom edge (0) of the histogram.

This attribute cannot be assigned to; however, the x and y lists can be modified in place as needed.

Examples

```python
>>> artist.sticky_edges.x[:] = (xmin, xmax)
>>> artist.sticky_edges.y[:] = (ymin, ymax)
```

to_rgba($x$, $alpha$=None, $bytes$=False, norm=True)

Return a normalized rgba array corresponding to $x$.

In the normal case, $x$ is a 1-D or 2-D sequence of scalars, and the corresponding ndarray of rgba values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already rgb or rgba, such as might have been read from an image file. If $x$ is an ndarray with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an rgb or rgba array, and no mapping will be done. If the last dimension is 3, the $alpha$ kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the $alpha$ kwarg is ignored; it does not replace the pre-existing alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if $bytes$ is False (default), the rgba array will be floats in the 0-1 range; if it is True, the returned rgba array will be uint8 in the 0 to 255 range.

If norm is False, no normalization of the input data is performed, and it is assumed to already be in the range (0-1).

Note: this method assumes the input is well-behaved; it does not check for anomalies such as $x$ being a masked rgba array, or being an integer type other than uint8, or being a floating point rgba array with values outside the 0-1 range.

update($props$)

Update the properties of this Artist from the dictionary $prop$.

update_from($other$)

copy properties from other to self

update_scalarmappable()

If the scalar mappable array is not none, update colors from scalar data

zorder = 0

class matplotlib.collections.PathCollection($paths$, $sizes$=None, **$kwargs$)

Bases: matplotlib.collections._CollectionWithSizes

This is the most basic Collection subclass.
paths is a sequence of matplotlib.path.Path instances.

Valid Collection keyword arguments:

- **edgecolors**: None
- **facecolors**: None
- **linewidths**: None
- **antialiases**: None
- **offsets**: None
- **transOffset**: transforms.IdentityTransform()
- **norm**: None (optional for matplotlib.cm.ScalarMappable)
- **cmap**: None (optional for matplotlib.cm.ScalarMappable)

offsets and transOffset are used to translate the patch after rendering (default no offsets)

If any of edgecolors, facecolors, linewidths, antialiases are None, they default to their matplotlib.rcParams patch setting, in sequence form.

**add_callback(func)**

Adds a callback function that will be called whenever one of the Artist's properties changes.

Returns an id that is useful for removing the callback with remove_callback() later.

**add_checker(checker)**

Add an entry to a dictionary of boolean flags that are set to True when the mappable is changed.

**aname** = ‘Artist’

**autoscale()**

Autoscale the scalar limits on the norm instance using the current array

**autoscale_None()**

Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

**axes**

The Axes instance the artist resides in, or None.

**changed()**

Call this whenever the mappable is changed to notify all the callbackSM listeners to the ‘changed’ signal

**check_update(checker)**

If mappable has changed since the last check, return True; else return False

**contains(mouseevent)**

Test whether the mouse event occurred in the collection.

Returns True | False, dict(ind=itemlist), where every item in itemlist contains the event.
convert_xunits(x)
    For artists in an axes, if the xaxis has units support, convert x using
    xaxis unit type

calculate_yunits(y)
    For artists in an axes, if the yaxis has units support, convert y using
    yaxis unit type

draw(artist, renderer, *args, **kwargs)

findobj(match=None, include_self=True)
    Find artist objects.
    Recursively find all Artist instances contained in self.
    match can be
    • None: return all objects contained in artist.
    • function with signature boolean = match(artist) used to filter matches
    • class instance: e.g., Line2D. Only return artists of class type.
    If include_self is True (default), include self in the list to be checked
    for a match.

format_cursor_data(data)
    Return cursor data string formatted.

get_agg_filter()
    return filter function to be used for agg filter

get_alpha()
    Return the alpha value used for blending - not supported on all backends

get_animated()
    Return the artist’s animated state

get_array()
    Return the array

get_axes()
    Return the Axes instance the artist resides in, or None.
    This has been deprecated in mpl 1.5, please use the axes property. Will be
    removed in 1.7 or 2.0.

get_children()
    Return a list of the child Artist’s this :class:`Artist`

get_clim()
    return the min, max of the color limits for image scaling

get_clip_box()
    Return artist clipbox

get_clip_on()
    Return whether artist uses clipping

get_clip_path()
    Return artist clip path
get_cmap()  
return the colormap

get_contains()  
Return the _contains test used by the artist, or None for default.

get_cursor_data(event)  
Get the cursor data for a given event.

get_dashes()  

get_datalim(transData)  

get_edgecolor()  

get_edgecolors()  

get_facecolor()  

get_facecolors()  

get_figure()  
Return the Figure instance the artist belongs to.

get_fill()  
return whether fill is set

get_gid()  
Returns the group id

get_hatch()  
Return the current hatching pattern

get_label()  
Get the label used for this artist in the legend.

get_linestyle()  

get_linestyles()  

get_linewidth()  

get_linewidths()  

get_offset_position()  
Returns how offsets are applied for the collection. If offset_position is ‘screen’, the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates.
If offset_position is ‘data’, the offset is applied before the master transform, i.e., the offsets are in data coordinates.

\texttt{get\_offset\_transform()}

\texttt{get\_offsets()}

Return the offsets for the collection.

\texttt{get\_path\_effects()}

\texttt{get\_paths()}

\texttt{get\_picker()}

Return the picker object used by this artist

\texttt{get\_pickradius()}

\texttt{get\_rasterized()}

Return True if the artist is to be rasterized

\texttt{get\_sizes()}

Returns the sizes of the elements in the collection. The value represents the ‘area’ of the element.  
\textbf{Returns} \texttt{sizes} : array

The ‘area’ of each element.

\texttt{get\_sketch\_params()}

Returns the sketch parameters for the artist.

\textbf{Returns} \texttt{sketch\_params} : tuple or \texttt{None}

A 3-tuple with the following elements:

\begin{itemize}
  \item \texttt{scale}: The amplitude of the wiggle perpendicular to the source line.
  \item \texttt{length}: The length of the wiggle along the line.
  \item \texttt{randomness}: The scale factor by which the length is shrunken or expanded.
\end{itemize}

May return \texttt{None} if no sketch parameters were set.

\texttt{get\_snap()}

Returns the snap setting which may be:

\begin{itemize}
  \item True: snap vertices to the nearest pixel center
  \item False: leave vertices as-is
  \item None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center
\end{itemize}

Only supported by the Agg and MacOSX backends.
get_transform()  
Return the Transform instance used by this artist.

get_transformed_clip_path_and_affine()  
Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

get_transforms()  

get_url()  
Returns the url

get_urls()  

get_visible()  
Return the artist’s visibility

get_window_extent(renderer)  

get_zorder()  
Return the Artist’s zorder.

have_units()  
Return True if units are set on the x or y axes

hitlist(event)  
List the children of the artist which contain the mouse event event.

is_figure_set()  
Returns True if the artist is assigned to a Figure.

is_transform_set()  
Returns True if Artist has a transform explicitly set.

mouseover

pchanged()  
Fire an event when property changed, calling all of the registered callbacks.

pick(mouseevent)  
Process pick event

pickable()  
Return True if Artist is pickable.

properties()  
return a dictionary mapping property name -> value for all Artist props

remove()  
Remove the artist from the figure if possible. The effect will not be visible until the figure is
redrawn, e.g., with `matplotlib.axes.Axes.draw_idle()`. Call `matplotlib.axes.Axes.relim()` to update the axes limits if desired.

Note: `relim()` will not see collections even if the collection was added to axes with `autolim = True`.

Note: there is no support for removing the artist’s legend entry.

```python
remove_callback(oid)
Remove a callback based on its id.
```

See also:

```python
add_callback() For adding callbacks
```

```python
set(**kwargs)
A property batch setter. Pass `kwargs` to set properties.
```

```python
set_agg_filter(filter_func)
s set agg_filter function.
```

```python
set_alpha(alpha)
Set the alpha transparencies of the collection. `alpha` must be a float or `None`.
```

`ACCEPTS`: float or `None`

```python
set_animated(b)
Set the artist’s animation state.
```

`ACCEPTS`: `[True | False]`

```python
set_antialiased(aa)
Set the antialiasing state for rendering.
```

`ACCEPTS`: Boolean or sequence of booleans

```python
set_antialiaseds(aa)
alias for `set_antialiased`
```

```python
set_array(A)
Set the image array from numpy array `A`
```

```python
set_axes(axes)
Set the `Axes` instance in which the artist resides, if any.
```

This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

`ACCEPTS`: an `Axes` instance

```python
set_clim(vmin=None, vmax=None)
set the norm limits for image scaling; if `vmin` is a length2 sequence, interpret it as `(vmin, vmax)` which is used to support `setp`
```

`ACCEPTS`: a length 2 sequence of floats

```python
set_clip_box(clipbox)
Set the artist’s clip `Bbox`
```
set_clip_on(b)
Set whether artist uses clipping.
When False artists will be visible out side of the axes which can lead to unexpected results.
ACCEPTS: [True | False]

set_clip_path(path, transform=None)
Set the artist’s clip path, which may be:
• a Patch (or subclass) instance
• a Path instance, in which case an optional Transform instance may be provided, which will be applied to the path before using it for clipping.
• None, to remove the clipping path
For efficiency, if the path happens to be an axis-aligned rectangle, this method will set the clipping box to the corresponding rectangle and set the clipping path to None.
ACCEPTS: [ (Path, Transform) | Patch | None ]

set_cmap(cmap)
set the colormap for luminance data
ACCEPTS: a colormap or registered colormap name

set_color(c)
Set both the edgecolor and the facecolor.
ACCEPTS: matplotlib color arg or sequence of rgba tuples
See also:
set_facecolor(), set_edgecolor() For setting the edge or face color individually.

set_contains(picker)
Replace the contains test used by this artist. The new picker should be a callable function which determines whether the artist is hit by the mouse event:

```python
hit, props = picker(artist, mouseevent)
```
If the mouse event is over the artist, return hit = True and props is a dictionary of properties you want returned with the contains test.
ACCEPTS: a callable function

set_dashes(ls)
alias for set_linestyle

set_edgecolor(c)
Set the edgecolor(s) of the collection. c can be a matplotlib color spec (all patches have same color), or a sequence of specs; if it is a sequence the patches will cycle through the sequence.
If `c` is ‘face’, the edge color will always be the same as the face color. If it is ‘none’, the patch boundary will not be drawn.

**ACCEPTS:** matplotlib color spec or sequence of specs

```python
set_edgecolors(c)
```

alias for `set_edgecolor`

```python
set_facecolor(c)
```

Set the facecolor(s) of the collection. `c` can be a matplotlib color spec (all patches have same color), or a sequence of specs; if it is a sequence the patches will cycle through the sequence.

If `c` is ‘none’, the patch will not be filled.

**ACCEPTS:** matplotlib color spec or sequence of specs

```python
set_facecolors(c)
```

alias for `set_facecolor`

```python
set_figure(fig)
```

Set the `Figure` instance the artist belongs to.

**ACCEPTS:** a `matplotlib.figure.Figure` instance

```python
set_gid(gid)
```

Sets the (group) id for the artist

**ACCEPTS:** an id string

```python
set_hatch(hatch)
```

Set the hatching pattern

`hatch` can be one of:

```
/ - diagonal hatching
\ - back diagonal
| - vertical
- - horizontal
+ - crossed
x - crossed diagonal
o - small circle
O - large circle
. - dots
* - stars
```

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

**ACCEPTS:** `[ / \ | `\` | `\` | `\` | `\` | `\` | `\` | `\` | `\` | `\` ]`

```python
set_label(s)
```

Set the label to `s` for auto legend.
ACCEPTS: string or anything printable with ‘%s’ conversion.

**set_linestyle**(ls)
Set the linestyle(s) for the collection.

<table>
<thead>
<tr>
<th>linestyle</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'-' or 'solid'</td>
<td>solid line</td>
</tr>
<tr>
<td>'-' or 'dashed'</td>
<td>dashed line</td>
</tr>
<tr>
<td>'-' or 'dashdot'</td>
<td>dash-dotted line</td>
</tr>
<tr>
<td>':' or 'dotted'</td>
<td>dotted line</td>
</tr>
</tbody>
</table>

Alternatively a dash tuple of the following form can be provided:

```
(offset, onoffseq),
```

where onoffseq is an even length tuple of on and off ink in points.

**ACCEPTS:** ['solid', 'dashed', 'dashdot', 'dotted', (offset, on-off-dash-seq) | '-' | '--' | '-.' | ':' | 'None' | ' ' | ' ']

**Parameters ls:** { '-', '-', '-', ':' } and more see description

The line style.

**set_linestyles**(ls)
alias for set_linestyle

**set_linewidth**(lw)
Set the linewidth(s) for the collection. *lw* can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence

**ACCEPTS:** float or sequence of floats

**set_linewidths**(lw)
alias for set_linewidth

**set_lw**(lw)
alias for set_linewidth

**set_norm**(norm)
set the normalization instance

**set_offset_position**(offset_position)
Set how offsets are applied. If offset_position is ‘screen’ (default) the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is ‘data’, the offset is applied before the master transform, i.e., the offsets are in data coordinates.

**set_offsets**(offsets)
Set the offsets for the collection. *offsets* can be a scalar or a sequence.

**ACCEPTS:** float or sequence of floats
**set_path_effects**(path_effects)

Set path_effects, which should be a list of instances of matplotlib.pathEffects._Base class or its derivatives.

**set_paths**(paths)

**set_picker**(picker)

Set the epsilon for picking used by this artist

*picker* can be one of the following:

- *None*: picking is disabled for this artist (default)
- A boolean: if *True* then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist
- A float: if picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if it’s data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event
- A function: if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```python
hit, props = picker(artist, mouseevent)
```

to determine the hit test. If the mouse event is over the artist, return *hit=True* and props is a dictionary of properties you want added to the PickEvent attributes.

ACCEPTS: [None|float|boolean|callable]

**set_pickradius**(pr)

**set_rasterized**(rasterized)

Force rasterized (bitmap) drawing in vector backend output.

Defaults to None, which implies the backend’s default behavior

ACCEPTS: [True | False | None]

**set_sizes**(sizes, dpi=72.0)

Set the sizes of each member of the collection.

**Parameters**

- sizes : ndarray or None

  The size to set for each element of the collection. The value is the ‘area’ of the element.

- dpi : float

  The dpi of the canvas. Defaults to 72.0.

**set_sketch_params**(scale=None, length=None, randomness=None)

Sets the sketch parameters.
Parameters

scale : float, optional

The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is `None`, or not provided, no sketch filter will be provided.

length : float, optional

The length of the wiggle along the line, in pixels (default 128.0)

randomness : float, optional

The scale factor by which the length is shrunken or expanded (default 16.0)

set_snap(snap)

Sets the snap setting which may be:

- **True**: snap vertices to the nearest pixel center
- **False**: leave vertices as-is
- **None**: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

set_transform(t)

Set the `Transform` instance used by this artist.

Accepts: `Transform` instance

set_url(url)

Sets the url for the artist

Accepts: a url string

set_urls(urls)

set_visible(b)

Set the artist’s visibility.

Accepts: [True | False]

set_zorder(level)

Set the zorder for the artist. Artists with lower zorder values are drawn first.

Accepts: any number

stale

If the artist is ‘stale’ and needs to be re-drawn for the output to match the internal state of the artist.

sticky_edges

x and y sticky edge lists.

When performing autoscaling, if a data limit coincides with a value in the corresponding sticky_edges list, then no margin will be added—the view limit “sticks” to the edge. A typical usecase is histograms, where one usually expects no margin on the bottom edge (0) of the histogram.
This attribute cannot be assigned to; however, the x and y lists can be modified in place as needed.

**Examples**

```python
>>> artist.sticky_edges.x[:] = (xmin, xmax)
>>> artist.sticky_edges.y[:] = (ymin, ymax)
```

to_rgba($x$, $alpha$=None, $bytes$=False, $norm$=True)

Return a normalized rgba array corresponding to $x$.

In the normal case, $x$ is a 1-D or 2-D sequence of scalars, and the corresponding ndarray of rgba values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already rgb or rgba, such as might have been read from an image file. If $x$ is an ndarray with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an rgb or rgba array, and no mapping will be done. If the last dimension is 3, the $alpha$ kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the $alpha$ kwarg is ignored; it does not replace the pre-existing alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if $bytes$ is False (default), the rgba array will be floats in the 0-1 range; if it is True, the returned rgba array will be uint8 in the 0 to 255 range.

If norm is False, no normalization of the input data is performed, and it is assumed to already be in the range (0-1).

Note: this method assumes the input is well-behaved; it does not check for anomalies such as $x$ being a masked rgba array, or being an integer type other than uint8, or being a floating point rgba array with values outside the 0-1 range.

update($props$)

Update the properties of this Artist from the dictionary $prop$.

update_from($other$)

copy properties from other to self

update_scalarmappable()

If the scalar mappable array is not none, update colors from scalar data

zorder = 0

class matplotlib.collections.PolyCollection($verts$, $sizes$=None, $closed$=True, **$kwargs$)

Bases: matplotlib.collections._CollectionWithSizes

$verts$ is a sequence of ($verts0$, $verts1$, ...) where $verts_i$ is a sequence of $xy$ tuples of vertices, or an equivalent numpy array of shape ($nv$, 2).

$sizes$ is None (default) or a sequence of floats that scale the corresponding $verts_i$. The scaling is applied before the Artist master transform; if the latter is an identity transform, then the overall scaling is such that if $verts_i$ specify a unit square, then $sizes_i$ is the area of that square in points$^2$. If len($sizes$) < $nv$, the additional values will be taken cyclically from the array.
closed, when True, will explicitly close the polygon.

Valid Collection keyword arguments:

- **edgecolors**: None
- **facecolors**: None
- **linewidths**: None
- **antialiaseds**: None
- **offsets**: None
- **transOffset**: transforms.IdentityTransform()
- **norm**: None (optional for matplotlib.cm.ScalarMappable)
- **cmap**: None (optional for matplotlib.cm.ScalarMappable)

offsets and transOffset are used to translate the patch after rendering (default no offsets)

If any of edgecolors, facecolors, linewidths, antialiaseds are None, they default to their matplotlib.rcParams patch setting, in sequence form.

**add_callback(func)**

Adds a callback function that will be called whenever one of the Artist’s properties changes.

Returns an id that is useful for removing the callback with remove_callback() later.

**add_checker(checker)**

Add an entry to a dictionary of boolean flags that are set to True when the mappable is changed.

**aname = ‘Artist’**

**autoscale()**

Autoscale the scalar limits on the norm instance using the current array

**autoscale_None()**

Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

**axes**

The Axes instance the artist resides in, or None.

**changed()**

Call this whenever the mappable is changed to notify all the callbackSM listeners to the ‘changed’ signal

**check_update(checker)**

If mappable has changed since the last check, return True; else return False

**contains(mouseevent)**

Test whether the mouse event occurred in the collection.

Returns True | False, dict(ind=itemlist), where every item in itemlist contains the event.
convert_xunits(x)
    For artists in an axes, if the xaxis has units support, convert x using xaxis unit type

convert_yunits(y)
    For artists in an axes, if the yaxis has units support, convert y using yaxis unit type

draw(artist, renderer, *args, **kwargs)

findobj(match=None, include_self=True)
    Find artist objects.
    Recursively find all Artist instances contained in self.
    match can be
    • None: return all objects contained in artist.
    • function with signature boolean = match(artist) used to filter matches
    • class instance: e.g., Line2D. Only return artists of class type.
    If include_self is True (default), include self in the list to be checked for a match.

format_cursor_data(data)
    Return cursor data string formatted.

get_agg_filter()
    return filter function to be used for agg filter

get_alpha()
    Return the alpha value used for blending - not supported on all backends

get_animated()
    Return the artist’s animated state

get_array()
    Return the array

get_axes()
    Return the Axes instance the artist resides in, or None.
    This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

get_children()
    Return a list of the child Artist’s this :class:`Artist contains.

get_clim()
    return the min, max of the color limits for image scaling

get_clip_box()
    Return artist clipbox

get_clip_on()
    Return whether artist uses clipping

get_clip_path()
    Return artist clip path
get_cmap()

return the colormap

get_contains()

Return the _contains test used by the artist, or None for default.

get_cursor_data(event)

Get the cursor data for a given event.

get_dashes()

get_datalim(transData)

get_edgecolor()

get_edgecolors()

get_facecolor()

get_facecolors()

get_figure()

Return the Figure instance the artist belongs to.

get_fill()

return whether fill is set

get_gid()

Returns the group id

get_hatch()

Return the current hatching pattern

get_label()

Get the label used for this artist in the legend.

get_linestyle()

get_linestyles()

get_linewidth()

get_linewidths()

get_offset_position()

Returns how offsets are applied for the collection. If offset_position is ‘screen’, the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates.
If offset_position is ‘data’, the offset is applied before the master transform, i.e., the offsets are in data coordinates.

**get_offset_transform()**

**get_offsets()**

Return the offsets for the collection.

**get_path_effects()**

**get_paths()**

**get_picker()**

Return the picker object used by this artist

**get_pickradius()**

**get_rasterized()**

return True if the artist is to be rasterized

**get_sizes()**

Returns the sizes of the elements in the collection. The value represents the ‘area’ of the element.

**Returns** sizes : array

The ‘area’ of each element.

**get_sketch_params()**

Returns the sketch parameters for the artist.

**Returns** sketch_params : tuple or None

A 3-tuple with the following elements:
- **scale**: The amplitude of the wiggle perpendicular to the source line.
- **length**: The length of the wiggle along the line.
- **randomness**: The scale factor by which the length is shrunken or expanded.

May return None if no sketch parameters were set.

**get_snap()**

Returns the snap setting which may be:
- **True**: snap vertices to the nearest pixel center
- **False**: leave vertices as-is
- **None**: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.
get_transform()  
Return the Transform instance used by this artist.

get_transformed_clip_path_and_affine()  
Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

get_transforms()  

get_url()  
Returns the url

get_urls()  

get_visible()  
Return the artist’s visibility

get_window_extent(renderer)  

get_zorder()  
Return the Artist’s zorder.

have_units()  
Return True if units are set on the x or y axes

hitlist(event)  
List the children of the artist which contain the mouse event event.

is_figure_set()  
Returns True if the artist is assigned to a Figure.

is_transform_set()  
Returns True if Artist has a transform explicitly set.

mouseover  

pchanged()  
Fire an event when property changed, calling all of the registered callbacks.

pick(mouseevent)  
Process pick event

    each child artist will fire a pick event if mouseevent is over the artist and the artist has picker set

pickable()  
Return True if Artist is pickable.

properties()  
return a dictionary mapping property name -> value for all Artist props

remove()  
Remove the artist from the figure if possible. The effect will not be visible until the figure is
redrawn, e.g., with `matplotlib.axes.Axes.draw_idle()`. Call `matplotlib.axes.Axes.relim()` to update the axes limits if desired.

Note: `relim()` will not see collections even if the collection was added to axes with `autolim = True`.

Note: there is no support for removing the artist’s legend entry.

`remove_callback(oid)`  
Remove a callback based on its id.

See also:

`add_callback()` For adding callbacks

`s**kwargs`  
A property batch setter. Pass `kwargs` to set properties.

`set_agg_filter(filter_func)`  
set agg_filter function.

`set_alpha(alpha)`  
Set the alpha tranparencies of the collection. `alpha` must be a float or None.

ACCEPTS: float or None

`set_animated(b)`  
Set the artist’s animation state.

ACCEPTS: [True | False]

`set_antialiased(aa)`  
Set the antialiasing state for rendering.

ACCEPTS: Boolean or sequence of bools

`set_antialiaseds(aa)`  
alias for set_antialiased

`set_array(A)`  
Set the image array from numpy array A

`set_axes(axes)`  
Set the `Axes` instance in which the artist resides, if any.

This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

ACCEPTS: an `Axes` instance

`set_clim(vmin=None, vmax=None)`  
set the norm limits for image scaling; if `vmin` is a length2 sequence, interpret it as `(vmin, vmax)` which is used to support setp

ACCEPTS: a length 2 sequence of floats

`set_clip_box(clipbox)`  
Set the artist’s clip `Bbox`. 

Chapter 24. collections
set_clip_on(b)
Set whether artist uses clipping.
When False artists will be visible out side of the axes which can lead to unexpected results.
ACCEPTS: [True | False]

set_clip_path(path, transform=None)
Set the artist’s clip path, which may be:
• a Patch (or subclass) instance
• a Path instance, in which case an optional Transform instance may be provided, which
  will be applied to the path before using it for clipping.
• None, to remove the clipping path
For efficiency, if the path happens to be an axis-aligned rectangle, this method will set the clip-
ping box to the corresponding rectangle and set the clipping path to None.
ACCEPTS: [ (Path, Transform) | Patch | None ]

set_cmap(cmap)
set the colormap for luminance data
ACCEPTS: a colormap or registered colormap name

set_color(c)
Set both the edgecolor and the facecolor.
ACCEPTS: matplotlib color arg or sequence of rgba tuples
See also:
set_facecolor(), set_edgecolor() For setting the edge or face color individually.

set_contains(picker)
Replace the contains test used by this artist. The new picker should be a callable function which
determines whether the artist is hit by the mouse event:

hit, props = picker(artist, mouseevent)
If the mouse event is over the artist, return hit = True and props is a dictionary of properties you
want returned with the contains test.
ACCEPTS: a callable function

set_dashes(ls)
alias for set_linestyle

set_edgecolor(c)
Set the edgecolor(s) of the collection. c can be a matplotlib color spec (all patches have same
color), or a sequence of specs; if it is a sequence the patches will cycle through the sequence.
If *c* is ‘face’, the edge color will always be the same as the face color. If it is ‘none’, the patch boundary will not be drawn.

**set_edgecolors**(*c*)  
alias for set_edgecolor

**set_facecolor**(*c*)  
Set the facecolor(s) of the collection. *c* can be a matplotlib color spec (all patches have same color), or a sequence of specs; if it is a sequence the patches will cycle through the sequence.

If *c* is ‘none’, the patch will not be filled.

**set_facecolors**(*c*)  
alias for set_facecolor

**set_figure**(*fig*)  
Set the *Figure* instance the artist belongs to.

**set_gid**(*gid*)  
Sets the (group) id for the artist

**set_hatch**(*hatch*)  
Set the hatching pattern

*hatch* can be one of:

```
/ - diagonal hatching  
\ - back diagonal  
| - vertical  
- - horizontal  
+ - crossed  
x - crossed diagonal  
o - small circle  
o - large circle  
. - dots  
* - stars
```

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

**set_label**(*s*)  
Set the label to *s* for auto legend.
ACCEPTS: string or anything printable with ‘%s’ conversion.

**set_linestyle**(ls)
Set the linestyle(s) for the collection.

<table>
<thead>
<tr>
<th>linestyle</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'-' or 'solid'</td>
<td>solid line</td>
</tr>
<tr>
<td>'−−' or 'dashed'</td>
<td>dashed line</td>
</tr>
<tr>
<td>'-.' or 'dashdot'</td>
<td>dash-dotted line</td>
</tr>
<tr>
<td>':' or 'dotted'</td>
<td>dotted line</td>
</tr>
</tbody>
</table>

Alternatively a dash tuple of the following form can be provided:

```
(offset, onoffseq),
```

where onoffseq is an even length tuple of on and off ink in points.

ACCEPTS: ['solid'] | ‘dashed’, ‘dashdot’, ‘dotted’ | (offset, on-off-dash-seq) | '−' | '−−' |
| '−.' | ':' | 'None' | ' ' | ''

**Parameters ls** : { '−', '−−', '−.', ':' } and more see description

The line style.

**set_linestyles**(ls)
alias for set_linestyle

**set_linewidth**(lw)
Set the linewidth(s) for the collection. lw can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence

ACCEPTS: float or sequence of floats

**set_linewidths**(lw)
alias for set_linewidth

**set_lw**(lw)
alias for set_linewidth

**set_norm**(norm)
set the normalization instance

**set_offset_position**(offset_position)
Set how offsets are applied. If offset_position is ‘screen’ (default) the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is ‘data’, the offset is applied before the master transform, i.e., the offsets are in data coordinates.

**set_offsets**(offsets)
Set the offsets for the collection. offsets can be a scalar or a sequence.

ACCEPTS: float or sequence of floats
**set_path_effects**(*path_effects*)

set path_effects, which should be a list of instances of matplotlib.path._Base class or its derivatives.

**set_paths**(*verts, closed=True*)

This allows one to delay initialization of the vertices.

**set_picker**(*picker*)

Set the epsilon for picking used by this artist

*picker* can be one of the following:

- *None*: picking is disabled for this artist (default)
- A boolean: if *True* then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist
- A float: if picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if it’s data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event
- A function: if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

  ```python
  hit, props = picker(artist, mouseevent)
  ```

  to determine the hit test. if the mouse event is over the artist, return *hit=True* and props is a dictionary of properties you want added to the PickEvent attributes.

ACCEPTS: [None|float|boolean|callable]

**set_pickradius**(*pr*)

**set_rasterized**(*rasterized*)

Force rasterized (bitmap) drawing in vector backend output.

Defaults to *None*, which implies the backend’s default behavior

ACCEPTS: [True | False | None]

**set_sizes**(*sizes, dpi=72.0*)

Set the sizes of each member of the collection.

**Parameters**

- **sizes**: ndarray or *None*

  The size to set for each element of the collection. The value is the ‘area’ of the element.

- **dpi**: float

  The dpi of the canvas. Defaults to 72.0.

**set_sketch_params**(*scale=None, length=None, randomness=None*)

Sets the sketch parameters.
Parameters scale : float, optional
The amplitude of the wiggle perpendicular to the source line, in pixels. If
scale is None, or not provided, no sketch filter will be provided.

length : float, optional
The length of the wiggle along the line, in pixels (default 128.0)

randomness : float, optional
The scale factor by which the length is shrunken or expanded (default 16.0)

set_snap\(\text{\texttt{(snap)}}\)
Sets the snap setting which may be:

- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel
center

Only supported by the Agg and MacOSX backends.

set_transform\(\text{\texttt{(t)}}\)
Set the Transform instance used by this artist.

ACCEPTS: Transform instance

set_url\(\text{\texttt{(url)}}\)
Sets the url for the artist

ACCEPTS: a url string

set_urls\(\text{\texttt{(urls)}}\)

set_verts\(\text{\texttt{(verts, closed=True)}}\)
This allows one to delay initialization of the vertices.

set_verts_and_codes\(\text{\texttt{(verts, codes)}}\)
This allows one to initialize vertices with path codes.

set_visible\(\text{\texttt{(b)}}\)
Set the artist’s visibility.

ACCEPTS: [True | False]

set_zorder\(\text{\texttt{(level)}}\)
Set the zorder for the artist. Artists with lower zorder values are drawn first.

ACCEPTS: any number

stale
If the artist is ‘stale’ and needs to be re-drawn for the output to match the internal state of the
artist.
**sticky_edges**

x and y sticky edge lists.

When performing autoscaling, if a data limit coincides with a value in the corresponding sticky_edges list, then no margin will be added—the view limit “sticks” to the edge. A typical usecase is histograms, where one usually expects no margin on the bottom edge (0) of the histogram.

This attribute cannot be assigned to; however, the x and y lists can be modified in place as needed.

**Examples**

```python
>>> artist.sticky_edges.x[:] = (xmin, xmax)
>>> artist.sticky_edges.y[:] = (ymin, ymax)
```

**to_rgba**(x, alpha=None, bytes=False, norm=True)

Return a normalized rgba array corresponding to x.

In the normal case, x is a 1-D or 2-D sequence of scalars, and the corresponding ndarray of rgba values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already rgb or rgba, such as might have been read from an image file. If x is an ndarray with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an rgb or rgba array, and no mapping will be done. If the last dimension is 3, the alpha kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the alpha kwarg is ignored; it does not replace the pre-existing alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if bytes is False (default), the rgba array will be floats in the 0-1 range; if it is True, the returned rgba array will be uint8 in the 0 to 255 range.

If norm is False, no normalization of the input data is performed, and it is assumed to already be in the range (0-1).

Note: this method assumes the input is well-behaved; it does not check for anomalies such as x being a masked rgba array, or being an integer type other than uint8, or being a floating point rgba array with values outside the 0-1 range.

**update**(props)

Update the properties of this Artist from the dictionary prop.

**update_from**(other)

copy properties from other to self

**update_scalarmappable**()

If the scalar mappable array is not none, update colors from scalar data

**zorder** = 0
class matplotlib.collections.QuadMesh(meshWidth, meshHeight, coordinates, antialiased=True, shading='flat', **kwargs)

Bases: matplotlib.collections.Collection

Class for the efficient drawing of a quadrilateral mesh.

A quadrilateral mesh consists of a grid of vertices. The dimensions of this array are \((\text{meshWidth} + 1, \text{meshHeight} + 1)\). Each vertex in the mesh has a different set of “mesh coordinates” representing its position in the topology of the mesh. For any values \((m, n)\) such that \(0 \leq m \leq \text{meshWidth}\) and \(0 \leq n \leq \text{meshHeight}\), the vertices at mesh coordinates \((m, n), (m, n + 1), (m + 1, n + 1), \text{and} (m + 1, n)\) form one of the quadrilaterals in the mesh. There are thus \((\text{meshWidth} \times \text{meshHeight})\) quadrilaterals in the mesh. The mesh need not be regular and the polygons need not be convex.

A quadrilateral mesh is represented by a \((2 \times ((\text{meshWidth} + 1) \times (\text{meshHeight} + 1)))\) numpy array \(coordinates\), where each row is the \(x\) and \(y\) coordinates of one of the vertices. To define the function that maps from a data point to its corresponding color, use the \(set_cmap()\) method. Each of these arrays is indexed in row-major order by the mesh coordinates of the vertex (or the mesh coordinates of the lower left vertex, in the case of the colors).

For example, the first entry in \(coordinates\) is the coordinates of the vertex at mesh coordinates \((0, 0)\), then the one at \((0, 1)\), then at \((0, 2)\) .. \((0, \text{meshWidth})\), \((1, 0)\), \((1, 1)\), and so on.

\(shading\) may be ‘flat’, or ‘gouraud’

**add_callback**(*func*)

- Adds a callback function that will be called whenever one of the Artist’s properties changes.

- Returns an \(id\) that is useful for removing the callback with \(remove_callback()\) later.

**add_checker**(*checker*)

- Add an entry to a dictionary of boolean flags that are set to True when the mappable is changed.

aname = ‘Artist’

**autoscale**()

- Autoscale the scalar limits on the norm instance using the current array

**autoscale_None**()

- Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

**axes**

- The Axes instance the artist resides in, or None.

**changed**()

- Call this whenever the mappable is changed to notify all the callbackSM listeners to the ‘changed’ signal

**check_update**(*checker*)

- If mappable has changed since the last check, return True; else return False

**contains**(*mouseevent*)

- Test whether the mouse event occurred in the collection.

- Returns True | False, \(\text{dict}(\text{ind}=\text{itemlist})\), where every item in itemlist contains the event.
static convert_mesh_to_paths(meshWidth, meshHeight, coordinates)
Converts a given mesh into a sequence of matplotlib.path.Path objects for easier rendering by backends that do not directly support quadmeshes.
This function is primarily of use to backend implementers.

convert_mesh_to_triangles(meshWidth, meshHeight, coordinates)
Converts a given mesh into a sequence of triangles, each point with its own color. This is useful for experiments using draw_qouraud_triangle.

convert_xunits(x)
For artists in an axes, if the xaxis has units support, convert x using xaxis unit type

calculate_yunits(y)
For artists in an axes, if the yaxis has units support, convert y using yaxis unit type

draw(artist, renderer, *args, **kwargs)

findobj(match=None, include_self=True)
Find artist objects.
Recursively find all Artist instances contained in self.
m can be

• None: return all objects contained in artist.
• function with signature boolean = match(artist) used to filter matches
• class instance: e.g., Line2D. Only return artists of class type.
If include_self is True (default), include self in the list to be checked for a match.

format_cursor_data(data)
Return cursor data string formatted.

get_agg_filter()
return filter function to be used for agg filter

get_alpha()
Return the alpha value used for blending - not supported on all backends

get_animated()
Return the artist’s animated state

get_array()
Return the array

get_axes()
Return the Axes instance the artist resides in, or None.
This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

get_children()
Return a list of the child Artist’s this :class:`Artist` contains.
get_clim()
    return the min, max of the color limits for image scaling

g_get_clip_box()
    Return artist clipbox

g_get_clip_on()
    Return whether artist uses clipping

g_get_clip_path()
    Return artist clip path

g_get_cmap()
    return the colormap

g_get_contains()
    Return the _contains test used by the artist, or None for default.

g_get_cursor_data(event)
    Get the cursor data for a given event.

g_get_dashes()

g_get_datalim(transData)


g_get_edgecolor()


g_get_edgecolors()


g_get_facecolor()


g_get_facecolors()


g_get_figure()
    Return the Figure instance the artist belongs to.

g_get_fill()
    return whether fill is set

g_get_gid()
    Returns the group id

g_get_hatch()
    Return the current hatching pattern

g_get_label()
    Get the label used for this artist in the legend.


g_get_linestyle()
get_linestyles()

get_linewidth()

get_linewidths()

get_offset_position()
   Returns how offsets are applied for the collection. If offset_position is ‘screen’, the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is ‘data’, the offset is applied before the master transform, i.e., the offsets are in data coordinates.

get_offset_transform()

get_offsets()
   Return the offsets for the collection.

get_path_effects()

get_paths()

get_picker()
   Return the picker object used by this artist

get_pickradius()

get_rasterized()
   return True if the artist is to be rasterized

get_sketch_params()
   Returns the sketch parameters for the artist.

   Returns sketch_params : tuple or None
   A 3-tuple with the following elements:
   • scale: The amplitude of the wiggle perpendicular to the source line.
   • length: The length of the wiggle along the line.
   • randomness: The scale factor by which the length is shrunken or expanded.
   May return None if no sketch parameters were set.

get_snap()
   Returns the snap setting which may be:
   • True: snap vertices to the nearest pixel center
   • False: leave vertices as-is
• None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

get_transform()  
Return the Transform instance used by this artist.

get_transformed_clip_path_and_affine()  
Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

get_transforms()  

get_url()  
Returns the url

get_urls()  

get_visible()  
Return the artist’s visibility

get_window_extent(renderer)  

get_zorder()  
Return the Artist’s zorder.

have_units()  
Return True if units are set on the x or y axes

hitlist(event)  
List the children of the artist which contain the mouse event event.

is_figure_set()  
Returns True if the artist is assigned to a Figure.

is_transform_set()  
Returns True if Artist has a transform explicitly set.

mouseover

pchanged()  
Fire an event when property changed, calling all of the registered callbacks.

pick(mouseevent)  
Process pick event

each child artist will fire a pick event if mouseevent is over the artist and the artist has picker set

pickable()  
Return True if Artist is pickable.
properties()
return a dictionary mapping property name -> value for all Artist props

remove()
Remove the artist from the figure if possible. The effect will not be visible until the figure is
redrawn, e.g., with matplotlib.axes.Axes.draw_idle(). Call matplotlib.axes.Axes.relim() to update the axes limits if desired.

Note: relim() will not see collections even if the collection was added to axes with autolim =
True.

Note: there is no support for removing the artist’s legend entry.

remove_callback(oid)
Remove a callback based on its id.

See also:

add_callback() For adding callbacks

set(**kwargs)
A property batch setter. Pass kwargs to set properties.

set_agg_filter(filter_func)
set agg_filter function.

set_alpha(alpha)
Set the alpha tranparencies of the collection. alpha must be a float or None.

ACCEPTS: float or None

set_animated(b)
Set the artist’s animation state.

ACCEPTS: [True | False]

set_antialiased(aa)
Set the antialiasing state for rendering.

ACCEPTS: Boolean or sequence of booleans

set_antialiaseds(aa)
alias for set_antialiased

set_array(A)
Set the image array from numpy array A

set_axes(axes)
Set the Axes instance in which the artist resides, if any.

This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

ACCEPTS: an Axes instance

set_clim(vmin=None, vmax=None)
set the norm limits for image scaling; if vmin is a length2 sequence, interpret it as (vmin,
vmax) which is used to support setp
set_clip_box(clipbox)
Set the artist’s clip Bbox.
ACCEPTS: a matplotlib.transforms.Bbox instance

set_clip_on(b)
Set whether artist uses clipping.
When False artists will be visible out side of the axes which can lead to unexpected results.
ACCEPTS: [True | False]

set_clip_path(path, transform=None)
Set the artist’s clip path, which may be:
- a Patch (or subclass) instance
- a Path instance, in which case an optional Transform instance may be provided, which will be applied to the path before using it for clipping.
- None, to remove the clipping path
For efficiency, if the path happens to be an axis-aligned rectangle, this method will set the clipping box to the corresponding rectangle and set the clipping path to None.
ACCEPTS: [ (Path, Transform) | Patch | None ]

set_cmap(cmap)
set the colormap for luminance data
ACCEPTS: a colormap or registered colormap name

set_color(c)
Set both the edgecolor and the facecolor.
ACCEPTS: matplotlib color arg or sequence of rgba tuples
See also:
set_facecolor(), set_edgecolor() For setting the edge or face color individually.

set_contains(picker)
Replace the contains test used by this artist. The new picker should be a callable function which determines whether the artist is hit by the mouse event:

hit, props = picker(artist, mouseevent)

If the mouse event is over the artist, return hit = True and props is a dictionary of properties you want returned with the contains test.
ACCEPTS: a callable function

set_dashes(ls)
alias for set_linestyle
**set_edgecolor(c)**
Set the edgecolor(s) of the collection. *c* can be a matplotlib color spec (all patches have same color), or a sequence of specs; if it is a sequence the patches will cycle through the sequence.

If *c* is ‘face’, the edge color will always be the same as the face color. If it is ‘none’, the patch boundary will not be drawn.

ACCEPTS: matplotlib color spec or sequence of specs

**set_edgecolors(c)**
alias for set_edgecolor

**set_facecolor(c)**
Set the facecolor(s) of the collection. *c* can be a matplotlib color spec (all patches have same color), or a sequence of specs; if it is a sequence the patches will cycle through the sequence.

If *c* is ‘none’, the patch will not be filled.

ACCEPTS: matplotlib color spec or sequence of specs

**set_facecolors(c)**
alias for set_facecolor

**set_figure(fig)**
Set the *Figure* instance the artist belongs to.

ACCEPTS: a *matplotlib.figure.Figure* instance

**set_gid(gid)**
Sets the (group) id for the artist

ACCEPTS: an id string

**set_hatch(hatch)**
Set the hatching pattern

*hatch* can be one of:

```
/  - diagonal hatching
\ - back diagonal
|  - vertical
-  - horizontal
+  - crossed
x  - crossed diagonal
o  - small circle
O  - large circle
. - dots
* - stars
```

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.
ACCEPES: [‘/’ | ‘\’ | ‘|’ | ‘-’ | ‘+’ | ‘x’ | ‘o’ | ‘O’ | ‘.’ | ‘*’]

`set_label(s)`
Set the label to s for auto legend.
ACCEPES: string or anything printable with ‘%s’ conversion.

`set_linestyle(ls)`
Set the linestyle(s) for the collection.

<table>
<thead>
<tr>
<th>linestyle</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'-' or 'solid'</td>
<td>solid line</td>
</tr>
<tr>
<td>'---' or 'dashed'</td>
<td>dashed line</td>
</tr>
<tr>
<td>':' or 'dashdot'</td>
<td>dash-dotted line</td>
</tr>
<tr>
<td>'dotted'</td>
<td>dotted line</td>
</tr>
</tbody>
</table>

Alternatively a dash tuple of the following form can be provided:

```
(offset, onoffseq),
```

where onoffseq is an even length tuple of on and off ink in points.

ACCEPES: ['solid’ | ‘dashed’, ‘dashdot’, ‘dotted’ | (offset, on-off-dash-seq) | ' -' | ' ---' | ' .-' | ':' | 'None' | '' | ' ']

Parameters `ls`: {‘-’, ‘-‘, ‘-.-’, ‘:’} and more see description

The line style.

`set_linestyles(ls)`
alias for `set_linestyle`

`set_linewidth(lw)`
Set the linewidth(s) for the collection. `lw` can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence.

ACCEPES: float or sequence of floats

`set_linewidths(lw)`
alias for `set_linewidth`

`set_lw(lw)`
alias for `set_linewidth`

`set_norm(norm)`
set the normalization instance

`set_offset_position(offset_position)`
Set how offsets are applied. If `offset_position` is ‘screen’ (default) the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If `offset_position` is ‘data’, the offset is applied before the master transform, i.e., the offsets are in data coordinates.

`set_offsets(offsets)`
Set the offsets for the collection. `offsets` can be a scalar or a sequence.
set_path_effects(path_effects)
    set path effects, which should be a list of instances of matplotlib.patheffect._Base class or its derivatives.

set_paths()

set_picker(picker)
    Set the epsilon for picking used by this artist

    picker can be one of the following:
    
    • **None**: picking is disabled for this artist (default)
    
    • A boolean: if True then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist
    
    • A float: if picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if it’s data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event
    
    • A function: if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

        ```python
        hit, props = picker(artist, mouseevent)
        ```

        to determine the hit test. if the mouse event is over the artist, return hit=True and props is a dictionary of properties you want added to the PickEvent attributes.

    ACCEPTS: [None|float|boolean|callable]

set_pickradius(pr)

set_rasterized(rasterized)
    Force rasterized (bitmap) drawing in vector backend output.

    Defaults to None, which implies the backend’s default behavior

    ACCEPTS: [True | False | None]

set_sketch_params(scale=None, length=None, randomness=None)
    Sets the sketch parameters.

    Parameters
    scale : float, optional
        The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is None, or not provided, no sketch filter will be provided.

    length : float, optional
        The length of the wiggle along the line, in pixels (default 128.0)

    randomness : float, optional
The scale factor by which the length is shrunken or expanded (default 16.0)

```python
set_snap(snapshot)
```
Sets the snap setting which may be:
- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

```python
set_transform(transform)
```
Set the `Transform` instance used by this artist.

ACCEPTS: `Transform` instance

```python
set_url(url)
```
Sets the url for the artist

ACCEPTS: a url string

```python
set_urls(urls)
```

```python
set_visible(bool)
```
Set the artist’s visibility.

ACCEPTS: [True | False]

```python
set_zorder(level)
```
Set the zorder for the artist. Artists with lower zorder values are drawn first.

ACCEPTS: any number

```python
stale
```
If the artist is ‘stale’ and needs to be re-drawn for the output to match the internal state of the artist.

```python
sticky_edges
```
x and y sticky edge lists.

When performing autoscaling, if a data limit coincides with a value in the corresponding sticky_edges list, then no margin will be added—the view limit “sticks” to the edge. A typical use case is histograms, where one usually expects no margin on the bottom edge (0) of the histogram.

This attribute cannot be assigned to; however, the x and y lists can be modified in place as needed.
Examples

```python
>>> artist.sticky_edges.x[:] = (xmin, xmax)
>>> artist.sticky_edges.y[:] = (ymin, ymax)
```

to_rgba($x$, alpha=None, bytes=False, norm=True)

Return a normalized rgba array corresponding to $x$.

In the normal case, $x$ is a 1-D or 2-D sequence of scalars, and the corresponding ndarray of rgba values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already rgb or rgba, such as might have been read from an image file. If $x$ is an ndarray with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an rgb or rgba array, and no mapping will be done. If the last dimension is 3, the alpha kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the alpha kwarg is ignored; it does not replace the pre-existing alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if bytes is False (default), the rgba array will be floats in the 0-1 range; if it is True, the returned rgba array will be uint8 in the 0 to 255 range.

If norm is False, no normalization of the input data is performed, and it is assumed to already be in the range (0-1).

Note: this method assumes the input is well-behaved; it does not check for anomalies such as $x$ being a masked rgba array, or being an integer type other than uint8, or being a floating point rgba array with values outside the 0-1 range.

update(props)

Update the properties of this Artist from the dictionary prop.

update_from(other)

copy properties from other to self

update_scalarmappable()

If the scalar mappable array is not none, update colors from scalar data

zorder = 0

class matplotlib.collections.RegularPolyCollection(numsides, rotation=0, sizes=(1, ), **kwargs)

Bases: matplotlib.collections._CollectionWithSizes

Draw a collection of regular polygons with numsides.

numsides the number of sides of the polygon

rotation the rotation of the polygon in radians

sizes gives the area of the circle circumscribing the regular polygon in points^2

Valid Collection keyword arguments:

- `edgecolors`: None
• **facecolors**: None
• **linewidths**: None
• **antialiaseds**: None
• **offsets**: None
• **transOffset**: transforms.IdentityTransform()
• **norm**: None (optional for matplotlib.cm.ScalarMappable)
• **cmap**: None (optional for matplotlib.cm.ScalarMappable)

*offsets* and *transOffset* are used to translate the patch after rendering (default no offsets)

If any of *edgecolors*, *facecolors*, *linewidths*, *antialiaseds* are None, they default to their *matplotlib.rcParams* patch setting, in sequence form.

Example: see examples/dynamic_collection.py for complete example:

```python
offsets = np.random.rand(20,2)
facecolors = [cm.jet(x) for x in np.random.rand(20)]
black = (0,0,0,1)

collection = RegularPolyCollection(
    numsides=5, # a pentagon
    rotation=0, sizes=(50,),
    facecolors = facecolors,
    edgecolors = (black,),
    linewidths = (1,),
    offsets = offsets,
    transOffset = ax.transData,
)
```

**add_callback**(*func*)

Adds a callback function that will be called whenever one of the *Artist*’s properties changes.

Returns an *id* that is useful for removing the callback with *remove_callback()* later.

**add_checker**(*checker*)

Add an entry to a dictionary of boolean flags that are set to True when the mappable is changed.

**aname** = ‘*Artist*’

**autoscale**()

Autoscale the scalar limits on the norm instance using the current array

**autoscale_None**()

Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

**axes**

The *Axes* instance the artist resides in, or *None*. 

---

24.1. matplotlib.collections 1109
changed()
Call this whenever the mappable is changed to notify all the callback listeners to the 'changed' signal

check_update(checker)
If mappable has changed since the last check, return True; else return False

contains(mouseevent)
Test whether the mouse event occurred in the collection.
Returns True|False, dict(ind=itemlist), where every item in itemlist contains the event.

convert_xunits(x)
For artists in an axes, if the xaxis has units support, convert x using xaxis unit type

convert_yunits(y)
For artists in an axes, if the yaxis has units support, convert y using yaxis unit type

draw(artist, renderer, *args, **kwargs)

findobj(match=None, include_self=True)
Find artist objects.
Recursively find all Artist instances contained in self.

match can be
- None: return all objects contained in artist.
- function with signature boolean = match(artist) used to filter matches
- class instance: e.g., Line2D. Only return artists of class type.

If include_self is True (default), include self in the list to be checked for a match.

format_cursor_data(data)
Return cursor data string formatted.

get_agg_filter()
return filter function to be used for agg filter

get_alpha()
Return the alpha value used for blending - not supported on all backends

get_animated()
Return the artist’s animated state

get_array()
Return the array

get_axes()
Return the Axes instance the artist resides in, or None.

This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

get_children()
Return a list of the child Artist’s this :class:`Artist` contains.
get_clim()
    return the min, max of the color limits for image scaling

get_clip_box()
    Return artist clipbox

get_clip_on()
    Return whether artist uses clipping

get_clip_path()
    Return artist clip path

get_cmap()
    return the colormap

get_contains()
    Return the _contains test used by the artist, or None for default.

get_cursor_data(event)
    Get the cursor data for a given event.

get_dashes()

get_datalim(transData)

get_edgecolor()

get_edgecolors()

get_facecolor()

get_facecolors()

get_figure()
    Return the Figure instance the artist belongs to.

get_fill()
    return whether fill is set

get_gid()
    Returns the group id

get_hatch()
    Return the current hatching pattern

get_label()
    Get the label used for this artist in the legend.

get_linestyle()
get_linestyles()

get_linewidth()

get_linewidths()

get_numsides()

get_offset_position()
Returns how offsets are applied for the collection. If offset_position is ‘screen’, the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is ‘data’, the offset is applied before the master transform, i.e., the offsets are in data coordinates.

get_offset_transform()

get_offsets()
Return the offsets for the collection.

get_path_effects()

get_paths()

get_picker()
Return the picker object used by this artist

get_pickradius()

get_rasterized()
return True if the artist is to be rasterized

get_rotation()

get_sizes()
Returns the sizes of the elements in the collection. The value represents the ‘area’ of the element.

Returns sizes : array
    The ‘area’ of each element.

get_sketch_params()
Returns the sketch parameters for the artist.

Returns sketch_params : tuple or None
    A 3-tuple with the following elements:
    • scale: The amplitude of the wiggle perpendicular to the source line.
• length: The length of the wiggle along the line.
• randomness: The scale factor by which the length is shrunken or expanded.

May return None if no sketch parameters were set.

get_snap()
Returns the snap setting which may be:
• True: snap vertices to the nearest pixel center
• False: leave vertices as-is
• None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

get_transform()
Return the Transform instance used by this artist.

get_transformed_clip_path_and_affine()
Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

get_transforms()

get_url()
Returns the url

get_urls()

get_visible()
Return the artist’s visiblity

get_window_extent(renderer)

get_zorder()
Return the Artist’s zorder.

have_units()
Return True if units are set on the x or y axes

hitlist(event)
List the children of the artist which contain the mouse event event.

is_figure_set()
Returns True if the artist is assigned to a Figure.

is_transform_set()
Returns True if Artist has a transform explicitly set.

mouseover
Fire an event when property changed, calling all of the registered callbacks.

Process pick event

each child artist will fire a pick event if mouseevent is over the artist and the artist has picker set

Return True if Artist is pickable.

return a dictionary mapping property name -> value for all Artist props

Remove the artist from the figure if possible. The effect will not be visible until the figure is redrawn, e.g., with matplotlib.axes.Axes.draw_idle(). Call matplotlib.axes.Axes.relim() to update the axes limits if desired.

Note: relim() will not see collections even if the collection was added to axes with autolim = True.

Note: there is no support for removing the artist’s legend entry.

Remove a callback based on its id.

See also:

add_callback() For adding callbacks

A property batch setter. Pass kwargs to set properties.

set_agg_filter(filter_func)

set antialiased state for rendering.

ACCEPTS: Boolean or sequence of booleans

Set the antialiased state for rendering.

ACCEPTS: Boolean or sequence of booleans

Set the image array from numpy array A
**set_axes**(axes)
Set the Axes instance in which the artist resides, if any.
This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.
ACCEPTS: an Axes instance

**set_clim**(vmin=None, vmax=None)
set the norm limits for image scaling; if vmin is a length2 sequence, interpret it as (vmin, vmax) which is used to support setp
ACCEPTS: a length 2 sequence of floats

**set_clip_box**(clipbox)
Set the artist’s clip Bbox.
ACCEPTS: a matplotlib.transforms.Bbox instance

**set_clip_on**(b)
Set whether artist uses clipping.
When False artists will be visible out side of the axes which can lead to unexpected results.
ACCEPTS: [True | False]

**set_clip_path**(path, transform=None)
Set the artist’s clip path, which may be:
- a Patch (or subclass) instance
- a Path instance, in which case an optional Transform instance may be provided, which will be applied to the path before using it for clipping.
- None, to remove the clipping path
For efficiency, if the path happens to be an axis-aligned rectangle, this method will set the clipping box to the corresponding rectangle and set the clipping path to None.
ACCEPTS: [ (Path, Transform) | Patch | None ]

**set_cmap**(cmap)
set the colormap for luminance data
ACCEPTS: a colormap or registered colormap name

**set_color**(c)
Set both the edgecolor and the facecolor.
ACCEPTS: matplotlib color arg or sequence of rgba tuples
See also:

**set_facecolor**, **set_edgecolor** For setting the edge or face color individually.

**set_contains**(picker)
Replace the contains test used by this artist. The new picker should be a callable function which determines whether the artist is hit by the mouse event:
hit, props = picker(artist, mouseevent)

If the mouse event is over the artist, return hit = True and props is a dictionary of properties you want returned with the contains test.

ACCEPtS: a callable function

set_dashes(ls)
alias for set_linestyle

set_edgecolor(c)
Set the edgecolor(s) of the collection. c can be a matplotlib color spec (all patches have same color), or a sequence of specs; if it is a sequence the patches will cycle through the sequence.

If c is ‘face’, the edge color will always be the same as the face color. If it is ‘none’, the patch boundary will not be drawn.

ACCEPtS: matplotlib color spec or sequence of specs

set_edgecolors(c)
alias for set_edgecolor

set_facecolor(c)
Set the facecolor(s) of the collection. c can be a matplotlib color spec (all patches have same color), or a sequence of specs; if it is a sequence the patches will cycle through the sequence.

If c is ‘none’, the patch will not be filled.

ACCEPtS: matplotlib color spec or sequence of specs

set_facecolors(c)
alias for set_facecolor

set_figure(fig)
Set the Figure instance the artist belongs to.

ACCEPtS: a matplotlib.figure.Figure instance

set_gid(gid)
Sets the (group) id for the artist

ACCEPtS: an id string

set_hatch(hatch)
Set the hatching pattern

hatch can be one of:

/  - diagonal hatching
\ - back diagonal
| - vertical
- - horizontal
+ - crossed
x - crossed diagonal
o - small circle
O - large circle
Letters can be combined, in which case all the specified hatchings are done. If same letter
repeats, it increases the density of hatching of that pattern.

Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Unlike other properties such as linewidth and colors, hatching can only be specified for the
collection as a whole, not separately for each member.

ACCEPTS: ['\'' | '\\' | '|' | '-.' | '+' | 'x' | 'o' | 'O' | ':' | '*']

**set_label(s)**
Set the label to s for auto legend.

ACCEPTS: string or anything printable with '%s' conversion.

**set_linestyle(ls)**
Set the linestyle(s) for the collection.

<table>
<thead>
<tr>
<th>linestyle</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'-' or 'solid'</td>
<td>solid line</td>
</tr>
<tr>
<td>'--' or 'dashed'</td>
<td>dashed line</td>
</tr>
<tr>
<td>'-.' or 'dashdot'</td>
<td>dash-dotted line</td>
</tr>
<tr>
<td>':' or 'dotted'</td>
<td>dotted line</td>
</tr>
</tbody>
</table>

Alternatively a dash tuple of the following form can be provided:

```
(offset, onoffseq),
```

where onoffseq is an even length tuple of on and off ink in points.

ACCEPTS: ['solid' | 'dashed', 'dashdot', 'dotted'] (offset, on-off-dash-seq) | '-' | '--' |

| Parameters ls | {)+'-','--','-.'|' ':'|'None'|''|'' |

The line style.

**set_linestyles(ls)**
alias for set_linestyle

**set_linewidth(lw)**
Set the linewidth(s) for the collection. lw can be a scalar or a sequence; if it is a sequence the
patches will cycle through the sequence

ACCEPTS: float or sequence of floats

**set_linewidths(lw)**
alias for set_linewidth

**set_lw(lw)**
alias for set_linewidth
**set_norm**(norm)

set the normalization instance

**set_offset_position**(offset_position)

Set how offsets are applied. If offset_position is ‘screen’ (default) the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is ‘data’, the offset is applied before the master transform, i.e., the offsets are in data coordinates.

**set_offsets**(offsets)

Set the offsets for the collection. offsets can be a scalar or a sequence.

ACCEPTS: float or sequence of floats

**set_path_effects**(path_effects)

Set path_effects, which should be a list of instances of matplotlib.path{}Effect._Base class or its derivatives.

**set_paths**()

**set_picker**(picker)

Set the epsilon for picking used by this artist

picker can be one of the following:

- None: picking is disabled for this artist (default)
- A boolean: if True then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist
- A float: if picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if it’s data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event
- A function: if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```python
hit, props = picker(artist, mouseevent)
```

to determine the hit test. if the mouse event is over the artist, return hit=True and props is a dictionary of properties you want added to the PickEvent attributes.

ACCEPTS: [None|float|boolean|callable]

**set_pickradius**(pr)

**set_rasterized**(rasterized)

Force rasterized (bitmap) drawing in vector backend output.

Defaults to None, which implies the backend’s default behavior

ACCEPTS: [True | False | None]
set_sizes(sizes, dpi=72.0)
Set the sizes of each member of the collection.

Parameters sizes : ndarray or None
The size to set for each element of the collection. The value is the ‘area’ of
the element.

dpi : float
The dpi of the canvas. Defaults to 72.0.

set_sketch_params(scale=None, length=None, randomness=None)
Sets the sketch parameters.

Parameters scale : float, optional
The amplitude of the wiggle perpendicular to the source line, in pixels. If
scale is None, or not provided, no sketch filter will be provided.

length : float, optional
The length of the wiggle along the line, in pixels (default 128.0)

randomness : float, optional
The scale factor by which the length is shrunken or expanded (default 16.0)

set_snap(snapshot)
Sets the snap setting which may be:

- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel
center

Only supported by the Agg and MacOSX backends.

set_transform(t)
Set the Transform instance used by this artist.

ACCEPTS: Transform instance

set_url(url)
Sets the url for the artist

ACCEPTS: a url string

set_urls(urls)

set_visible(b)
Set the artist’s visibility.

ACCEPTS: [True | False]
**set_zorder**(level)
Set the zorder for the artist. Artists with lower zorder values are drawn first.

ACCEPTS: any number

**stale**
If the artist is ‘stale’ and needs to be re-drawn for the output to match the internal state of the artist.

**sticky_edges**
x and y sticky edge lists.
When performing autoscaling, if a data limit coincides with a value in the corresponding sticky_edges list, then no margin will be added—the view limit “sticks” to the edge. A typical use case is histograms, where one usually expects no margin on the bottom edge (0) of the histogram.

This attribute cannot be assigned to; however, the x and y lists can be modified in place as needed.

**Examples**

```python
>>> artist.sticky_edges.x[:] = (xmin, xmax)
>>> artist.sticky_edges.y[:] = (ymin, ymax)
```

**to_rgba**(x, alpha=None, bytes=False, norm=True)
Return a normalized rgba array corresponding to x.

In the normal case, x is a 1-D or 2-D sequence of scalars, and the corresponding ndarray of rgba values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already rgb or rgba, such as might have been read from an image file. If x is an ndarray with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an rgb or rgba array, and no mapping will be done. If the last dimension is 3, the alpha kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the alpha kwarg is ignored; it does not replace the pre-existing alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if bytes is False (default), the rgba array will be floats in the 0-1 range; if it is True, the returned rgba array will be uint8 in the 0 to 255 range.

If norm is False, no normalization of the input data is performed, and it is assumed to already be in the range (0-1).

Note: this method assumes the input is well-behaved; it does not check for anomalies such as x being a masked rgba array, or being an integer type other than uint8, or being a floating point rgba array with values outside the 0-1 range.

**update**(props)
Update the properties of this Artist from the dictionary prop.

**update_from**(other)
copy properties from other to self
**update_scalarmappable()**
If the scalar mappable array is not none, update colors from scalar data

**zorder = 0**

class matplotlib.collections.StarPolygonCollection(numsides, rotation=0, sizes=(1, ), **kwargs)
Bases: matplotlib.collections.RegularPolyCollection
Draw a collection of regular stars with numsides points.

*numsides* the number of sides of the polygon

*rotation* the rotation of the polygon in radians

*sizes* gives the area of the circle circumscribing the regular polygon in points^2

Valid Collection keyword arguments:

- *edgecolors*: None
- *facecolors*: None
- *linewidths*: None
- *antialiaseds*: None
- *offsets*: None
- *transOffset*: transforms.IdentityTransform()
- *norm*: None (optional for matplotlib.cm.ScalarMappable)
- *cmap*: None (optional for matplotlib.cm.ScalarMappable)

*offsets* and *transOffset* are used to translate the patch after rendering (default no offsets)

If any of *edgecolors*, *facecolors*, *linewidths*, *antialiaseds* are None, they default to their matplotlib.rcParams patch setting, in sequence form.

Example: see examples/dynamic_collection.py for complete example:

```python
offsets = np.random.rand(20,2)
facecolors = [cm.jet(x) for x in np.random.rand(20)]
black = (0,0,0,1)

collection = RegularPolyCollection(
    numsides=5, # a pentagon
    rotation=0, sizes=(50,),
    facecolors = facecolors,
    edgecolors = (black,),
    linewidths = (1,),
    offsets = offsets,
    transOffset = ax.transData,
)
```

**add_callback(func)**
Adds a callback function that will be called whenever one of the Artist's properties changes.
Returns an id that is useful for removing the callback with remove_callback() later.

add_checker(checker)
Add an entry to a dictionary of boolean flags that are set to True when the mappable is changed.

aname = ‘Artist’

autoscale()
Autoscale the scalar limits on the norm instance using the current array

autoscale_None()
Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

axes
The Axes instance the artist resides in, or None.

changed()
Call this whenever the mappable is changed to notify all the callbackSM listeners to the ‘changed’ signal

check_update(checker)
If mappable has changed since the last check, return True; else return False

contains(mouseevent)
Test whether the mouse event occurred in the collection.
Returns True | False, dict(ind=itemlist), where every item in itemlist contains the event.

convert_xunits(x)
For artists in an axes, if the xaxis has units support, convert x using xaxis unit type

convert_yunits(y)
For artists in an axes, if the yaxis has units support, convert y using yaxis unit type

draw(artist, renderer, *args, **kwargs)

findobj(match=None, include_self=True)
Find artist objects.
Recursively find all Artist instances contained in self.

match can be
• None: return all objects contained in artist.
• function with signature boolean = match(artist) used to filter matches
• class instance: e.g., Line2D. Only return artists of class type.

If include_self is True (default), include self in the list to be checked for a match.

format_cursor_data(data)
Return cursor data string formatted.
get_agg_filter()
    return filter function to be used for agg filter

get_alpha()
    Return the alpha value used for blending - not supported on all backends

get_animated()
    Return the artist’s animated state

get_array()
    Return the array

get_axes()
    Return the Axes instance the artist resides in, or None.
    
    This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

get_children()
    Return a list of the child Artist’s this :class:`Artist contains.

get_clim()
    return the min, max of the color limits for image scaling

get_clip_box()
    Return artist clipbox

get_clip_on()
    Return whether artist uses clipping

get_clip_path()
    Return artist clip path

get_cmap()
    return the colormap

get_contains()
    Return the _contains test used by the artist, or None for default.

get_cursor_data(event)
    Get the cursor data for a given event.

get_dashes()

get_datalim(transData)

get_edgecolor()

get_edgecolors()

get_facecolor()
get_facecolors()

get_figure()
    Return the Figure instance the artist belongs to.

get_fill()
    return whether fill is set

get_gid()
    Returns the group id

get_hatch()
    Return the current hatching pattern

get_label()
    Get the label used for this artist in the legend.

get_linestyle()

get_linestyles()

get_linewidth()

get_linewidths()

get_numsides()

get_offset_position()
    Returns how offsets are applied for the collection. If offset_position is ‘screen’, the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is ‘data’, the offset is applied before the master transform, i.e., the offsets are in data coordinates.

get_offset_transform()

get_offsets()
    Return the offsets for the collection.

get_path_effects()

get_paths()

get_picker()
    Return the picker object used by this artist

get_pickradius()
get_rasterized()
    return True if the artist is to be rasterized

def get_rotation()

def get_sizes()
    Returns the sizes of the elements in the collection. The value represents the ‘area’ of the element.

    Returns sizes : array
        The ‘area’ of each element.

def get_sketch_params()
    Returns the sketch parameters for the artist.

    Returns sketch_params : tuple or None
        A 3-tuple with the following elements:
        • scale: The amplitude of the wiggle perpendicular to the source line.
        • length: The length of the wiggle along the line.
        • randomness: The scale factor by which the length is shrunken or expanded.

        May return None if no sketch parameters were set.

def get_snap()
    Returns the snap setting which may be:
        • True: snap vertices to the nearest pixel center
        • False: leave vertices as-is
        • None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

    Only supported by the Agg and MacOSX backends.

def get_transform()
    Return the Transform instance used by this artist.

def get_transformed_clip_path_and_affine()
    Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

def get_transforms()

def get_url()
    Returns the url

def get_urls()

def get_visible()
    Return the artist’s visibility
get_window_extent(renderer)

get_zorder()
   Return the Artist's zorder.

have_units()
   Return True if units are set on the x or y axes

hitlist(event)
   List the children of the artist which contain the mouse event event.

is_figure_set()
   Returns True if the artist is assigned to a Figure.

is_transform_set()
   Returns True if Artist has a transform explicitly set.

mouseover

pchanged()
   Fire an event when property changed, calling all of the registered callbacks.

pick(mouseevent)
   Process pick event

      each child artist will fire a pick event if mouseevent is over the artist and the artist has picker set

pickable()
   Return True if Artist is pickable.

properties()
   return a dictionary mapping property name -> value for all Artist props

remove()
   Remove the artist from the figure if possible. The effect will not be visible until the figure is redrawn, e.g., with matplotlib.axes.Axes.draw_idle(). Call matplotlib.axes.Axes.relim() to update the axes limits if desired.

      Note: reлим() will not see collections even if the collection was added to axes with autolim = True.

      Note: there is no support for removing the artist’s legend entry.

remove_callback(oid)
   Remove a callback based on its id.

See also:

   add_callback() For adding callbacks

set(**kwargs)
   A property batch setter. Pass kwargs to set properties.
set_agg_filter\( (\text{filter\_func}) \)
set agg_filter function.

set_alpha\( (\alpha) \)
Set the alpha transparencies of the collection. \( \alpha \) must be a float or None.

ACCEPTS: float or None

set_animated\( (b) \)
Set the artist's animation state.

ACCEPTS: [True | False]

set_antialiased\( (\text{aa}) \)
Set the antialiasing state for rendering.

ACCEPTS: Boolean or sequence of booleans

set_antialiaseds\( (\text{aa}) \)
alias for set_antialiased

set_array\( (A) \)
Set the image array from numpy array \( A \)

set_axes\( (\text{axes}) \)
Set the \textit{Axes} instance in which the artist resides, if any.

This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

ACCEPTS: an \textit{Axes} instance

set_clim\( (\text{vmin=None, vmax=None}) \)
set the norm limits for image scaling; if \( \text{vmin} \) is a length2 sequence, interpret it as \( (\text{vmin, vmax}) \) which is used to support setp

ACCEPTS: a length 2 sequence of floats

set_clip_box\( (\text{clipbox}) \)
Set the artist’s clip \textit{Bbox}.

ACCEPTS: a \textit{matplotlib.transforms.Bbox} instance

set_clip_on\( (b) \)
Set whether artist uses clipping.

When False artists will be visible out side of the axes which can lead to unexpected results.

ACCEPTS: [True | False]

set_clip_path\( (\text{path, transform=None}) \)
Set the artist’s clip path, which may be:

- a \textit{Patch} (or subclass) instance
- a \textit{Path} instance, in which case an optional \textit{Transform} instance may be provided, which will be applied to the path before using it for clipping.
- None, to remove the clipping path
For efficiency, if the path happens to be an axis-aligned rectangle, this method will set the clipping box to the corresponding rectangle and set the clipping path to None.

**set_cmap**

- **set the colormap for luminance data**
  - ACCEPTS: a colormap or registered colormap name

**set_color**

- Set both the edgecolor and the facecolor.
  - ACCEPTS: matplotlib color arg or sequence of rgba tuples
  - See also:
    - `set_facecolor()`, `set_edgecolor()` For setting the edge or face color individually.

**set_contains**

- Replace the contains test used by this artist. The new picker should be a callable function which determines whether the artist is hit by the mouse event:
  - `hit, props = picker(artist, mouseevent)`
  - If the mouse event is over the artist, return `hit = True` and `props` is a dictionary of properties you want returned with the contains test.
  - ACCEPTS: a callable function

**set_dashes**

- alias for set_linestyle

**set_edgecolor**

- Set the edgecolor(s) of the collection. `c` can be a matplotlib color spec (all patches have same color), or a sequence of specs; if it is a sequence the patches will cycle through the sequence.
  - If `c` is ‘face’, the edge color will always be the same as the face color. If it is ‘none’, the patch boundary will not be drawn.
  - ACCEPTS: matplotlib color spec or sequence of specs

**set_edgecolors**

- alias for set_edgecolor

**set_facecolor**

- Set the facecolor(s) of the collection. `c` can be a matplotlib color spec (all patches have same color), or a sequence of specs; if it is a sequence the patches will cycle through the sequence.
  - If `c` is ‘none’, the patch will not be filled.
  - ACCEPTS: matplotlib color spec or sequence of specs

**set_facecolors**

- alias for set_facecolor
set_figure(fig)
Set the Figure instance the artist belongs to.

ACCEPTS: a matplotlib.figure.Figure instance

set_gid(gid)
Sets the (group) id for the artist

ACCEPTS: an id string

set_hatch(hatch)
Set the hatching pattern

*hatch* can be one of:

<table>
<thead>
<tr>
<th>Hatch</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/</td>
<td>diagonal hatching</td>
</tr>
<tr>
<td>\</td>
<td>back diagonal</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>horizontal</td>
</tr>
<tr>
<td>+</td>
<td>crossed</td>
</tr>
<tr>
<td>x</td>
<td>crossed diagonal</td>
</tr>
<tr>
<td>o</td>
<td>small circle</td>
</tr>
<tr>
<td>O</td>
<td>large circle</td>
</tr>
<tr>
<td>.</td>
<td>dots</td>
</tr>
<tr>
<td>*</td>
<td>stars</td>
</tr>
</tbody>
</table>

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

ACCEPTS: ['/' | '\' | '|' | 'v' | '+' | 'x' | 'o' | 'O' | '.' | '*' ]

set_label(s)
Set the label to *s* for auto legend.

ACCEPTS: string or anything printable with ‘%s’ conversion.

set_linestyle(ls)
Set the linestyle(s) for the collection.

<table>
<thead>
<tr>
<th>Linestyle</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'-.' or 'solid'</td>
<td>solid line</td>
</tr>
<tr>
<td>'--' or 'dashed'</td>
<td>dashed line</td>
</tr>
<tr>
<td>'---' or 'dashdot'</td>
<td>dash-dotted line</td>
</tr>
<tr>
<td>':' or 'dotted'</td>
<td>dotted line</td>
</tr>
</tbody>
</table>

Alternatively a dash tuple of the following form can be provided:

```python
(offset, onoffseq),
```

where `onoffseq` is an even length tuple of on and off ink in points.
ACCEETS: ['solid', 'dashed', 'dashdot', 'dotted'] (offset, on-off-dash-seq) | '-' | '--' | '-.' | ':' | 'None' | ' ' | '
'

Parameters ls : { ' collectively see description

The line style.

\texttt{set_linestyles}(ls)
alias for set linestyle

\texttt{set_linewidth}(lw)
Set the linewidth(s) for the collection. lw can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence

ACCEETS: float or sequence of floats

\texttt{set_linewidths}(lw)
alias for set linewidth

\texttt{set_lw}(lw)
alias for set linewidth

\texttt{set_norm}(norm)
set the normalization instance

\texttt{set_offset_position}(offset_position)
Set how offsets are applied. If offset_position is ‘screen’ (default) the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is ‘data’, the offset is applied before the master transform, i.e., the offsets are in data coordinates.

\texttt{set_offsets}(offsets)
Set the offsets for the collection. offsets can be a scalar or a sequence.

ACCEETS: float or sequence of floats

\texttt{set_path_effects}(path_effects)
set path effects, which should be a list of instances of matplotlib.path.effect._Base class or its derivatives.

\texttt{set_paths}()

\texttt{set_picker}(picker)
Set the epsilon for picking used by this artist

picker can be one of the following:

- None: picking is disabled for this artist (default)
- A boolean: if True then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist
- A float: if picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if it’s data is within epsilon of the mouse event. For some artists like
lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event

- A function: if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

  ```python
  hit, props = picker(artist, mouseevent)
  ```

to determine the hit test. if the mouse event is over the artist, return \(\text{hit} = \text{True}\) and props is a dictionary of properties you want added to the PickEvent attributes.

ACCEPTS: [None|float|boolean|callable]

```python
set_pickradius(pr)
```

```python
set_rasterized(rasterized)
```

Force rasterized (bitmap) drawing in vector backend output.

Defaults to None, which implies the backend’s default behavior

ACCEPTS: [True | False | None]

```python
set_sizes(sizes, dpi=72.0)
```

Set the sizes of each member of the collection.

**Parameters**

- **sizes**: ndarray or None
  
  The size to set for each element of the collection. The value is the ‘area’ of the element.

- **dpi**: float
  
  The dpi of the canvas. Defaults to 72.0.

```python
set_sketch_params(scale=None, length=None, randomness=None)
```

Sets the sketch parameters.

**Parameters**

- **scale**: float, optional
  
  The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is None, or not provided, no sketch filter will be provided.

- **length**: float, optional
  
  The length of the wiggle along the line, in pixels (default 128.0)

- **randomness**: float, optional
  
  The scale factor by which the length is shrunken or expanded (default 16.0)

```python
set_snap(snapsnap)
```

Sets the snap setting which may be:

- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
•None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

set_transform(t)
Set the Transform instance used by this artist.

ACCEPTS: Transform instance

set_url(url)
Sets the url for the artist

ACCEPTS: a url string

set_urls(urls)

set_visible(b)
Set the artist’s visibility.

ACCEPTS: [True | False]

set_zorder(level)
Set the zorder for the artist. Artists with lower zorder values are drawn first.

ACCEPTS: any number

stale
If the artist is ‘stale’ and needs to be re-drawn for the output to match the internal state of the artist.

sticky_edges
x and y sticky edge lists.

When performing autoscaling, if a data limit coincides with a value in the corresponding sticky_edges list, then no margin will be added—the view limit “sticks” to the edge. A typical usecase is histograms, where one usually expects no margin on the bottom edge (0) of the histogram.

This attribute cannot be assigned to; however, the x and y lists can be modified in place as needed.

Examples

```python
>>> artist.sticky_edges.x[:] = (xmin, xmax)
>>> artist.sticky_edges.y[:] = (ymin, ymax)
```

to_rgba(x, alpha=None, bytes=False, norm=True)
Return a normalized rgba array corresponding to x.

In the normal case, x is a 1-D or 2-D sequence of scalars, and the corresponding ndarray of rgba values will be returned, based on the norm and colormap set for this ScalarMappable.
There is one special case, for handling images that are already rgb or rgba, such as might have been read from an image file. If \( x \) is an ndarray with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an rgb or rgba array, and no mapping will be done. If the last dimension is 3, the \( \text{alpha} \) kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the \( \text{alpha} \) kwarg is ignored; it does not replace the pre-existing alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if \( \text{bytes} \) is \text{False} (default), the rgba array will be floats in the 0-1 range; if it is \text{True}, the returned rgba array will be uint8 in the 0 to 255 range.

If norm is \text{False}, no normalization of the input data is performed, and it is assumed to already be in the range (0-1).

Note: this method assumes the input is well-behaved; it does not check for anomalies such as \( x \) being a masked rgba array, or being an integer type other than uint8, or being a floating point rgba array with values outside the 0-1 range.

```
update(props)
```

Update the properties of this \text{Artist} from the dictionary \text{prop}.

```
update_from(other)
```

copy properties from other to self

```
update_scalarmappable()
```

If the scalar mappable array is not none, update colors from scalar data

```
zorder = 0
```

class \text{matplotlib.collections.TriMesh}(triangulation, **kwargs)

\text{Bases: \text{matplotlib.collections.Collection}}

Class for the efficient drawing of a triangular mesh using Gouraud shading.

A triangular mesh is a \text{Triangulation} object.

```
add_callback(func)
```

Adds a callback function that will be called whenever one of the \text{Artist}'s properties changes.

Returns an \text{id} that is useful for removing the callback with \text{remove_callback()} later.

```
add_checker(checker)
```

Add an entry to a dictionary of boolean flags that are set to \text{True} when the mappable is changed.

```
aname = 'Artist'
```

```
autoscale()
```

Autoscale the scalar limits on the norm instance using the current array

```
autoscale_None()
```

Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

```
axes
```

The \text{Axes} instance the artist resides in, or \text{None}. 

24.1. \text{matplotlib.collections} 1133
changed()
  Call this whenever the mappable is changed to notify all the callback listeners to the
  'changed' signal

check_update(checker)
  If mappable has changed since the last check, return True; else return False

contains(mouseevent)
  Test whether the mouse event occurred in the collection.
  Returns True | False, dict(ind=itemlist), where every item in itemlist contains the event.

static convert_mesh_to_paths(tri)
  Converts a given mesh into a sequence of matplotlib.path.Path objects for easier rendering
  by backends that do not directly support meshes.
  This function is primarily of use to backend implementers.

convert_xunits(x)
  For artists in an axes, if the xaxis has units support, convert x using xaxis unit type

convert_yunits(y)
  For artists in an axes, if the yaxis has units support, convert y using yaxis unit type

draw(artist, renderer, *args, **kwargs)

findobj(match=None, include_self=True)
  Find artist objects.
  Recursively find all Artist instances contained in self.
  match can be
    • None: return all objects contained in artist.
    • function with signature boolean = match(artist) used to filter matches
    • class instance: e.g., Line2D. Only return artists of class type.
  If include_self is True (default), include self in the list to be checked for a match.

format_cursor_data(data)
  Return cursor data string formatted.

get_agg_filter()
  return filter function to be used for agg filter

geta
ta()()  
  Return the alpha value used for blending - not supported on all backends

g
timated()()  
  Return the artist’s animated state

g
t()()  
  Return the array
get_axes()
    Return the Axes instance the artist resides in, or None.
    
    This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

get_children()
    Return a list of the child Artist's this :class:`Artist` contains.

get_clim()
    return the min, max of the color limits for image scaling

get_clip_box()
    Return artist clipbox

get_clip_on()
    Return whether artist uses clipping

get_clip_path()
    Return artist clip path

get_cmap()
    return the colormap

get_contains()
    Return the _contains test used by the artist, or None for default.

get_cursor_data(event)
    Get the cursor data for a given event.

get_dashes()

get_datalim(transData)

get_edgecolor()

get_edgecolors()

get_facecolor()

get_facecolors()

get_figure()
    Return the Figure instance the artist belongs to.

get_fill()
    return whether fill is set

get_gid()
    Returns the group id
get_hatch()
Return the current hatching pattern

get_label()
Get the label used for this artist in the legend.

get_linestyle()

get_linestyles()

get_linewidth()

get_linewidths()

get_offset_position()
Returns how offsets are applied for the collection. If offset_position is ‘screen’, the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is ‘data’, the offset is applied before the master transform, i.e., the offsets are in data coordinates.

get_offset_transform()

get_offsets()
Return the offsets for the collection.

get_path_effects()

get_paths()

get_picker()
Return the picker object used by this artist

get_pickradius()

get_rasterized()
return True if the artist is to be rasterized

get_sketch_params()
Returns the sketch parameters for the artist.

Returns sketch_params : tuple or None
A 3-tuple with the following elements:
• scale: The amplitude of the wiggle perpendicular to the source line.
• length: The length of the wiggle along the line.
• randomness: The scale factor by which the length is shrunken or expanded.
May return None if no sketch parameters were set.

**get_snap()**

Returns the snap setting which may be:
- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

**get_transform()**

Return the *Transform* instance used by this artist.

**get_transformed_clip_path_and_affine()**

Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

**get_transforms()**

**get_url()**

Returns the url

**get_urls()**

**get_visible()**

Return the artist’s visibility

**get_window_extent(renderer)**

**get_zorder()**

Return the Artist’s zorder.

**have_units()**

Return True if units are set on the x or y axes

**hitlist(event)**

List the children of the artist which contain the mouse event event.

**is_figure_set()**

Returns True if the artist is assigned to a *Figure*.

**is_transform_set()**

Returns True if Artist has a transform explicitly set.

**mouseover**

**pchanged()**

Fire an event when property changed, calling all of the registered callbacks.
**pick**(mouseevent)
Process pick event

each child artist will fire a pick event if mouseevent is over the artist and the artist has picker set

**pickable()**
Return True if Artist is pickable.

**properties()**
return a dictionary mapping property name -> value for all Artist props

**remove()**
Remove the artist from the figure if possible. The effect will not be visible until the figure is redrawn, e.g., with `matplotlib.axes.Axes.draw_idle()`. Call `matplotlib.axes.Axes.relim()` to update the axes limits if desired.

Note: `relin() will not see collections even if the collection was added to axes with autolim = True.

Note: there is no support for removing the artist’s legend entry.

**remove_callback**(oid)
Remove a callback based on its id.

See also:

```
add_callback() For adding callbacks
```

**set(**kwargs)**
A property batch setter. Pass kwargs to set properties.

**set_agg_filter**(filter_func)
set agg_filter fuction.

**set_alpha**(alpha)
Set the alpha tranparencies of the collection. alpha must be a float or None.

ACCEPTS: float or None

**set_animated**(b)
Set the artist’s animation state.

ACCEPTS: [True | False]

**set_antialiased**(aa)
Set the antialiasing state for rendering.

ACCEPTS: Boolean or sequence of booleans

**set_antialiaseds**(aa)
alias for set_antialiased

**set_array**(A)
Set the image array from numpy array A
**set_axes**
Set the *Axes* instance in which the artist resides, if any.

This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

ACCEPTS: an *Axes* instance

**set_clim**
set the norm limits for image scaling; if *vmin* is a length2 sequence, interpret it as (*vmin*, *vmax*) which is used to support setp

ACCEPTS: a length 2 sequence of floats

**set_clip_box**
Set the artist’s clip *Bbox*.

ACCEPTS: a *matplotlib.transforms.Bbox* instance

**set_clip_on**
Set whether artist uses clipping.

When False artists will be visible out side of the axes which can lead to unexpected results.

ACCEPTS: [True | False]

**set_clip_path**
Set the artist’s clip path, which may be:

- a *Patch* (or subclass) instance
- a *Path* instance, in which case an optional *Transform* instance may be provided, which will be applied to the path before using it for clipping.
- *None*, to remove the clipping path

For efficiency, if the path happens to be an axis-aligned rectangle, this method will set the clipping box to the corresponding rectangle and set the clipping path to *None*.

ACCEPTS: [ (Path, Transform) | Patch | None ]

**set_cmap**
set the colormap for luminance data

ACCEPTS: a colormap or registered colormap name

**set_color**
Set both the edgecolor and the facecolor.

ACCEPTS: matplotlib color arg or sequence of rgba tuples

See also:

**set_facecolor**, **set_edgecolor** For setting the edge or face color individually.

**set_contains**
Replace the contains test used by this artist. The new picker should be a callable function which determines whether the artist is hit by the mouse event:
hit, props = picker(artist, mouseevent)

If the mouse event is over the artist, return hit = True and props is a dictionary of properties you want returned with the contains test.

ACCEPTS: a callable function

set_dashes(ls)
alias for set_linestyle

set_edgecolor(c)
Set the edgecolor(s) of the collection. c can be a matplotlib color spec (all patches have same color), or a sequence of specs; if it is a sequence the patches will cycle through the sequence.

If c is ‘face’, the edge color will always be the same as the face color. If it is ‘none’, the patch boundary will not be drawn.

ACCEPTS: matplotlib color spec or sequence of specs

set_edgecolors(c)
alias for set_edgecolor

set_facecolor(c)
Set the facecolor(s) of the collection. c can be a matplotlib color spec (all patches have same color), or a sequence of specs; if it is a sequence the patches will cycle through the sequence.

If c is ‘none’, the patch will not be filled.

ACCEPTS: matplotlib color spec or sequence of specs

set_facecolors(c)
alias for set_facecolor

set_figure(fig)
Set the Figure instance the artist belongs to.

ACCEPTS: a matplotlib.figure.Figure instance

set_gid(gid)
Sets the (group) id for the artist

ACCEPTS: an id string

set_hatch(hatch)
Set the hatching pattern

hatch can be one of:

/ - diagonal hatching
\ - back diagonal
| - vertical
- - horizontal
+ - crossed
x - crossed diagonal
o - small circle
O - large circle
Letters can be combined, in which case all the specified hatchings are done. If same letter
repeats, it increases the density of hatching of that pattern.

Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Unlike other properties such as linewidth and colors, hatching can only be specified for the
collection as a whole, not separately for each member.

ACCEPTS: [ ‘.’ | ‘\’ | ‘|’ | ‘-.’ | ‘+’ | ‘x’ | ‘o’ | ‘O’ | ‘.’ | ‘*’ ]

set_label(s)
Set the label to s for auto legend.

ACCEPTS: string or anything printable with ‘%s’ conversion.

set_linestyle(ls)
Set the linestyle(s) for the collection.

<table>
<thead>
<tr>
<th>linestyle</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘-‘ or ’solid‘</td>
<td>solid line</td>
</tr>
<tr>
<td>‘--‘ or ’dashed‘</td>
<td>dashed line</td>
</tr>
<tr>
<td>‘-.‘ or ’dashdot‘</td>
<td>dash-dotted line</td>
</tr>
<tr>
<td>‘:‘ or ’dotted‘</td>
<td>dotted line</td>
</tr>
</tbody>
</table>

Alternatively a dash tuple of the following form can be provided:

(offset, onoffseq),

where onoffseq is an even length tuple of on and off ink in points.

ACCEPTS: ['solid' | ‘dashed’, ‘dashdot’, ‘dotted’] (offset, on-off-dash-seq) | ‘-‘ | ‘--‘ | ‘-.‘ | ‘:‘ | 'None' | ‘ ’ | ‘ ’]

Parameters ls : { ‘-‘, ‘–‘, ‘-.‘, ‘:‘} and more see description

The line style.

set_linestyles(ls)
alias for set_linestyle

set_linewidth(lw)
Set the linewidth(s) for the collection. lw can be a scalar or a sequence; if it is a sequence the
patches will cycle through the sequence

ACCEPTS: float or sequence of floats

set_linewidths(lw)
alias for set_linewidth

set_lw(lw)
alias for set_linewidth
**set_norm**(*norm*)

set the normalization instance

**set_offset_position**(*offset_position*)

Set how offsets are applied. If *offset_position* is ‘screen’ (default) the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If *offset_position* is ‘data’, the offset is applied before the master transform, i.e., the offsets are in data coordinates.

**set_offsets**(*offsets*)

Set the offsets for the collection. *offsets* can be a scalar or a sequence.

**set_path_effects**(*path_effects*)

set path effects, which should be a list of instances of matplotlib.path.combine.PathEffects or its derivatives.

**set_paths**()

**set_picker**(*picker*)

Set the epsilon for picking used by this artist

*picker* can be one of the following:

- *None*: picking is disabled for this artist (default)
- A boolean: if *True* then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist
- A float: if picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if it’s data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event
- A function: if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

to determine the hit test. if the mouse event is over the artist, return *hit=True* and props is a dictionary of properties you want added to the PickEvent attributes.

**set_pickradius**(*pr*)

**set_rasterized**(*rasterized*)

Force rasterized (bitmap) drawing in vector backend output.

Defaults to None, which implies the backend’s default behavior

**ACCEPTS**: [True | False | None]
**set_sketch_params** *(scale=None, length=None, randomness=None)*
Sets the sketch parameters.

**Parameters**

- **scale**: float, optional
  The amplitude of the wiggle perpendicular to the source line, in pixels. If `scale` is `None`, or not provided, no sketch filter will be provided.

- **length**: float, optional
  The length of the wiggle along the line, in pixels (default 128.0)

- **randomness**: float, optional
  The scale factor by which the length is shrunken or expanded (default 16.0)

**set_snap**(snap)
Sets the snap setting which may be:

- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

**set_transform**(t)
Set the Transform instance used by this artist.

ACCEPETS: Transform instance

**set_url**(url)
Sets the url for the artist

ACCEPETS: a url string

**set_urls**(urls)

**set_visible**(b)
Set the artist’s visiblity.

ACCEPETS: [True | False]

**set_zorder**(level)
Set the zorder for the artist. Artists with lower zorder values are drawn first.

ACCEPETS: any number

**stale**
If the artist is ‘stale’ and needs to be re-drawn for the output to match the internal state of the artist.

**sticky_edges**
x and y sticky edge lists.
When performing autoscaling, if a data limit coincides with a value in the corresponding sticky_edges list, then no margin will be added—the view limit “sticks” to the edge. A typical usecase is histograms, where one usually expects no margin on the bottom edge (0) of the histogram.

This attribute cannot be assigned to; however, the x and y lists can be modified in place as needed.

Examples

```python
>>> artist.sticky_edges.x[:] = (xmin, xmax)
>>> artist.sticky_edges.y[:] = (ymin, ymax)
```

to_rgba(x, alpha=None, bytes=False, norm=True)

Return a normalized rgba array corresponding to x.

In the normal case, x is a 1-D or 2-D sequence of scalars, and the corresponding ndarray of rgba values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already rgb or rgba, such as might have been read from an image file. If x is an ndarray with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an rgb or rgba array, and no mapping will be done. If the last dimension is 3, the alpha kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the alpha kwarg is ignored; it does not replace the pre-existing alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if bytes is False (default), the rgba array will be floats in the 0-1 range; if it is True, the returned rgba array will be uint8 in the 0 to 255 range.

If norm is False, no normalization of the input data is performed, and it is assumed to already be in the range (0-1).

Note: this method assumes the input is well-behaved; it does not check for anomalies such as x being a masked rgba array, or being an integer type other than uint8, or being a floating point rgba array with values outside the 0-1 range.

update(props)

Update the properties of this Artist from the dictionary prop.

update_from(other)

copy properties from other to self

update_scalarmappable()

If the scalar mappable array is not none, update colors from scalar data

zorder = 0
CHAPTER
TWENTYFIVE

COLORBAR

25.1 matplotlib.colorbar

Colorbar toolkit with two classes and a function:

- **ColorbarBase** the base class with full colorbar drawing functionality. It can be used as-is to make a colorbar for a given colormap; a mappable object (e.g., image) is not needed.
- **Colorbar** the derived class for use with images or contour plots.
- **make_axes()** a function for resizing an axes and adding a second axes suitable for a colorbar

The `colorbar()` method uses `make_axes()` and `Colorbar`; the `colorbar()` function is a thin wrapper over `colorbar()`.

```python
class matplotlib.colorbar.Colorbar(ax, mappable, **kw)
```

Bases: `matplotlib.colorbar.ColorbarBase`

This class connects a `ColorbarBase` to a `ScalarMappable` such as a `AxesImage` generated via `imshow()`.

It is not intended to be instantiated directly; instead, use `colorbar()` or `colorbar()` to make your colorbar.

- **add_lines(CS, erase=True)**
  Add the lines from a non-filled `ContourSet` to the colorbar.
  Set `erase` to False if these lines should be added to any pre-existing lines.

- **on_mappable_changed(mappable)**
  Updates this colorbar to match the mappable’s properties.
  Typically this is automatically registered as an event handler by `colorbar_factory()` and should not be called manually.

- **remove()**
  Remove this colorbar from the figure. If the colorbar was created with `use_gridspec=True` then restore the gridspec to its previous value.

- **update_bruteforce(mappable)**
  Destroy and rebuild the colorbar. This is intended to become obsolete, and will probably be
deprecated and then removed. It is not called when the pyplot.colorbar function or the Figure.colorbar method are used to create the colorbar.

**update_normal** *(mappable)*
update solid, lines, etc. Unlike update_bruteforce, it does not clear the axes. This is meant to be called when the image or contour plot to which this colorbar belongs is changed.

```python
class matplotlib.colorbar.ColorbarBase(ax, cmap=None, norm=None, alpha=None, values=None, boundaries=None, orientation='vertical', ticklocation='auto', extend='neither', spacing='uniform', ticks=None, format=None, drawedges=False, filled=True, extendfrac=None, extendrect=False, label='')
```

Bases: `matplotlib.cm.ScalarMappable`

Draw a colorbar in an existing axes.

This is a base class for the `Colorbar` class, which is the basis for the `colorbar()` function and the `colorbar()` method, which are the usual ways of creating a colorbar.

It is also useful by itself for showing a colormap. If the `cmap` kwarg is given but `boundaries` and `values` are left as `None`, then the colormap will be displayed on a 0-1 scale. To show the under- and over-value colors, specify the `norm` as:

```python
colors.Normalize(clip=False)
```

To show the colors versus index instead of on the 0-1 scale, use:

```python
norm=colors.NoNorm.
```

Useful attributes:

- `ax` the Axes instance in which the colorbar is drawn
- `lines` a list of LineCollection if lines were drawn, otherwise an empty list
- `dividers` a LineCollection if `drawedges` is True, otherwise `None`

Useful public methods are `set_label()` and `add_lines()`.

```python
add_lines(levels, colors, linewights, erase=True)
```

Draw lines on the colorbar.

- `colors` and `linewights` must be scalars or sequences the same length as `levels`.
- Set `erase` to `False` to add lines without first removing any previously added lines.

```python
ax = None
```

The axes that this colorbar lives in.

```python
config_axis()
```

```python
draw_all()
```

Calculate any free parameters based on the current `cmap` and `norm`, and do all the drawing.
n_rasterize = 50

remove()
Remove this colorbar from the figure

set_alpha(alpha)

set_label(label, **kw)
Label the long axis of the colorbar

set_ticklabels(ticklabels, update_ticks=True)
set tick labels. Tick labels are updated immediately unless update_ticks is False. To manually update the ticks, call update_ticks method explicitly.

set_ticks(ticks, update_ticks=True)
set tick locations. Tick locations are updated immediately unless update_ticks is False. To manually update the ticks, call update_ticks method explicitly.

update_ticks()
Force the update of the ticks and ticklabels. This must be called whenever the tick locator and/or tick formatter changes.

class matplotlib.colorbar.ColorbarPatch(ax, mappable, **kw)
    Bases: matplotlib.colorbar.Colorbar

A Colorbar which is created using Patch rather than the default pcolor().

It uses a list of Patch instances instead of a PatchCollection because the latter does not allow the hatch pattern to vary among the members of the collection.

matplotlib.colorbar.colorbar_factory(cax, mappable, **kwargs)
Creates a colorbar on the given axes for the given mappable.

Typically, for automatic colorbar placement given only a mappable use colorbar().

matplotlib.colorbar.make_axes(parents, location=None, orientation=None, fraction=0.15, shrink=1.0, aspect=20, **kw)
Resize and reposition parent axes, and return a child axes suitable for a colorbar.

Keyword arguments may include the following (with defaults):

**location** [[None]['left']['right']['top']['bottom']] The position, relative to parents, where the colorbar axes should be created. If None, the value will either come from the given orientation, else it will default to ‘right’.

**orientation** [[None]['vertical']['horizontal']] The orientation of the colorbar. Typically, this keyword shouldn’t be used, as it can be derived from the location keyword.
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>orientation</td>
<td>vertical or horizontal</td>
</tr>
<tr>
<td>fraction</td>
<td>0.15; fraction of original axes to use for colorbar</td>
</tr>
<tr>
<td>pad</td>
<td>0.05 if vertical, 0.15 if horizontal; fraction of original axes between colorbar and new image axes</td>
</tr>
<tr>
<td>shrink</td>
<td>1.0; fraction by which to shrink the colorbar</td>
</tr>
<tr>
<td>aspect</td>
<td>20; ratio of long to short dimensions</td>
</tr>
<tr>
<td>anchor</td>
<td>(0.0, 0.5) if vertical; (0.5, 1.0) if horizontal; the anchor point of the colorbar axes</td>
</tr>
<tr>
<td>panchor</td>
<td>(1.0, 0.5) if vertical; (0.5, 0.0) if horizontal; the anchor point of the colorbar parent axes. If False, the parent axes’ anchor will be unchanged</td>
</tr>
</tbody>
</table>

Returns (cax, kw), the child axes and the reduced kw dictionary to be passed when creating the colorbar instance.

```
matplotlib.colorbar.make_axes_gridspec(parent, **kw)
```

Resize and reposition a parent axes, and return a child axes suitable for a colorbar. This function is similar to make_axes. Primary differences are

- `make_axes_gridspec` only handles the orientation keyword and cannot handle the “location” keyword.
- `make_axes_gridspec` should only be used with a subplot parent.
- `make_axes` creates an instance of Axes. `make_axes_gridspec` creates an instance of Subplot.
- `make_axes` updates the position of the parent. `make_axes_gridspec` replaces the grid_spec attribute of the parent with a new one.

While this function is meant to be compatible with `make_axes`, there could be some minor differences.

Keyword arguments may include the following (with defaults):

- `orientation` ‘vertical’ or ‘horizontal’
All but the first of these are stripped from the input kw set.

Returns (cax, kw), the child axes and the reduced kw dictionary to be passed when creating the colorbar instance.
For a visual representation of the matplotlib colormaps, see the “Color” section in the gallery.

### 26.1 matplotlib.colors

A module for converting numbers or color arguments to RGB or RGBA

RGB and RGBA are sequences of, respectively, 3 or 4 floats in the range 0-1.

This module includes functions and classes for color specification conversions, and for mapping numbers to colors in a 1-D array of colors called a colormap. Colormapping typically involves two steps: a data array is first mapped onto the range 0-1 using an instance of Normalize or of a subclass; then this number in the 0-1 range is mapped to a color using an instance of a subclass of Colormap. Two are provided here: LinearSegmentedColormap, which is used to generate all the built-in colormap instances, but is also useful for making custom colormaps, and ListedColormap, which is used for generating a custom colormap from a list of color specifications.

The module also provides functions for checking whether an object can be interpreted as a color (is_color_like()), for converting such an object to an RGBA tuple (to_rgb()) or to an HTML-like hex string in the #rrggbb format (to_hex()), and a sequence of colors to an (n, 4) RGBA array (to_rgba_array()). Caching is used for efficiency.

Commands which take color arguments can use several formats to specify the colors. For the basic built-in colors, you can use a single letter

- b: blue
- g: green
- r: red
- c: cyan
- m: magenta
- y: yellow
- k: black
- w: white
To use the colors that are part of the active color cycle in the current style, use C followed by a digit. For example:

- C0: The first color in the cycle
- C1: The second color in the cycle

Gray shades can be given as a string encoding a float in the 0-1 range, e.g.:

```
color = '0.75'
```

For a greater range of colors, you have two options. You can specify the color using an html hex string, as in:

```
color = '#eeefff'
```

(possibly specifying an alpha value as well), or you can pass an (r, g, b) or (r, g, b, a) tuple, where each of r, g, b and a are in the range [0,1].

Finally, legal html names for colors, like ‘red’, ‘burlywood’ and ‘chartreuse’ are supported.

```
class matplotlib.colors.BoundaryNorm(boundaries, ncolors, clip=False)
    Bases: matplotlib.colors.Normalize

    Generate a colormap index based on discrete intervals.

    Unlike Normalize or LogNorm, BoundaryNorm maps values to integers instead of to the interval 0-1.

    Mapping to the 0-1 interval could have been done via piece-wise linear interpolation, but using integers seems simpler, and reduces the number of conversions back and forth between integer and floating point.

    boundaries a monotonically increasing sequence

    ncolors number of colors in the colormap to be used

    If:

    ```
    b[i] <= v < b[i+1]
    ```

    then v is mapped to color j; as i varies from 0 to len(boundaries)-2, j goes from 0 to ncolors-1.

    Out-of-range values are mapped to -1 if low and ncolors if high; these are converted to valid indices by Colormap.__call__(). If clip == True, out-of-range values are mapped to 0 if low and ncolors-1 if high.

    inverse(value)
```

```
class matplotlib.colors.Colormap(name, N=256)
    Bases: object

    Baseclass for all scalar to RGBA mappings.

    Typically Colormap instances are used to convert data values (floats) from the interval [0, 1] to the RGBA color that the respective Colormap represents. For scaling of data into the [0, 1] interval
```
see `matplotlib.colors.Normalize`. It is worth noting that `matplotlib.cm.ScalarMappable` subclasses make heavy use of this `data->normalize->map-to-color` processing chain.

**Parameters**

- **name**: str
  - The name of the colormap.

- **N**: int
  - The number of rgb quantization levels.

- **colorbar_extend**: None
  - When this colormap exists on a scalar mappable and colorbar_extend is not False, colorbar creation will pick up colorbar_extend as the default value for the extend keyword in the `matplotlib.colorbar.Colorbar` constructor.

- **is_gray**

- **set_bad**(color='k', alpha=None)
  - Set color to be used for masked values.

- **set_over**(color='k', alpha=None)
  - Set color to be used for high out-of-range values. Requires norm.clip = False

- **set_under**(color='k', alpha=None)
  - Set color to be used for low out-of-range values. Requires norm.clip = False

**class matplotlib.colors.LightSource(azdeg=315, altdeg=45, hsv_min_val=0, hsv_max_val=1, hsv_min_sat=1, hsv_max_sat=0)**

Bases: object

Create a light source coming from the specified azimuth and elevation. Angles are in degrees, with the azimuth measured clockwise from north and elevation up from the zero plane of the surface.

The `shade()` is used to produce “shaded” rgb values for a data array. `shade_rgb()` can be used to combine an rgb image with The `shade_rgb()` The `hillshade()` produces an illumination map of a surface.

Specify the azimuth (measured clockwise from south) and altitude (measured up from the plane of the surface) of the light source in degrees.

**Parameters**

- **azdeg** : number, optional
  - The azimuth (0-360, degrees clockwise from North) of the light source. Defaults to 315 degrees (from the northwest).

- **altdeg** : number, optional
  - The altitude (0-90, degrees up from horizontal) of the light source. Defaults to 45 degrees from horizontal.
Notes

For backwards compatibility, the parameters `hsv_min_val`, `hsv_max_val`, `hsv_min_sat`, and `hsv_max_sat` may be supplied at initialization as well. However, these parameters will only be used if “blend_mode=’hsv’” is passed into `shade()` or `shade_rgb()`. See the documentation for `blend_hsv()` for more details.

```python
blend_hsv(rgb, intensity, hsv_max_sat=None, hsv_max_val=None, hsv_min_val=None, hsv_min_sat=None)
```

Take the input data array, convert to HSV values in the given colormap, then adjust those color values to give the impression of a shaded relief map with a specified light source. RGBA values are returned, which can then be used to plot the shaded image with imshow.

The color of the resulting image will be darkened by moving the (s,v) values (in hsv colorspace) toward (hsv_min_sat, hsv_min_val) in the shaded regions, or lightened by sliding (s,v) toward (hsv_max_sat hsv_max_val) in regions that are illuminated. The default extremes are chose so that completely shaded points are nearly black (s = 1, v = 0) and completely illuminated points are nearly white (s = 0, v = 1).

**Parameters**

- **rgb**: ndarray
  An MxNx3 RGB array of floats ranging from 0 to 1 (color image).

- **intensity**: ndarray
  An MxNx1 array of floats ranging from 0 to 1 (grayscale image).

- **hsv_max_sat**: number, optional
  The maximum saturation value that the `intensity` map can shift the output image to. Defaults to 1.

- **hsv_min_sat**: number, optional
  The minimum saturation value that the `intensity` map can shift the output image to. Defaults to 0.

- **hsv_max_val**: number, optional
  The maximum value (“v” in “hsv”) that the `intensity` map can shift the output image to. Defaults to 1.

- **hsv_min_val**: number, optional
  The minimum value (“v” in “hsv”) that the `intensity` map can shift the output image to. Defaults to 0.

**Returns**

- **rgb**: ndarray
  An MxNx3 RGB array representing the combined images.

```python
blend_overlay(rgb, intensity)
```

Combines an rgb image with an intensity map using “overlay” blending.

**Parameters**

- **rgb**: ndarray
  An MxNx3 RGB array of floats ranging from 0 to 1 (color image).
intensity : ndarray
    An MxNx1 array of floats ranging from 0 to 1 (grayscale image).

Returns rgb : ndarray
    An MxNx3 RGB array representing the combined images.

blend_soft_light(rgb, intensity)
    Combines an rgb image with an intensity map using “soft light” blending. Uses the “pegtop” formula.

    Parameters rgb : ndarray
        An MxNx3 RGB array of floats ranging from 0 to 1 (color image).

    intensity : ndarray
        An MxNx1 array of floats ranging from 0 to 1 (grayscale image).

    Returns rgb : ndarray
        An MxNx3 RGB array representing the combined images.

hillshade(elevation, vert_exag=1, dx=1, dy=1, fraction=1.0)
    Calculates the illumination intensity for a surface using the defined azimuth and elevation for the light source.

    Imagine an artificial sun placed at infinity in some azimuth and elevation position illuminating our surface. The parts of the surface that slope toward the sun should brighten while those sides facing away should become darker.

    Parameters elevation : array-like
        A 2d array (or equivalent) of the height values used to generate an illumination map.

    vert_exag : number, optional
        The amount to exaggerate the elevation values by when calculating illumination. This can be used either to correct for differences in units between the x-y coordinate system and the elevation coordinate system (e.g. decimal degrees vs meters) or to exaggerate or de-emphasize topographic effects.

    dx : number, optional
        The x-spacing (columns) of the input elevation grid.

    dy : number, optional
        The y-spacing (rows) of the input elevation grid.

    fraction : number, optional
        Increases or decreases the contrast of the hillshade. Values greater than one will cause intermediate values to move closer to full illumination or shadow (and clipping any values that move beyond 0 or 1). Note that this is not visually or mathematically the same as vertical exaggeration.
Returns
——

**intensity** : ndarray

A 2d array of illumination values between 0-1, where 0 is completely in shadow and 1 is completely illuminated.

```python
shade(data, cmap, norm=None, blend_mode='overlay', vmin=None, vmax=None, vert_exag=1, dx=1, dy=1, fraction=1, **kwargs)
```

Combine colormapped data values with an illumination intensity map (a.k.a. “hillshade”) of the values.

**Parameters**

- **data** : array-like
  A 2d array (or equivalent) of the height values used to generate a shaded map.

- **cmap** : Colormap instance
  The colormap used to color the data array. Note that this must be a Colormap instance. For example, rather than passing in cmap='gist_earth', use cmap=plt.get_cmap('gist_earth') instead.

- **norm** : Normalize instance, optional
  The normalization used to scale values before colormapping. If None, the input will be linearly scaled between its min and max.

- **blend_mode** : {'hsv', 'overlay', 'soft'} or callable, optional
  The type of blending used to combine the colormapped data values with the illumination intensity. Default is “overlay”. Note that for most topographic surfaces, “overlay” or “soft” appear more visually realistic. If a user-defined function is supplied, it is expected to combine an MxNx3 RGB array of floats (ranging 0 to 1) with an MxNx1 hillshade array (also 0 to 1). (Call signature func(rgb, illum, **kwargs)) Additional kwargs supplied to this function will be passed on to the blend_mode function.

- **vmin** : scalar or None, optional
  The minimum value used in colormapping data. If None the minimum value in data is used. If norm is specified, then this argument will be ignored.

- **vmax** : scalar or None, optional
  The maximum value used in colormapping data. If None the maximum value in data is used. If norm is specified, then this argument will be ignored.

- **vert_exag** : number, optional
  The amount to exaggerate the elevation values by when calculating illumination. This can be used either to correct for differences in units between the x-y coordinate system and the elevation coordinate system (e.g. decimal degrees vs meters) or to exaggerate or de-emphasize topography.

- **dx** : number, optional
The x-spacing (columns) of the input elevation grid.

dy : number, optional

The y-spacing (rows) of the input elevation grid.

fraction : number, optional

Increases or decreases the contrast of the hillshade. Values greater than one will cause intermediate values to move closer to full illumination or shadow (and clipping any values that move beyond 0 or 1). Note that this is not visually or mathematically the same as vertical exaggeration.

Additional kwargs are passed on to the *blend_mode* function.

Returns rgba : ndarray

An MxNx4 array of floats ranging between 0-1.

shade_rgb(rgb, elevation, fraction=1.0, blend_mode='hsv', vert_exag=1, dx=1, dy=1, **kwargs)

Take the input RGB array (ny*nx*3) adjust their color values to given the impression of a shaded relief map with a specified light source using the elevation (ny*nx). A new RGB array ((ny*nx*3)) is returned.

Parameters rgb : array-like

An MxNx3 RGB array, assumed to be in the range of 0 to 1.

elevation : array-like

A 2d array (or equivalent) of the height values used to generate a shaded map.

fraction : number

Increases or decreases the contrast of the hillshade. Values greater than one will cause intermediate values to move closer to full illumination or shadow (and clipping any values that move beyond 0 or 1). Note that this is not visually or mathematically the same as vertical exaggeration.

blend_mode : {'hsv', 'overlay', 'soft'} or callable, optional

The type of blending used to combine the colormapped data values with the illumination intensity. For backwards compatibility, this defaults to “hsv”. Note that for most topographic surfaces, “overlay” or “soft” appear more visually realistic. If a user-defined function is supplied, it is expected to combine an MxNx3 RGB array of floats (ranging 0 to 1) with an MxNx1 hillshade array (also 0 to 1). (Call signature func(rgb, illum, **kwargs)) Additional kwargs supplied to this function will be passed on to the blend_mode function.

vert_exag : number, optional

The amount to exaggerate the elevation values by when calculating illumination. This can be used either to correct for differences in units between the x-y coordinate system and the elevation coordinate system (e.g. decimal degrees vs meters) or to exaggerate or de-emphasize topography.
dx : number, optional
The x-spacing (columns) of the input elevation grid.

dy : number, optional
The y-spacing (rows) of the input elevation grid.

Additional kwargs are passed on to the *blend_mode* function.

Returns shaded_rgb : ndarray
An MxNx3 array of floats ranging between 0-1.

class matplotlib.colors.LinearSegmentedColormap(name, segmentdata, N=256, gamma=1.0)
Bases: matplotlib.colors.Colormap

Colormap objects based on lookup tables using linear segments.

The lookup table is generated using linear interpolation for each primary color, with the 0-1 domain divided into any number of segments.

Create color map from linear mapping segments

segmentdata argument is a dictionary with a red, green and blue entries. Each entry should be a list of x, y0, y1 tuples, forming rows in a table. Entries for alpha are optional.

Example: suppose you want red to increase from 0 to 1 over the bottom half, green to do the same over the middle half, and blue over the top half. Then you would use:

cdict = {'red': [(0.0, 0.0, 0.0),
                 (0.5, 1.0, 1.0),
                 (1.0, 1.0, 1.0)],

green': [(0.0, 0.0, 0.0),
          (0.25, 0.0, 0.0),
          (0.75, 1.0, 1.0),
          (1.0, 1.0, 1.0)],

'blue': [(0.0, 0.0, 0.0),
         (0.5, 0.0, 0.0),
         (1.0, 1.0, 1.0)]}

Each row in the table for a given color is a sequence of x, y0, y1 tuples. In each sequence, x must increase monotonically from 0 to 1. For any input value z falling between x[i] and x[i+1], the output value of a given color will be linearly interpolated between y1[i] and y0[i+1]:

row i: x y0 y1
/ / / row i+1: x y0 y1

Hence y0 in the first row and y1 in the last row are never used.

See also:
**LinearSegmentedColormap.from_list()** Static method; factory function for generating a smoothly-varying LinearSegmentedColormap.

**makeMappingArray()** For information about making a mapping array.

**static from_list(name, colors, N=256, gamma=1.0)**
Make a linear segmented colormap with name from a sequence of colors which evenly transitions from colors[0] at val=0 to colors[-1] at val=1. N is the number of rgb quantization levels. Alternatively, a list of (value, color) tuples can be given to divide the range unevenly.

**set_gamma(gamma)**
Set a new gamma value and regenerate color map.

**class matplotlib.colors.ListedColormap(colors, name='from_list', N=None)**
Colormap object generated from a list of colors.

This may be most useful when indexing directly into a colormap, but it can also be used to generate special colormaps for ordinary mapping.

Make a colormap from a list of colors.

**colors** a list of matplotlib color specifications, or an equivalent Nx3 or Nx4 floating point array (N rgb or rgba values)

**name** a string to identify the colormap

**N** the number of entries in the map. The default is None, in which case there is one colormap entry for each element in the list of colors. If:

\[ N < \text{len(colors)} \]

the list will be truncated at N. If:

\[ N > \text{len(colors)} \]

the list will be extended by repetition.

**class matplotlib.colors.LogNorm(vmin=None, vmax=None, clip=False)**
Normalize a given value to the 0-1 range on a log scale

If vmin or vmax is not given, they are initialized from the minimum and maximum value respectively of the first input processed. That is, __call__(A) calls autoscale_None(A). If clip is True and the given value falls outside the range, the returned value will be 0 or 1, whichever is closer. Returns 0 if:

\[ \text{vmin} == \text{vmax} \]

Works with scalars or arrays, including masked arrays. If clip is True, masked values are set to 1; otherwise they remain masked. Clipping silently defeats the purpose of setting the over, under, and masked colors in the colormap, so it is likely to lead to surprises; therefore the default is clip = False.
autoscale(A)
    Set vmin, vmax to min, max of A.

autoscale_None(A)
    autoscale only None-valued vmin or vmax

inverse(value)

class matplotlib.colors.NoNorm(vmin=None, vmax=None, clip=False)
    Bases: matplotlib.colors.Normalize

    Dummy replacement for Normalize, for the case where we want to use indices directly in a
    ScalarMappable .

    If vmin or vmax is not given, they are initialized from the minimum and maximum value respectively
    of the first input processed. That is, __call__(A) calls autoscale_None(A). If clip is True and the given
    value falls outside the range, the returned value will be 0 or 1, whichever is closer. Returns 0 if:

    \[
    \text{vmin} = \text{vmax}
    \]

    Works with scalars or arrays, including masked arrays. If clip is True, masked values are set to 1;
    otherwise they remain masked. Clipping silently defeats the purpose of setting the over, under, and
    masked colors in the colormap, so it is likely to lead to surprises; therefore the default is clip = False.

inverse(value)

class matplotlib.colors.Normalize(vmin=None, vmax=None, clip=False)
    Bases: object

    A class which, when called, can normalize data into the \([0.0, 1.0]\) interval.

    If vmin or vmax is not given, they are initialized from the minimum and maximum value respectively
    of the first input processed. That is, __call__(A) calls autoscale_None(A). If clip is True and the given
    value falls outside the range, the returned value will be 0 or 1, whichever is closer. Returns 0 if:

    \[
    \text{vmin} = \text{vmax}
    \]

    Works with scalars or arrays, including masked arrays. If clip is True, masked values are set to 1;
    otherwise they remain masked. Clipping silently defeats the purpose of setting the over, under, and
    masked colors in the colormap, so it is likely to lead to surprises; therefore the default is clip = False.

autoscale(A)
    Set vmin, vmax to min, max of A.

autoscale_None(A)
    autoscale only None-valued vmin or vmax

inverse(value)
static **process_value**(*value*)

Homogenize the input *value* for easy and efficient normalization.

*value* can be a scalar or sequence.

Returns *result*, *is_scalar*, where *result* is a masked array matching *value*. Float dtypes are preserved; integer types with two bytes or smaller are converted to np.float32, and larger types are converted to np.float. Preserving float32 when possible, and using in-place operations, can greatly improve speed for large arrays.

Experimental; we may want to add an option to force the use of float32.

**scaled()**

return true if vmin and vmax set

class **matplotlib.colors.PowerNorm**(*gamma*, vmin=None, vmax=None, clip=False)

Bases: **matplotlib.colors.Normalize**

Normalize a given value to the \([0, 1]\) interval with a power-law scaling. This will clip any negative data points to 0.

**autoscale**(*A*)

Set *vmin*, *vmax* to min, max of *A*.

**autoscale_None**(*A*)

autoscale only None-valued vmin or vmax

**inverse**(*value*)

class **matplotlib.colors.SymLogNorm**(*linthresh*, *linscale=1.0*, *vmin=None*, *vmax=None*, *clip=False*)

Bases: **matplotlib.colors.Normalize**

The symmetrical logarithmic scale is logarithmic in both the positive and negative directions from the origin.

Since the values close to zero tend toward infinity, there is a need to have a range around zero that is linear. The parameter *linthresh* allows the user to specify the size of this range \((-\text{linthresh}, \text{linthresh})\).

*linthresh*: The range within which the plot is linear (to avoid having the plot go to infinity around zero).

*linscale*: This allows the linear range \((-\text{linthresh} \text{ to } \text{linthresh})\) to be stretched relative to the logarithmic range. Its value is the number of decades to use for each half of the linear range. For example, when *linscale* \(== 1.0\) (the default), the space used for the positive and negative halves of the linear range will be equal to one decade in the logarithmic range. Defaults to 1.

**autoscale**(*A*)

Set *vmin*, *vmax* to min, max of *A*.

**autoscale_None**(*A*)

autoscale only None-valued vmin or vmax

**inverse**(*value*)
matplotlib.colors.from_levels_and_colors(levels, colors, extend='neither')

A helper routine to generate a cmap and a norm instance which behave similar to contourf’s levels and colors arguments.

**Parameters**

- **levels**: sequence of numbers
  The quantization levels used to construct the `BoundaryNorm`. Values \( v \) are quantized to level \( i \) if \( \text{lev}[i] \leq v < \text{lev}[i+1] \).

- **colors**: sequence of colors
  The fill color to use for each level. If `extend` is “neither” there must be \( n_{\text{level}} - 1 \) colors. For an `extend` of “min” or “max” add one extra color, and for an `extend` of “both” add two colors.

- **extend**: {'neither', 'min', 'max', 'both'}, optional
  The behaviour when a value falls out of range of the given levels. See `contourf()` for details.

**Returns**

- (cmap, norm): tuple containing a `Colormap` and a `Normalize` instance

matplotlib.colors.get_named_colors_mapping()
Return the global mapping of names to named colors.

matplotlib.colors.hsv_to_rgb(hsv)
convert hsv values in a numpy array to rgb values all values assumed to be in range [0, 1]

**Parameters**

- **hsv**: (..., 3) array-like
  All values assumed to be in range [0, 1]

**Returns**

- **rgb**: (..., 3) ndarray
  Colors converted to RGB values in range [0, 1]

matplotlib.colors.is_color_like(c)
Return whether \( c \) can be interpreted as an RGB(A) color.

matplotlib.colors.makeMappingArray(N, data, gamma=1.0)
Create an \( N \) -element 1-d lookup table

\( data \) represented by a list of \( x,y_0,y_1 \) mapping correspondences. Each element in this list represents how a value between 0 and 1 (inclusive) represented by \( x \) is mapped to a corresponding value between 0 and 1 (inclusive). The two values of \( y \) are to allow for discontinuous mapping functions (say as might be found in a sawtooth) where \( y_0 \) represents the value of \( y \) for values of \( x \leq \) to that given, and \( y_1 \) is the value to be used for \( x > \) than that given). The list must start with \( x=0 \), end with \( x=1 \), and all values of \( x \) must be in increasing order. Values between the given mapping points are determined by simple linear interpolation.

Alternatively, \( data \) can be a function mapping values between \( 0 - 1 \) to \( 0 - 1 \).

The function returns an array “result” where \( \text{result}[x*(N-1)] \) gives the closest value for values of \( x \) between 0 and 1.

matplotlib.colors.rgb_to_hsv(arr)
convert float rgb values (in the range [0, 1]), in a numpy array to hsv values.
Parameters

: arr : (..., 3) array-like

All values must be in the range [0, 1]

Returns

: hsv : (..., 3) ndarray

Colors converted to hsv values in range [0, 1]

matplotlib.colors.to_hex(c, keep_alpha=False)

Convert c to a hex color.

Uses the #rrggb format if keep_alpha is False (the default), #rrggbbaa otherwise.

matplotlib.colors.to_rgb(c)

Convert c to an RGB color, silently dropping the alpha channel.

matplotlib.colors.to_rgba(c, alpha=None)

Convert c to an RGBA color.

If alpha is not None, it forces the alpha value, except if c is “none” (case-insensitive), which always maps to (0, 0, 0, 0).

matplotlib.colors.to_rgba_array(c, alpha=None)

Convert c to a (n, 4) array of RGBA colors.

If alpha is not None, it forces the alpha value. If c is “none” (case-insensitive) or an empty list, an empty array is returned.
27.1 `matplotlib.container`

```python
class matplotlib.container.BarContainer(patches, errorbar=None, **kwargs)
    Bases: matplotlib.container.Container

class matplotlib.container.Container(kl, label=None)
    Bases: tuple
    Base class for containers.

    add_callback(func)
        Adds a callback function that will be called whenever one of the Artist's properties changes.
        Returns an id that is useful for removing the callback with `remove_callback()` later.

get_children()

get_label()
        Get the label used for this artist in the legend.

pchanged()
        Fire an event when property changed, calling all of the registered callbacks.

remove()

remove_callback(oid)
        Remove a callback based on its id.

    See also:

        `add_callback()` For adding callbacks

set_label(s)
        Set the label to s for auto legend.
        ACCEPTS: string or anything printable with ‘%s’ conversion.

set_remove_method(f)
```
class matplotlib.container.ErrorbarContainer(lines, has_xerr=False, has_yerr=False, **kwargs)
    Bases: matplotlib.container.Container

class matplotlib.container.StemContainer(markerline_stemlines_baseline, **kwargs)
    Bases: matplotlib.container.Container
28.1 matplotlib.dates

Matplotlib provides sophisticated date plotting capabilities, standing on the shoulders of Python datetime, the add-on modules pytz and dateutil. datetime objects are converted to floating point numbers which represent time in days since 0001-01-01 UTC, plus 1. For example, 0001-01-01, 06:00 is 1.25, not 0.25. The helper functions date2num(), num2date() and drange() are used to facilitate easy conversion to and from datetime and numeric ranges.

Note: Like Python’s datetime, mpl uses the Gregorian calendar for all conversions between dates and floating point numbers. This practice is not universal, and calendar differences can cause confusing differences between what Python and mpl give as the number of days since 0001-01-01 and what other software and databases yield. For example, the US Naval Observatory uses a calendar that switches from Julian to Gregorian in October, 1582. Hence, using their calculator, the number of days between 0001-01-01 and 2006-04-01 is 732403, whereas using the Gregorian calendar via the datetime module we find:

```
In [31]: date(2006,4,1).toordinal() - date(1,1,1).toordinal()
Out[31]: 732401
```
A wide range of specific and general purpose date tick locators and formatters are provided in this module. See `matplotlib.ticker` for general information on tick locators and formatters. These are described below.

All the matplotlib date converters, tickers and formatters are timezone aware, and the default timezone is given by the timezone parameter in your `matplotlibrc` file. If you leave out a tz timezone instance, the default from your rc file will be assumed. If you want to use a custom time zone, pass a `pytz.timezone` instance to `num2date()`, `plot_date()`, and any custom date tickers or locators you create. See `pytz` for information on `pytz` and timezone handling.

The `dateutil` module provides additional code to handle date ticking, making it easy to place ticks on any kinds of dates. See examples below.

### 28.1.1 Date tickers

Most of the date tickers can locate single or multiple values. For example:

```python
# import constants for the days of the week
from matplotlib.dates import MO, TU, WE, TH, FR, SA, SU

# tick on mondays every week
loc = WeekdayLocator(byweekday=MO, tz=tz)

# tick on mondays and saturdays
loc = WeekdayLocator(byweekday=(MO, SA))
```

In addition, most of the constructors take an interval argument:

```python
# tick on mondays every second week
loc = WeekdayLocator(byweekday=MO, interval=2)
```

The `rrule` locator allows completely general date ticking:

```python
# tick every 5th easter
rule = rrulewrapper(YEARLY, byeaster=1, interval=5)
loc = RRuleLocator(rule)
```

Here are all the date tickers:

- `MinuteLocator`: locate minutes
- `HourLocator`: locate hours
- `DayLocator`: locate specified days of the month
- `WeekdayLocator`: locate days of the week, e.g., MO, TU
- `MonthLocator`: locate months, e.g., 7 for july
- `YearLocator`: locate years that are multiples of base
• **RRuleLocator**: locate using a `matplotlib.dates.rrulewrapper`. The `rrulewrapper` is a simple wrapper around a `dateutil.rrule` function which allows almost arbitrary date tick specifications. See rrule example.

• **AutoDateLocator**: On autoscale, this class picks the best `MultipleDateLocator` to set the view limits and the tick locations.

### 28.1.2 Date formatters

Here all the date formatters:

• **AutoDateFormatter**: attempts to figure out the best format to use. This is most useful when used with the `AutoDateLocator`.

• **DateFormatter**: use `strftime()` format strings

• **IndexDateFormatter**: date plots with implicit `x` indexing.

```python
def date2num(d)
    d is either a `datetime` instance or a sequence of datetimes.
    Return value is a floating point number (or sequence of floats) which gives the number of days (fraction part represents hours, minutes, seconds) since 0001-01-01 00:00:00 UTC, plus one. The addition of one here is a historical artifact. Also, note that the Gregorian calendar is assumed; this is not universal practice. For details, see the module docstring.

def num2date(x, tz=None)
    x is a float value which gives the number of days (fraction part represents hours, minutes, seconds) since 0001-01-01 00:00:00 UTC plus one. The addition of one here is a historical artifact. Also, note that the Gregorian calendar is assumed; this is not universal practice. For details, see the module docstring.
    If `x` is a sequence, a sequence of `datetime` objects will be returned.

def drange(dstart, dend, delta)
    Return a date range as float Gregorian ordinals. `dstart` and `dend` are `datetime` instances. `delta` is a `datetime.timedelta` instance.

def epoch2num(e)
    Convert an epoch or sequence of epochs to the new date format, that is days since 0001.

def num2epoch(d)
    Convert days since 0001 to epoch. `d` can be a number or sequence.

def mx2num(mxdates)
    Convert mx `datetime` instance (or sequence of mx instances) to the new date format.

class DateFormatter(fmt, tz=None)
    Bases: `matplotlib.ticker.Formatter`

    Tick location is seconds since the epoch. Use a `strftime()` format string.
```
Python only supports `datetime strftime()` formatting for years greater than 1900. Thanks to Andrew Dalke, Dalke Scientific Software who contributed the `strftime()` code below to include dates earlier than this year.

`fmt` is a `strftime()` format string; `tz` is the `tzinfo` instance.

illegal_s = re.compile('((^[^%])(%[^%]*%s)')

set_tzinfo(tz)

`strftime(dt, fmt=None)`
Refer to documentation for `datetime.strftime`.

`fmt` is a `strftime()` format string.

Warning: For years before 1900, depending upon the current locale it is possible that the year displayed with `%x` might be incorrect. For years before 100, `%y` and `%Y` will yield zero-padded strings.

`strftime_pre_1900(dt, fmt=None)`
Call `time.strftime` for years before 1900 by rolling forward a multiple of 28 years.

`fmt` is a `strftime()` format string.

Dalke: I hope I did this math right. Every 28 years the calendar repeats, except through century leap years excepting the 400 year leap years. But only if you’re using the Gregorian calendar.

class matplotlib.dates.IndexDateFormatter(t, fmt, tz=None)
Bases: matplotlib.ticker.Formatter
Use with `IndexLocator` to cycle format strings by index.

`t` is a sequence of dates (floating point days). `fmt` is a `strftime()` format string.

class matplotlib.dates.AutoDateFormatter(locator, tz=None, defaultfmt='%Y-%m-%d')
Bases: matplotlib.ticker.Formatter
This class attempts to figure out the best format to use. This is most useful when used with the `AutoDateLocator`.

The `AutoDateFormatter` has a scale dictionary that maps the scale of the tick (the distance in days between one major tick) and a format string. The default looks like this:

```python
self.scaled = {
    DAYS_PER_YEAR: rcParams['date.autoformat.year'],
    DAYS_PER_MONTH: rcParams['date.autoformat.month'],
    1.0: rcParams['date.autoformat.day'],
    1. / HOURS_PER_DAY: rcParams['date.autoformat.hour'],
    1. / (MINUTES_PER_DAY): rcParams['date.autoformat.minute'],
    1. / (SEC_PER_DAY): rcParams['date.autoformat.second'],
    1. / (MILLISECONDS_PER_DAY): rcParams['date.autoformat.millisecond'],
}
```
The algorithm picks the key in the dictionary that is >= the current scale and uses that format string. You can customize this dictionary by doing:

```python
>>> locator = AutoDateLocator()
>>> formatter = AutoDateFormatter(locator)
>>> formatter.scaled[1/(24.*60.)] = '%M:%S' # only show min and sec
```

A custom `FuncFormatter` can also be used. The following example shows how to use a custom format function to strip trailing zeros from decimal seconds and adds the date to the first ticklabel:

```python
>>> def my_format_function(x, pos=None):
...     x = matplotlib.dates.num2date(x)
...     if pos == 0:
...         fmt = '%D %H:%M:%S.%f'
...     else:
...         fmt = '%H:%M:%S.%f'
...     label = x.strftime(fmt)
...     label = label.rstrip('0')
...     label = label.rstrip('.
...     return label
>>> from matplotlib.ticker import FuncFormatter
>>> formatter.scaled[1/(24.*60.)] = FuncFormatter(my_format_function)
```

Autoformat the date labels. The default format is the one to use if none of the values in `self.scaled` are greater than the unit returned by `locator._get_unit()`.

```python
class matplotlib.dates.DateLocator(tz=None)
Bases: matplotlib.ticker.Locator

Determines the tick locations when plotting dates.
tz is a tzinfo instance.

datalim_to_dt()
    Convert axis data interval to datetime objects.

hms0d = {'byhour': 0, 'byminute': 0, 'bysecond': 0}

nonsingular(vmin, vmax)
    Given the proposed upper and lower extent, adjust the range if it is too close to being singular (i.e. a range of ~0).

set_tzinfo(tz)
    Set time zone info.

viewlim_to_dt()
    Converts the view interval to datetime objects.

class matplotlib.dates.RRuleLocator(o, tz=None)
Bases: matplotlib.dates.DateLocator

autoscale()
    Set the view limits to include the data range.
```
static get_unit_generic(freq)

tick_values(vmin, vmax)

class matplotlib.dates.AutoDateLocator(tz=None, minticks=5, maxticks=None, interval_multiples=False)

Bases: matplotlib.dates.DateLocator

On autoscale, this class picks the best DateLocator to set the view limits and the tick locations.

minticks is the minimum number of ticks desired, which is used to select the type of ticking (yearly, monthly, etc.).

maxticks is the maximum number of ticks desired, which controls any interval between ticks (ticking every other, every 3, etc.). For really fine-grained control, this can be a dictionary mapping individual rrule frequency constants (YEARLY, MONTHLY, etc.) to their own maximum number of ticks. This can be used to keep the number of ticks appropriate to the format chosen in AutoDateFormatter. Any frequency not specified in this dictionary is given a default value.

tz is a tzinfo instance.

interval_multiples is a boolean that indicates whether ticks should be chosen to be multiple of the interval. This will lock ticks to ‘nicer’ locations. For example, this will force the ticks to be at hours 0, 6, 12, 18 when hourly ticking is done at 6 hour intervals.

The AutoDateLocator has an interval dictionary that maps the frequency of the tick (a constant from dateutil.rrule) and a multiple allowed for that ticking. The default looks like this:

```python
self.interval = {
    YEARLY : [1, 2, 4, 5, 10, 20, 40, 50, 100, 200, 400, 500, 1000, 2000, 4000, 5000, 10000],
    MONTHLY : [1, 2, 3, 4, 6],
    DAILY : [1, 2, 3, 7, 14],
    HOURLY : [1, 2, 3, 4, 6, 12],
    MINUTELY: [1, 5, 10, 15, 30],
    SECONDLY: [1, 5, 10, 15, 30],
    MICROSECONDLY: [1, 2, 5, 10, 20, 50, 100, 200, 500, 1000, 2000, 5000, 10000, 20000, 50000, 100000, 1000000],
}
```

The interval is used to specify multiples that are appropriate for the frequency of ticking. For instance, every 7 days is sensible for daily ticks, but for minutes/seconds, 15 or 30 make sense. You can customize this dictionary by doing:

```python
locator = AutoDateLocator()
locator.interval[HOURLY] = [3] # only show every 3 hours
 autoscale()

Try to choose the view limits intelligently.
get_locator\((dmin, dmax)\)

   Pick the best locator based on a distance.

nonsingular\((vmin, vmax)\)

refresh()

   Refresh internal information based on current limits.

set_axis\((axis)\)

tick_values\((vmin, vmax)\)

class matplotlib.dates.YearLocator\((base=1, month=1, day=1, tz=None)\)

   Bases: matplotlib.dates.DateLocator

   Make ticks on a given day of each year that is a multiple of base.

   Examples:

   ```python
   # Tick every year on Jan 1st
   locator = YearLocator()
   # Tick every 5 years on July 4th
   locator = YearLocator(5, month=7, day=4)
   ```

   Mark years that are multiple of base on a given month and day (default jan 1).

   autoscale()

   Set the view limits to include the data range.

tick_values\((vmin, vmax)\)

class matplotlib.dates.MonthLocator\((bymonth=None, bymonthday=1, interval=1, tz=None)\)

   Bases: matplotlib.dates.RRuleLocator

   Make ticks on occurances of each month month, e.g., 1, 3, 12.

   Mark every month in bymonth; bymonth can be an int or sequence. Default is range(1,13), i.e. every month.

   interval is the interval between each iteration. For example, if interval=2, mark every second occurrence.

class matplotlib.dates.WeekdayLocator\((byweekday=1, interval=1, tz=None)\)

   Bases: matplotlib.dates.RRuleLocator

   Make ticks on occurances of each weekday.

   Mark every weekday in byweekday; byweekday can be a number or sequence.

   Elements of byweekday must be one of MO, TU, WE, TH, FR, SA, SU, the constants from dateutil.rrule, which have been imported into the matplotlib.dates namespace.
interval specifies the number of weeks to skip. For example, interval=2 plots every second week.

class matplotlib.dates.DayLocator(bymonthday=None, interval=1, tz=None)
   Bases: matplotlib.dates.RRuleLocator
   Make ticks on occurrences of each day of the month. For example, 1, 15, 30.
   Mark every day in bymonthday; bymonthday can be an int or sequence.
   Default is to tick every day of the month: bymonthday=range(1, 32)

class matplotlib.dates.HourLocator(byhour=None, interval=1, tz=None)
   Bases: matplotlib.dates.RRuleLocator
   Make ticks on occurrences of each hour.
   Mark every hour in byhour; byhour can be an int or sequence. Default is to tick every hour: byhour=range(24)
   interval is the interval between each iteration. For example, if interval=2, mark every second occurrence.

class matplotlib.dates.MinuteLocator(byminute=None, interval=1, tz=None)
   Bases: matplotlib.dates.RRuleLocator
   Make ticks on occurrences of each minute.
   Mark every minute in byminute; byminute can be an int or sequence. Default is to tick every minute: byminute=range(60)
   interval is the interval between each iteration. For example, if interval=2, mark every second occurrence.

class matplotlib.dates.SecondLocator(bysecond=None, interval=1, tz=None)
   Bases: matplotlib.dates.RRuleLocator
   Make ticks on occurrences of each second.
   Mark every second in bysecond; bysecond can be an int or sequence. Default is to tick every second: bysecond = range(60)
   interval is the interval between each iteration. For example, if interval=2, mark every second occurrence.

class matplotlib.dates.MicrosecondLocator(interval=1, tz=None)
   Bases: matplotlib.dates.DateLocator
   Make ticks on occurrences of each microsecond.
   interval is the interval between each iteration. For example, if interval=2, mark every second microsecond.

set_axis(axis)

set_data_interval(vmin, vmax)
set_view_interval(vmin, vmax)

tick_values(vmin, vmax)

class matplotlib.dates.rrule(freq, dtstart=None, interval=1, wkst=None, count=None, until=None, bysetpos=None, bymonth=None, bymonthday=None, byyearday=None, byeaster=None, byweekday=None, byweekno=None, byweekday=None, byhour=None, byminute=None, bysecond=None, cache=False)

Bases: dateutil.rrule.rrulebase

That’s the base of the rule operation. It accepts all the keywords defined in the RFC as its constructor parameters (except byday, which was renamed to byweekday) and more. The constructor prototype is:

rrule(freq)

Where freq must be one of YEARLY, MONTHLY, WEEKLY, DAILY, HOURLY, MINUTELY, or SECONDLY.

Note: Per RFC section 3.3.10, recurrence instances falling on invalid dates and times are ignored rather than coerced:

Recurrence rules may generate recurrence instances with an invalid date (e.g., February 30) or nonexistent local time (e.g., 1:30 AM on a day where the local time is moved forward by an hour at 1:00 AM). Such recurrence instances MUST be ignored and MUST NOT be counted as part of the recurrence set.

This can lead to possibly surprising behavior when, for example, the start date occurs at the end of the month:

```python
>>> from dateutil.rrule import rrule, MONTHLY
>>> from datetime import datetime

>>> start_date = datetime(2014, 12, 31)

>>> list(rrule(freq=MONTHLY, count=4, dtstart=start_date))
...
[datetime.datetime(2014, 12, 31, 0, 0),
 datetime.datetime(2015, 1, 31, 0, 0),
 datetime.datetime(2015, 3, 31, 0, 0),
 datetime.datetime(2015, 5, 31, 0, 0)]
```

Additionally, it supports the following keyword arguments:

Parameters

- **cache** – If given, it must be a boolean value specifying to enable or disable caching of results. If you will use the same rrule instance multiple times, enabling caching will improve the performance considerably.
- **dtstart** – The recurrence start. Besides being the base for the recurrence, missing parameters in the final recurrence instances will also be extracted from this date. If not given, datetime.now() will be used instead.

- **interval** – The interval between each freq iteration. For example, when using YEARLY, an interval of 2 means once every two years, but with HOURLY, it means once every two hours. The default interval is 1.

- **wkst** – The week start day. Must be one of the MO, TU, WE constants, or an integer, specifying the first day of the week. This will affect recurrences based on weekly periods. The default week start is got from calendar.firstweekday(), and may be modified by calendar.setfirstweekday().

- **count** – How many occurrences will be generated.

Note: As of version 2.5.0, the use of the **until** keyword together with the **count** keyword is deprecated per RFC-2445 Sec. 4.3.10.

- **until** – If given, this must be a datetime instance, that will specify the limit of the recurrence. The last recurrence in the rule is the greatest datetime that is less than or equal to the value specified in the **until** parameter.

Note: As of version 2.5.0, the use of the **until** keyword together with the **count** keyword is deprecated per RFC-2445 Sec. 4.3.10.

- **bysetpos** – If given, it must be either an integer, or a sequence of integers, positive or negative. Each given integer will specify an occurrence number, corresponding to the nth occurrence of the rule inside the frequency period. For example, a bysetpos of -1 if combined with a MONTHLY frequency, and a byweekday of (MO, TU, WE, TH, FR), will result in the last work day of every month.

- **bymonth** – If given, it must be either an integer, or a sequence of integers, meaning the months to apply the recurrence to.

- **bymonthday** – If given, it must be either an integer, or a sequence of integers, meaning the month days to apply the recurrence to.

- **byyearday** – If given, it must be either an integer, or a sequence of integers, meaning the year days to apply the recurrence to.

- **byweekno** – If given, it must be either an integer, or a sequence of integers, meaning the week numbers to apply the recurrence to. Week numbers have the meaning described in ISO8601, that is, the first week of the year is that containing at least four days of the new year.

- **byweekday** – If given, it must be either an integer (0 == MO), a sequence of integers, one of the weekday constants (MO, TU, etc), or a sequence of these constants. When given, these variables will define the weekdays where the recurrence will be applied. It’s also possible to use an argument n for the weekday instances, which will mean the nth occurrence of this weekday in the period. For
example, with MONTHLY, or with YEARLY and BYMONTH, using FR(+1) in
byweekday will specify the first friday of the month where the recurrence hap-
pens. Notice that in the RFC documentation, this is specified as BYDAY, but was
renamed to avoid the ambiguity of that keyword.

• **byhour** – If given, it must be either an integer, or a sequence of integers, meaning
  the hours to apply the recurrence to.

• **byminute** – If given, it must be either an integer, or a sequence of integers, mean-
ing the minutes to apply the recurrence to.

• **bysecond** – If given, it must be either an integer, or a sequence of integers, mean-
ing the seconds to apply the recurrence to.

• **byeaster** – If given, it must be either an integer, or a sequence of integers, posi-
tive or negative. Each integer will define an offset from the Easter Sunday. Passing
  the offset 0 to byeaster will yield the Easter Sunday itself. This is an extension to
  the RFC specification.

**replace(**kwargs**)

Return new rrule with same attributes except for those attributes given new values by whichever
keyword arguments are specified.

**class** matplotlib.dates.*relativedelta*(\dt1=None, dt2=None, years=0, months=0, days=0,
leapdays=0, weeks=0, hours=0, minutes=0, seconds=0, microseconds=0, year=None, month=None,
day=None, weekday=None, yearday=None, nlyear-day=None, hour=None, minute=None, second=None,
microsecond=None)

Bases: object

The relativedelta type is based on the specification of the excellent work done by M.-A. Lemburg in
his mx.DateTime extension. However, notice that this type does NOT implement the same algorithm
as his work. Do NOT expect it to behave like mx.DateTime’s counterpart.

There are two different ways to build a relativedelta instance. The first one is passing it two
date/datetime classes:

```python
relativedelta(datetimel, datetime2)
```

The second one is passing it any number of the following keyword arguments:

```python
relativedelta(arg1=x, arg2=y, arg3=z...)
```

- **year, month, day, hour, minute, second, microsecond:**
  Absolute information (argument is singular); adding or subtracting a
  relativedelta with absolute information does not perform an arithme-
tic operation, but rather REPLACES the corresponding value in the
  original datetime with the value(s) in relativedelta.

- **years, months, weeks, days, hours, minutes, seconds, microseconds:**
  Relative information, may be negative (argument is plural); adding
  or subtracting a relativedelta with relative information performs
  the corresponding arithmetic operation on the original datetime value.
weekday:
  One of the weekday instances (MO, TU, etc). These instances may receive a parameter N, specifying the Nth weekday, which could be positive or negative (like MO(+1) or MO(-2)). Not specifying it is the same as specifying +1. You can also use an integer, where 0=MO.

leapdays:
  Will add given days to the date found, if year is a leap year, and the date found is post 28 of February.

yearday, nlyearday:
  Set the yearday or the non-leap year day (jump leap days). These are converted to day/month/leapdays information.

Here is the behavior of operations with relativedelta:

  1. Calculate the absolute year, using the ‘year’ argument, or the original datetime year, if the argument is not present.
  2. Add the relative ‘years’ argument to the absolute year.
  3. Do steps 1 and 2 for month/months.
  4. Calculate the absolute day, using the ‘day’ argument, or the original datetime day, if the argument is not present. Then, subtract from the day until it fits in the year and month found after their operations.
  5. Add the relative ‘days’ argument to the absolute day. Notice that the ‘weeks’ argument is multiplied by 7 and added to ‘days’.
  6. Do steps 1 and 2 for hour/hours, minute/minutes, second/seconds, microsecond/microseconds.
  7. If the ‘weekday’ argument is present, calculate the weekday, with the given (wday, nth) tuple. wday is the index of the weekday (0-6, 0=Mon), and nth is the number of weeks to add forward or backward, depending on its signal. Notice that if the calculated date is already Monday, for example, using (0, 1) or (0, -1) won’t change the day.

normalized()
  Return a version of this object represented entirely using integer values for the relative attributes.

>>> relativedelta(days=1.5, hours=2).normalized()
relativedelta(days=1, hours=14)

Returns Returns a dateutil.relativedelta.relativedelta object.

weeks

matplotlib.dates.seconds(s)
  Return seconds as days.
matplotlib.dates.minutes(m)
    Return minutes as days.

matplotlib.dates.hours(h)
    Return hours as days.

matplotlib.dates.weeks(w)
    Return weeks as days.
29.1 matplotlib.dviread

An experimental module for reading dvi files output by TeX. Several limitations make this not (currently) useful as a general-purpose dvi preprocessor, but it is currently used by the pdf backend for processing usetex text.

Interface:

```python
dvi = Dvi(filename, 72)
# iterate over pages (but only one page is supported for now):
for page in dvi:
    w, h, d = page.width, page.height, page.descent
    for x,y,font,glyph,width in page.text:
        fontname = font.texname
        pointsize = font.size
        ...
    for x,y,height,width in page.boxes:
        ...
```

```python
class matplotlib.dviread.Dvi(filename, dpi)
    Bases: object

    A dvi (“device-independent”) file, as produced by TeX. The current implementation only reads the first page and does not even attempt to verify the postamble.

    Initialize the object. This takes the filename as input and opens the file; actually reading the file happens when iterating through the pages of the file.

    close()
        Close the underlying file if it is open.

class matplotlib.dviread.DviFont(scale, tfm, texname, vf)
    Bases: object

    Object that holds a font’s texname and size, supports comparison, and knows the widths of glyphs in the same units as the AFM file. There are also internal attributes (for use by dviread.py) that are not used for comparison.

    The size is in Adobe points (converted from TeX points).
```
texname
Name of the font as used internally by TeX and friends. This is usually very different from any external font names, and dviread.PsfontsMap can be used to find the external name of the font.

size
Size of the font in Adobe points, converted from the slightly smaller TeX points.

widths
Widths of glyphs in glyph-space units, typically 1/1000ths of the point size.

size
texname
widths

class matplotlib.dviread.Encoding(filename)
Bases: object

Parses a *.enc file referenced from a psfonts.map style file. The format this class understands is a very limited subset of PostScript.

Usage (subject to change):

```
for name in Encoding(filename):
    whatever(name)
```

class matplotlib.dviread.PsfontsMap(filename)
Bases: object

A psfonts.map formatted file, mapping TeX fonts to PS fonts. Usage:

```
>>> map = PsfontsMap(find_tex_file('pdftex.map'))
>>> entry = map['ptmbo8r']
>>> entry.texname
'ptmbo8r'
>>> entry.psname
'Times-Bold'
>>> entry.encoding
'/usr/local/texlive/2008/texmf-dist/fonts/enc/dvips/base/8r.enc'
>>> entry.effects
{'slant': 0.16700000000000001}
>>> entry.filename
```

For historical reasons, TeX knows many Type-1 fonts by different names than the outside world. (For one thing, the names have to fit in eight characters.) Also, TeX’s native fonts are not Type-1 but Metafont, which is nontrivial to convert to PostScript except as a bitmap. While high-quality conversions to Type-1 format exist and are shipped with modern TeX distributions, we need to know
which Type-1 fonts are the counterparts of which native fonts. For these reasons a mapping is needed from internal font names to font file names.

A `texmf` tree typically includes mapping files called e.g. `psfonts.map`, `pdftex.map`, `dvipdfm.map`. `psfonts.map` is used by dvips, `pdftex.map` by pdfTeX, and `dvipdfm.map` by dvipdfm. `psfonts.map` might avoid embedding the 35 PostScript fonts (i.e., have no filename for them, as in the Times-Bold example above), while the pdf-related files perhaps only avoid the “Base 14” pdf fonts. But the user may have configured these files differently.

```python
class matplotlib.dviread.Tfm(filename)
    Bases: object

    A TeX Font Metric file. This implementation covers only the bare minimum needed by the Dvi class.

    checksum
        Used for verifying against the dvi file.

    design_size
        Design size of the font (in what units?)

    width
        Width of each character, needs to be scaled by the factor specified in the dvi file. This is a dict because indexing may not start from 0.

    height
        Height of each character.

    depth
        Depth of each character.

    checksum

    depth

    design_size

    height

    width
```

```python
class matplotlib.dviread.Vf(filename)
    Bases: matplotlib.dviread.Dvi

    A virtual font (*.vf file) containing subroutines for dvi files.

    Usage:

    ```python
    vf = Vf(filename)
glyph = vf[code]
glyph.text, glyph.boxes, glyph.width
    ```
```
Call \texttt{kpsewhich} to find a file in the texmf tree. If \texttt{format} is not None, it is used as the value for the \texttt{--format} option.

Apparently most existing TeX distributions on Unix-like systems use kpathsea. I hear MikTeX (a popular distribution on Windows) doesn’t use kpathsea, so what do we do? (TODO)

\textbf{See also:}

\textbf{Kpathsea documentation} The library that \texttt{kpsewhich} is part of.
30.1 matplotlib.figure

The figure module provides the top-level Artist, the Figure, which contains all the plot elements. The following classes are defined

SubplotParams control the default spacing of the subplots

Figure top level container for all plot elements

class matplotlib.figure.AxesStack
    Bases: matplotlib.cbook.Stack

    Specialization of the Stack to handle all tracking of Axes in a Figure. This stack stores key, (ind, axes) pairs, where:
    • key should be a hash of the args and kwargs used in generating the Axes.
    • ind is a serial number for tracking the order in which axes were added.

    The AxesStack is a callable, where ax_stack() returns the current axes. Alternatively the current_key_axes() will return the current key and associated axes.

    add(key, a)
        Add Axes a, with key key, to the stack, and return the stack.

        If a is already on the stack, don’t add it again, but return None.

    as_list()
        Return a list of the Axes instances that have been added to the figure

    bubble(a)
        Move the given axes, which must already exist in the stack, to the top.

    current_key_axes()
        Return a tuple of (key, axes) for the active axes.

        If no axes exists on the stack, then returns (None, None).

    get(key)
        Return the Axes instance that was added with key. If it is not present, return None.
remove(a)

Remove the axes from the stack.

class matplotlib.figure.Figure(figsize=None, dpi=None, facecolor=None, edgecolor=None, linewidth=0.0, frameon=None, subplotpars=None, tight_layout=None)

Bases: matplotlib.artist.Artist

The Figure instance supports callbacks through a callbacks attribute which is a matplotlib.cbook.CallbackRegistry instance. The events you can connect to are ‘dpi_changed’, and the callback will be called with func(fig) where fig is the Figure instance.

patch The figure patch is drawn by a matplotlib.patches.Rectangle instance

suppressComposite For multiple figure images, the figure will make composite images depending on the renderer option_image_nocomposite function. If suppressComposite is True|False, this will override the renderer.

figsize w,h tuple in inches

dpi Dots per inch

facecolor The figure patch facecolor; defaults to rc figure.facecolor

edgecolor The figure patch edge color; defaults to rc figure.edgecolor

linewidth The figure patch edge linewidth; the default linewidth of the frame

frameon If False, suppress drawing the figure frame

subplotpars A SubplotParams instance, defaults to rc

tight_layout If False use subplotpars; if True adjust subplot parameters using tight_layout() with default padding. When providing a dict containing the keys pad, w_pad, h_pad and rect, the default tight_layout() paddings will be overridden. Defaults to rc figure.autolayout.

add_axes(*args, **kwargs)

Add an axes at position rect [left, bottom, width, height] where all quantities are in fractions of figure width and height. kwargs are legal Axes kwargs plus projection which sets the projection type of the axes. (For backward compatibility, polar=True may also be provided, which is equivalent to projection='polar'). Valid values for projection are: ['aitoff', 'hammer', 'lambert', 'mollweide', 'polar', 'rectilinear']. Some of these projections support additional kwargs, which may be provided to add_axes(). Typical usage:

rect = l,b,w,h
fig.add_axes(rect)
fig.add_axes(rect, frameon=False, facecolor='g')
fig.add_axes(rect, polar=True)
fig.add_axes(rect, projection='polar')
fig.add_axes(ax)

If the figure already has an axes with the same parameters, then it will simply make that axes current and return it. If you do not want this behavior, e.g., you want to force the creation of a new Axes, you must use a unique set of args and kwargs. The axes label attribute has been
exposed for this purpose. e.g., if you want two axes that are otherwise identical to be added to the figure, make sure you give them unique labels:

```python
fig.add_axes(rect, label='axes1')
fig.add_axes(rect, label='axes2')
```

In rare circumstances, add_axes may be called with a single argument, an Axes instance already created in the present figure but not in the figure’s list of axes. For example, if an axes has been removed with `delaxes()`, it can be restored with:

```python
fig.add_axes(ax)
```

In all cases, the `Axes` instance will be returned.

In addition to `projection`, the following kwargs are supported:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>adjustable</td>
<td>['box'</td>
</tr>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>anchor</td>
<td>unknown</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>aspect</td>
<td>unknown</td>
</tr>
<tr>
<td>autoscale_on</td>
<td>unknown</td>
</tr>
<tr>
<td>autoscalex_on</td>
<td>unknown</td>
</tr>
<tr>
<td>autoscaley_on</td>
<td>unknown</td>
</tr>
<tr>
<td>axes</td>
<td>an <code>Axes</code> instance</td>
</tr>
<tr>
<td>axes_locator</td>
<td>unknown</td>
</tr>
<tr>
<td>axisbelow</td>
<td>[True</td>
</tr>
<tr>
<td>clip_box</td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color_cycle</td>
<td>unknown</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>facecolor</td>
<td>unknown</td>
</tr>
<tr>
<td>fc</td>
<td>unknown</td>
</tr>
<tr>
<td>figure</td>
<td>unknown</td>
</tr>
<tr>
<td>frame_on</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>navigate</td>
<td>[True</td>
</tr>
<tr>
<td>navigate_mode</td>
<td>unknown</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>position</td>
<td>unknown</td>
</tr>
<tr>
<td>rasterization_zorder</td>
<td>unknown</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
</tbody>
</table>

Continued on next page
Table 30.1 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>title</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xbound</td>
<td>unknown</td>
</tr>
<tr>
<td>xlabel</td>
<td>unknown</td>
</tr>
<tr>
<td>xlim</td>
<td>unknown</td>
</tr>
<tr>
<td>xmargin</td>
<td>unknown</td>
</tr>
<tr>
<td>xscale</td>
<td>['linear'</td>
</tr>
<tr>
<td>xticklabels</td>
<td>sequence of strings</td>
</tr>
<tr>
<td>xticks</td>
<td>sequence of floats</td>
</tr>
<tr>
<td>ybound</td>
<td>unknown</td>
</tr>
<tr>
<td>ylabel</td>
<td>unknown</td>
</tr>
<tr>
<td>ylim</td>
<td>unknown</td>
</tr>
<tr>
<td>ymargin</td>
<td>unknown</td>
</tr>
<tr>
<td>yscale</td>
<td>['linear'</td>
</tr>
<tr>
<td>yticklabels</td>
<td>sequence of strings</td>
</tr>
<tr>
<td>yticks</td>
<td>sequence of floats</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

**add_axobserver**(*func*)

Whenever the axes state change, `func(self)` will be called.

**add_subplot**(*args, **kwargs*)

Add a subplot. Examples:

```python
fig.add_subplot(111)
# equivalent but more general
fig.add_subplot(1,1,1)

# add subplot with red background
fig.add_subplot(212, facecolor='r')

# add a polar subplot
fig.add_subplot(111, projection='polar')

# add Subplot instance sub
fig.add_subplot(sub)
```

`kwargs` are legal `Axes` `kwargs` plus `projection`, which chooses a projection type for the axes. (For backward compatibility, `polar=True` may also be provided, which is equivalent to `projection='polar'`). Valid values for `projection` are: ['aitoff', 'hammer', 'lambert', 'mollweide', 'po-
lar’, ‘rectilinear’]. Some of these projections support additional *kwargs*, which may be provided to *add_axes()*.

The *Axes* instance will be returned.

If the figure already has a subplot with key (*args, *kwargs*) then it will simply make that subplot current and return it.

**See also:**

*subplot()* for an explanation of the args.

The following *kwargs* are supported:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>adjustable</td>
<td>['box'</td>
</tr>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>anchor</td>
<td>unknown</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>aspect</td>
<td>unknown</td>
</tr>
<tr>
<td>autoscale_on</td>
<td>unknown</td>
</tr>
<tr>
<td>autoscalex_on</td>
<td>unknown</td>
</tr>
<tr>
<td>autoscaley_on</td>
<td>unknown</td>
</tr>
<tr>
<td>axes</td>
<td>an <em>Axes</em> instance</td>
</tr>
<tr>
<td>axes_locator</td>
<td>unknown</td>
</tr>
<tr>
<td>axisbelow</td>
<td>[True</td>
</tr>
<tr>
<td>clip_box</td>
<td>a <em>matplotlib.transforms.Bbox</em> instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color_cycle</td>
<td>unknown</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>facecolor</td>
<td>unknown</td>
</tr>
<tr>
<td>fc</td>
<td>unknown</td>
</tr>
<tr>
<td>figure</td>
<td>unknown</td>
</tr>
<tr>
<td>frame_on</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>navigate</td>
<td>[True</td>
</tr>
<tr>
<td>navigate_mode</td>
<td>unknown</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None]float[boolean]callable]</td>
</tr>
<tr>
<td>position</td>
<td>unknown</td>
</tr>
<tr>
<td>rasterization_zorder</td>
<td>unknown</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>title</td>
<td>unknown</td>
</tr>
</tbody>
</table>

Continued on next page
Table 30.2 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>transform</code></td>
<td><code>Transform</code> instance</td>
</tr>
<tr>
<td><code>url</code></td>
<td>a url string</td>
</tr>
<tr>
<td><code>visible</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>xbound</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>xlabel</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>xlim</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>xmargin</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>xscale</code></td>
<td>['linear'</td>
</tr>
<tr>
<td><code>xticklabels</code></td>
<td>sequence of strings</td>
</tr>
<tr>
<td><code>xticks</code></td>
<td>sequence of floats</td>
</tr>
<tr>
<td><code>ybound</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>ylabel</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>ylim</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>ymargin</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>yscale</code></td>
<td>['linear'</td>
</tr>
<tr>
<td><code>yticklabels</code></td>
<td>sequence of strings</td>
</tr>
<tr>
<td><code>yticks</code></td>
<td>sequence of floats</td>
</tr>
<tr>
<td><code>zorder</code></td>
<td>any number</td>
</tr>
</tbody>
</table>

`autofmt_xdate` *(bottom=0.2, rotation=30, ha='right')*

Date ticklabels often overlap, so it is useful to rotate them and right align them. Also, a common use case is a number of subplots with shared xaxes where the x-axis is date data. The ticklabels are often long, and it helps to rotate them on the bottom subplot and turn them off on other subplots, as well as turn off xlabel.

- **bottom** The bottom of the subplots for `subplots_adjust()`
- **rotation** The rotation of the xtick labels
- **ha** The horizontal alignment of the xticklabels

`axes`  
Read-only: list of axes in Figure

`clear()`  
Clear the figure – synonym for `clf()`.

`clf(keep_observers=False)`  
Clear the figure.

Set `keep_observers` to True if, for example, a gui widget is tracking the axes in the figure.

`colorbar(mappable, cax=None, ax=None, use_gridspec=True, **kw)`  
Create a colorbar for a ScalarMappable instance, `mappable`.

Documentation for the pylab thin wrapper:

Add a colorbar to a plot.
Function signatures for the `pyplot` interface; all but the first are also method signatures for the `colorbar()` method:

```
colorbar(**kwargs)
colorbar(mappable, **kwargs)
colorbar(mappable, cax=cax, **kwargs)
colorbar(mappable, ax=ax, **kwargs)
```

arguments:

- **mappable** the `Image`, `ContourSet`, etc. to which the colorbar applies; this argument is mandatory for the `colorbar()` method but optional for the `colorbar()` function, which sets the default to the current image.

keyword arguments:

- **cax** `None` | axes object into which the colorbar will be drawn
- **ax** `None` | parent axes object(s) from which space for a new colorbar axes will be stolen. If a list of axes is given they will all be resized to make room for the colorbar axes.
- **use_gridspec** `False` | If `cax` is `None`, a new `cax` is created as an instance of Axes. If `ax` is an instance of Subplot and `use_gridspec` is `True`, `cax` is created as an instance of Subplot using the `grid_spec` module.

Additional keyword arguments are of two kinds:

axes properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>orientation</td>
<td>vertical or horizontal</td>
</tr>
<tr>
<td>fraction</td>
<td>0.15; fraction of original axes to use for colorbar</td>
</tr>
<tr>
<td>pad</td>
<td>0.05 if vertical, 0.15 if horizontal; fraction of original axes between colorbar and new image axes</td>
</tr>
<tr>
<td>shrink</td>
<td>1.0; fraction by which to shrink the colorbar</td>
</tr>
<tr>
<td>aspect</td>
<td>20; ratio of long to short dimensions</td>
</tr>
<tr>
<td>anchor</td>
<td>(0.0, 0.5) if vertical; (0.5, 1.0) if horizontal; the anchor point of the colorbar axes</td>
</tr>
<tr>
<td>panchor</td>
<td>(1.0, 0.5) if vertical; (0.5, 0.0) if horizontal; the anchor point of the colorbar parent axes. If False, the parent axes’ anchor will be unchanged</td>
</tr>
</tbody>
</table>

colorbar properties:
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>extend</code></td>
<td>[ 'neither'</td>
</tr>
<tr>
<td><code>extendfrac</code></td>
<td>[ None</td>
</tr>
<tr>
<td><code>extendrect</code></td>
<td>[ False</td>
</tr>
<tr>
<td><code>spacing</code></td>
<td>[ ‘uniform’</td>
</tr>
<tr>
<td><code>ticks</code></td>
<td>[ None</td>
</tr>
<tr>
<td><code>format</code></td>
<td>[ None</td>
</tr>
<tr>
<td><code>drawedges</code></td>
<td>[ False</td>
</tr>
</tbody>
</table>

The following will probably be useful only in the context of indexed colors (that is, when the mappable has `norm=NoNorm()`), or other unusual circumstances.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>boundaries</code></td>
<td>None or a sequence</td>
</tr>
<tr>
<td><code>values</code></td>
<td>None or a sequence which must be of length 1 less than the sequence of <code>boundaries</code>. For each region delimited by adjacent entries in <code>boundaries</code>, the color mapped to the corresponding value in <code>values</code> will be used.</td>
</tr>
</tbody>
</table>

If `mappable` is a `ContourSet`, its `extend` kwarg is included automatically.

Note that the `shrink` kwarg provides a simple way to keep a vertical colorbar, for example, from being taller than the axes of the mappable to which the colorbar is attached; but it is a manual method requiring some trial and error. If the colorbar is too tall (or a horizontal colorbar is too wide) use a smaller value of `shrink`. 
For more precise control, you can manually specify the positions of the axes objects in which the mappable and the colorbar are drawn. In this case, do not use any of the axes properties kwargs.

It is known that some vector graphics viewer (svg and pdf) renders white gaps between segments of the colorbar. This is due to bugs in the viewers not matplotlib. As a workaround the colorbar can be rendered with overlapping segments:

```python
cbar = colorbar()
cbar.solids.set_edgecolor("face")
draw()
```

However this has negative consequences in other circumstances. Particularly with semi transparent images (alpha < 1) and colorbar extensions and is not enabled by default see (issue #1188).

**returns:** Colorbar instance; see also its base class, ColorbarBase. Call the set_label() method to label the colorbar.

**contains** (mouseevent)
Test whether the mouse event occurred on the figure.

Returns True,{}

**delaxes** (a)
remove a from the figure and update the current axes

**dpi**

**draw** (artist, renderer, *args, **kwargs)
Render the figure using matplotlib.backend_bases.RendererBase instance renderer.

**draw_artist** (a)
draw matplotlib.artist.Artist instance a only – this is available only after the figure is drawn

**figimage** (X, xo=0, yo=0, alpha=None, norm=None, cmap=None, vmin=None, vmax=None, origin=None, resize=False, **kwargs)
Adds a non-resampled image to the figure.

call signatures:

```python
figimage(X, **kwargs)
```

adds a non-resampled array X to the figure.

```python
figimage(X, xo, yo)
```

with pixel offsets xo, yo,

X must be a float array:

- If X is MxN, assume luminance (grayscale)
- If X is MxNx3, assume RGB
- If X is MxNx4, assume RGBA
Optional keyword arguments:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>resize</td>
<td>a boolean, True or False. If “True”, then re-size the Figure to match the given image size.</td>
</tr>
<tr>
<td>xo or yo</td>
<td>An integer, the x and y image offset in pixels</td>
</tr>
<tr>
<td>cmap</td>
<td>a matplotlib.colors.Colormap instance, e.g., cm.jet. If None, default to the rc image.cmap value</td>
</tr>
<tr>
<td>norm</td>
<td>a matplotlib.colors.Normalize instance. The default is normalization(). This scales luminance -&gt; 0-1</td>
</tr>
<tr>
<td>vmin, vmax</td>
<td>used to scale a luminance image to 0-1. If either is None, the min and max of the luminance values will be used. Note if you pass a norm instance, the settings for vmin and vmax will be ignored.</td>
</tr>
<tr>
<td>alpha</td>
<td>the alpha blending value, default is None</td>
</tr>
<tr>
<td>origin</td>
<td>[‘upper’</td>
</tr>
</tbody>
</table>

figimage complements the axes image (imshow()) which will be resampled to fit the current axes. If you want a resampled image to fill the entire figure, you can define an Axes with extent [0,0,1,1].

An matplotlib.image.FigureImage instance is returned.
Additional kwargs are Artist kwargs passed on to `FigureImage`

**gca(**kwargs**)

Get the current axes, creating one if necessary

The following kwargs are supported for ensuring the returned axes adheres to the given projection etc., and for axes creation if the active axes does not exist:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>adjustable</td>
<td>['box'</td>
</tr>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>anchor</td>
<td>unknown</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>aspect</td>
<td>unknown</td>
</tr>
<tr>
<td>autoscale_on</td>
<td>unknown</td>
</tr>
<tr>
<td>autoscalex_on</td>
<td>unknown</td>
</tr>
<tr>
<td>autoscaley_on</td>
<td>unknown</td>
</tr>
<tr>
<td>axes</td>
<td>an <code>Axes</code> instance</td>
</tr>
<tr>
<td>axes_locator</td>
<td>unknown</td>
</tr>
<tr>
<td>axisbelow</td>
<td>[ True</td>
</tr>
<tr>
<td>clip_box</td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
</tbody>
</table>

Continued on next page
Table 30.3 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clip_path</td>
<td>((Path, Transform)</td>
</tr>
<tr>
<td>color_cycle</td>
<td>unknown</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>facecolor</td>
<td>unknown</td>
</tr>
<tr>
<td>fc</td>
<td>unknown</td>
</tr>
<tr>
<td>figure</td>
<td>unknown</td>
</tr>
<tr>
<td>frame_on</td>
<td>[ True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>navigate</td>
<td>[ True</td>
</tr>
<tr>
<td>navigate_mode</td>
<td>unknown</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>position</td>
<td>unknown</td>
</tr>
<tr>
<td>rasterization_zorder</td>
<td>unknown</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>title</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xbound</td>
<td>unknown</td>
</tr>
<tr>
<td>xlabel</td>
<td>unknown</td>
</tr>
<tr>
<td>xlim</td>
<td>unknown</td>
</tr>
<tr>
<td>xmargin</td>
<td>unknown</td>
</tr>
<tr>
<td>xscale</td>
<td>['linear'</td>
</tr>
<tr>
<td>xticklabels</td>
<td>sequence of strings</td>
</tr>
<tr>
<td>xticks</td>
<td>sequence of floats</td>
</tr>
<tr>
<td>ybound</td>
<td>unknown</td>
</tr>
<tr>
<td>ylabel</td>
<td>unknown</td>
</tr>
<tr>
<td>ylim</td>
<td>unknown</td>
</tr>
<tr>
<td>ymargin</td>
<td>unknown</td>
</tr>
<tr>
<td>yscale</td>
<td>['linear'</td>
</tr>
<tr>
<td>yticklabels</td>
<td>sequence of strings</td>
</tr>
<tr>
<td>ticks</td>
<td>sequence of floats</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

**get_axes()**

**get_children()**

get a list of artists contained in the figure
get_default_bbox_extra_artists()

get_dpi()
    Return the dpi as a float

get_edgecolor()
    Get the edge color of the Figure rectangle

get_facecolor()
    Get the face color of the Figure rectangle

get_figheight()
    Return the figheight as a float

get_figwidth()
    Return the figwidth as a float

get_frameon()
    get the boolean indicating frameon

get_size_inches()
    Returns the current size of the figure in inches (1in == 2.54cm) as an numpy array.

    Returns size : ndarray
        The size of the figure in inches

    See also:
        matplotlib.Figure.set_size_inches

get_tight_layout()
    Return the Boolean flag, True to use :meth\'tight_layout\' when drawing.

get_tightbbox(renderer)
    Return a (tight) bounding box of the figure in inches.

    It only accounts axes title, axis labels, and axis ticklabels. Needs improvement.

get_window_extent(*args, **kwargs)
    get the figure bounding box in display space; kwargs are void

ginput(n=1, timeout=30, show_clicks=True, mouse_add=1, mouse_pop=3, mouse_stop=2)
    Blocking call to interact with the figure.

    This will wait for n clicks from the user and return a list of the coordinates of each click.

    If timeout is zero or negative, does not timeout.

    If n is zero or negative, accumulate clicks until a middle click (or potentially both mouse buttons
    at once) terminates the input.

    Right clicking cancels last input.

    The buttons used for the various actions (adding points, removing points, terminating the inputs) can be overliden via the arguments mouse_add, mouse_pop and mouse_stop, that give the associated mouse button: 1 for left, 2 for middle, 3 for right.
The keyboard can also be used to select points in case your mouse does not have one or more of
the buttons. The delete and backspace keys act like right clicking (i.e., remove last point), the
enter key terminates input and any other key (not already used by the window manager) selects
a point.

**hold**(*b=None*)

Deprecated since version 2.0: The hold function was deprecated in version 2.0.

Set the hold state. If hold is None (default), toggle the hold state. Else set the hold state to
boolean value b.

e.g.:

```python
hold()       # toggle hold
hold(True)   # hold is on
hold(False)  # hold is off
```

All “hold” machinery is deprecated.

**legend**(*handles, labels, *args, **kwargs*)

Place a legend in the figure. Labels are a sequence of strings, handles is a sequence of Line2D
or Patch instances, and loc can be a string or an integer specifying the legend location

**USAGE:**

```python
legend( (line1, line2, line3),
       ('label1', 'label2', 'label3'),
       'upper right')
```

The *loc* location codes are:

```python
'best' : 0,  (currently not supported for figure legends)
'upper right' : 1,
'upper left' : 2,
'lower left' : 3,
'lower right' : 4,
'right' : 5,
'center left' : 6,
'center right' : 7,
'lower center' : 8,
'upper center' : 9,
'center' : 10,
```

*loc* can also be an (x,y) tuple in figure coords, which specifies the lower left of the legend box.
figure coords are (0,0) is the left, bottom of the figure and 1,1 is the right, top.

Keyword arguments:

- **prop**: [None | FontProperties | dict] A matplotlib.font_manager.FontProperties instance. If *prop* is a dictionary, a new instance will be created with *prop*. If None, use rc settings.

- **numpoints**: integer The number of points in the legend line, default is 4

- **scatterpoints**: integer The number of points in the legend line, default is 4
scatteryoffsets: list of floats a list of yoffsets for scatter symbols in legend

markerscale: [ None | scalar ] The relative size of legend markers vs. original. If None, use rc settings.

markerfirst: [ True | False ] if True, legend marker is placed to the left of the legend label if False, legend marker is placed to the right of the legend label

frameon: [ None | bool ] Control whether the legend should be drawn on a patch (frame). Default is None which will take the value from the legend.frameon rcParam.

fancybox: [ None | False | True ] if True, draw a frame with a round fancybox. If None, use rc

shadow: [ None | False | True ] If True, draw a shadow behind legend. If None, use rc settings.

framealpha: [ None | float ] Control the alpha transparency of the legend’s background. Default is None which will take the value from the legend.framealpha rcParam.

facecolor: [ None | “inherit” | a color spec ] Control the legend’s background color. Default is None which will take the value from the legend.facecolor rcParam. If "inherit", it will take the axes.facecolor rcParam.

edgecolor: [ None | “inherit” | a color spec ] Control the legend’s background patch edge color. Default is None which will take the value from the legend.edgecolor rcParam. If "inherit", it will take the axes.edgecolor rcParam.

ncol [integer] number of columns. default is 1

mode [[ “expand” | None ]] if mode is “expand”, the legend will be horizontally expanded to fill the axes area (or bbox_to_anchor)

title [string] the legend title

Padding and spacing between various elements use following keywords parameters. The dimensions of these values are given as a fraction of the fontsize. Values from rcParams will be used if None.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>borderpad</td>
<td>the fractional whitespace inside the legend border</td>
</tr>
<tr>
<td>labelspacing</td>
<td>the vertical space between the legend entries</td>
</tr>
<tr>
<td>handlelength</td>
<td>the length of the legend handles</td>
</tr>
<tr>
<td>handletextpad</td>
<td>the pad between the legend handle and text</td>
</tr>
<tr>
<td>borderaxespad</td>
<td>the pad between the axes and legend border</td>
</tr>
<tr>
<td>columnsspacing</td>
<td>the spacing between columns</td>
</tr>
</tbody>
</table>

Note: Not all kinds of artist are supported by the legend. See LINK (FIXME) for details.

Example:
savefig(*args, **kwargs)
Save the current figure.

Call signature:
```python
savefig(fname, dpi=None, facecolor='w', edgecolor='w', orientation='portrait', papertype=None, format=None, transparent=False, bbox_inches=None, pad_inches=0.1, frameon=None)
```

The output formats available depend on the backend being used.

Arguments:

`fname`: A string containing a path to a filename, or a Python file-like object, or possibly some backend-dependent object such as `PdfPages`.

If `format` is `None` and `fname` is a string, the output format is deduced from the extension of the filename. If the filename has no extension, the value of the rc parameter `savefig.format` is used.

If `fname` is not a string, remember to specify `format` to ensure that the correct backend is used.

Keyword arguments:

`dpi`: `[None | scalar > 0 | ‘figure’] The resolution in dots per inch. If `None` it will
default to the value `savefig.dpi` in the matplotlibrc file. If `figure` it will set the
dpi to be the value of the figure.

**facecolor, edgecolor:** the colors of the figure rectangle

**orientation:** [‘landscape’ | ‘portrait’] not supported on all backends; currently only
on postscript output

**papertype:** One of ‘letter’, ‘legal’, ‘executive’, ‘ledger’, ‘a0’ through ‘a10’, ‘b0’
through ‘b10’. Only supported for postscript output.

**format:** One of the file extensions supported by the active backend. Most backends
support png, pdf, ps, eps and svg.

**transparent:** If `True`, the axes patches will all be transparent; the figure patch will also
be transparent unless facecolor and/or edgecolor are specified via kwargs. This is
useful, for example, for displaying a plot on top of a colored background on a web
page. The transparency of these patches will be restored to their original values
upon exit of this function.

**frameon:** If `True`, the figure patch will be colored, if `False`, the figure background will
be transparent. If not provided, the rcParam `savefig.frameon` will be used.

**bbox_inches:** Bbox in inches. Only the given portion of the figure is saved. If `tight`,
try to figure out the tight bbox of the figure.

**pad_inches:** Amount of padding around the figure when bbox_inches is ‘tight’.

**bbox_extra_artists:** A list of extra artists that will be considered when the tight bbox
is calculated.

**sca**(a)
Set the current axes to be a and return a

**set_canvas**(canvas)
Set the canvas that contains the figure

ACCEPTS: a FigureCanvas instance

**set_dpi**(val)
Set the dots-per-inch of the figure

ACCEPTS: float

**set_edgecolor**(color)
Set the edge color of the Figure rectangle

ACCEPTS: any matplotlib color - see help(colors)

**set_facecolor**(color)
Set the face color of the Figure rectangle

ACCEPTS: any matplotlib color - see help(colors)

**set_figheight**(val, forward=False)
Set the height of the figure in inches

ACCEPTS: float
**set_figwidth**(val, forward=False)
Set the width of the figure in inches

ACCEPTS: float

**set_frameon**(b)
Set whether the figure frame (background) is displayed or invisible

ACCEPTS: boolean

**set_size_inches**(w, h=None, forward=True)
Set the figure size in inches (1in == 2.54cm)

Usage

```
fig.set_size_inches(w,h)  # OR
fig.set_size_inches((w,h))
```

optional kwarg `forward=True` will cause the canvas size to be automatically updated; e.g., you can resize the figure window from the shell

ACCEPTS: a w,h tuple with w,h in inches

See also:

matplotlib.Figure.get_size_inches

**set_tight_layout**(tight)
Set whether `tight_layout()` is used upon drawing. If None, the rcParams['figure.autolayout'] value will be set.

When providing a dict containing the keys pad, w_pad, h_pad and rect, the default `tight_layout()` paddings will be overridden.

ACCEPTS: [True | False | dict | None ]

**show**(warn=True)
If using a GUI backend with pyplot, display the figure window.

If the figure was not created using `figure()`, it will lack a FigureManagerBase, and will raise an AttributeError.

For non-GUI backends, this does nothing, in which case a warning will be issued if warn is True (default).

**subplots_adjust**(args, **kwargs)
Call signature:

```
subplots_adjust(left=None, bottom=None, right=None, top=None,
                wspace=None, hspace=None)
```

Update the SubplotParams with `kwargs` (defaulting to rc when `None`) and update the subplot locations

**suptitle**(t, **kwargs)
Add a centered title to the figure.
kwargs are `matplotlib.text.Text` properties. Using figure coordinates, the defaults are:

- **x** [0.5] The x location of the text in figure coords
- **y** [0.98] The y location of the text in figure coords

**horizontalalignment** ['center'] The horizontal alignment of the text

**verticalalignment** ['top'] The vertical alignment of the text

If the `fontproperties` keyword argument is given then the rcParams defaults for `fontsize` (figure.titlesize) and `fontweight` (figure.titleweight) will be ignored in favour of the FontProperties defaults.

A `matplotlib.text.Text` instance is returned.

Example:

```python
fig.suptitle('this is the figure title', fontsize=12)
```

**text**(*x*, *y*, *s*, **args**, **kwargs)**

Add text to figure.

Call signature:

```python
text(*x*, *y*, *s*, fontdict=None, **kwargs)
```

Add text to figure at location *x*, *y* (relative 0-1 coords). See `text()` for the meaning of the other arguments.

**kwargs** control the `Text` properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>agg_filter</strong></td>
<td>unknown</td>
</tr>
<tr>
<td><strong>alpha</strong></td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td><strong>animated</strong></td>
<td>[True</td>
</tr>
<tr>
<td><strong>axes</strong></td>
<td>an Axes instance</td>
</tr>
<tr>
<td><strong>backgroundcolor</strong></td>
<td>any matplotlib color</td>
</tr>
<tr>
<td><strong>bbox</strong></td>
<td>FancyBboxPatch prop dict</td>
</tr>
<tr>
<td><strong>clip_box</strong></td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td><strong>clip_on</strong></td>
<td>[True</td>
</tr>
<tr>
<td><strong>clip_path</strong></td>
<td>[((Path, Transform)</td>
</tr>
<tr>
<td><strong>color</strong></td>
<td>any matplotlib color</td>
</tr>
<tr>
<td><strong>contains</strong></td>
<td>a callable function</td>
</tr>
<tr>
<td><strong>family</strong> or fontfamily or fontname or name</td>
<td>[FONTNAME</td>
</tr>
<tr>
<td><strong>figure</strong></td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td><strong>fontproperties</strong> or font_properties</td>
<td>a <code>matplotlib.font_manager.FontProperties</code> instance</td>
</tr>
<tr>
<td><strong>gid</strong></td>
<td>an id string</td>
</tr>
<tr>
<td><strong>horizontalalignment</strong> or <strong>ha</strong></td>
<td>[ ‘center’</td>
</tr>
<tr>
<td><strong>label</strong></td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td><strong>linespacing</strong></td>
<td>float (multiple of font size)</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>------------------------------------------</td>
</tr>
<tr>
<td>multialignment</td>
<td>['left'</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None]float</td>
</tr>
<tr>
<td>position</td>
<td>(x,y)</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>rotation</td>
<td>[ angle in degrees</td>
</tr>
<tr>
<td>rotation_mode</td>
<td>unknown</td>
</tr>
<tr>
<td>size or fontsize</td>
<td>[size in points</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>stretch or fontstretch</td>
<td>[a numeric value in range 0-1000</td>
</tr>
<tr>
<td>style or fontstyle</td>
<td>[ ‘normal’</td>
</tr>
<tr>
<td>text</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>usetex</td>
<td>unknown</td>
</tr>
<tr>
<td>variant or fontvariant</td>
<td>[ ‘normal’</td>
</tr>
<tr>
<td>verticalalignment or ma or va</td>
<td>[ ‘center’</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>wrap</td>
<td>unknown</td>
</tr>
<tr>
<td>x</td>
<td>float</td>
</tr>
<tr>
<td>y</td>
<td>float</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

**tight_layout**(renderer=\text{None}, pad=1.08, h\_pad=\text{None}, w\_pad=\text{None}, rect=\text{None})

Adjust subplot parameters to give specified padding.

Parameters:

- **pad** [float] padding between the figure edge and the edges of subplots, as a fraction of the font-size.

- **h\_pad, w\_pad** [float] padding (height/width) between edges of adjacent subplots. Defaults to pad\_inches.

- **rect** [if rect is given, it is interpreted as a rectangle] (left, bottom, right, top) in the normalized figure coordinate that the whole subplots area (including labels) will fit into. Default is (0, 0, 1, 1).

**waitforbuttonpress**(timeout=-1)

Blocking call to interact with the figure.

This will return True if a key was pressed, False if a mouse button was pressed and None if timeout was reached without either being pressed.
If `timeout` is negative, does not timeout.

```python
class matplotlib.figure.SubplotParams(left=None, bottom=None, right=None, top=None,
                                        wspace=None, hspace=None)
```

Bases: object

A class to hold the parameters for a subplot

All dimensions are fraction of the figure width or height. All values default to their rc params

The following attributes are available

- `left` [0.125] The left side of the subplots of the figure
- `right` [0.9] The right side of the subplots of the figure
- `bottom` [0.1] The bottom of the subplots of the figure
- `top` [0.9] The top of the subplots of the figure
- `wspace` [0.2] The amount of width reserved for blank space between subplots, expressed as a fraction of the average axis width
- `hspace` [0.2] The amount of height reserved for white space between subplots, expressed as a fraction of the average axis height

```python
update(left=None, bottom=None, right=None, top=None, wspace=None, hspace=None)
```

Update the current values. If any kwarg is None, default to the current value, if set, otherwise to rc

```python
matplotlib.figure.figaspect(arg)
```

Create a figure with specified aspect ratio. If `arg` is a number, use that aspect ratio. If `arg` is an array, `figaspect` will determine the width and height for a figure that would fit array preserving aspect ratio. The figure width, height in inches are returned. Be sure to create an axes with equal with and height, e.g.,

Example usage:

```python
# make a figure twice as tall as it is wide
w, h = figaspect(2.)
fig = Figure(figsize=(w,h))
ax = fig.add_axes([0.1, 0.1, 0.8, 0.8])
ax.imshow(A, **kwargs)

# make a figure with the proper aspect for an array
A = rand(5,3)
w, h = figaspect(A)
fig = Figure(figsize=(w,h))
ax = fig.add_axes([0.1, 0.1, 0.8, 0.8])
ax.imshow(A, **kwargs)
```

Thanks to Fernando Perez for this function
31.1 matplotlib.finance

A collection of functions for collecting, analyzing and plotting financial data.
This module is deprecated in 2.0 and has been moved to a module called mpl_finance.

```python
candlestick2_ochl(ax, opens, closes, highs, lows, width=4, colorup='k',
                  colordown='r', alpha=0.75)
```
Represent the open, close as a bar line and high low range as a vertical line.
Preserves the original argument order.

**Parameters**

- `ax` : Axes
  an Axes instance to plot to

- `opens` : sequence
  sequence of opening values

- `closes` : sequence
  sequence of closing values

- `highs` : sequence
  sequence of high values

- `lows` : sequence
  sequence of low values

- `ticksize` : int
  size of open and close ticks in points

- `colorup` : color
  the color of the lines where close >= open

- `colordown` : color
  the color of the lines where close < open

- `alpha` : float
Returns ret : tuple
(lineCollection, barCollection)

Represent the open, close as a bar line and high low range as a vertical line.

NOTE: this code assumes if any value open, low, high, close is missing they all are missing

Parameters ax : Axes
   an Axes instance to plot to
   opens : sequence
   sequence of opening values
   highs : sequence
   sequence of high values
   lows : sequence
   sequence of low values
   closes : sequence
   sequence of closing values
   ticksize : int
   size of open and close ticks in points
   colorup : color
   the color of the lines where close >= open
   colordown : color
   the color of the lines where close < open
   alpha : float
   bar transparency

Returns ret : tuple
(lineCollection, barCollection)

Plot the time, open, close, high, low as a vertical line ranging from low to high. Use a rectangular bar to represent the open-close span. If close >= open, use colorup to color the bar, otherwise use colordown

Parameters ax : Axes
   an Axes instance to plot to
quotes : sequence of (time, open, close, high, low, ...) sequences

As long as the first 5 elements are these values, the record can be as long as you want (e.g., it may store volume).

time must be in float days format - see date2num

width : float

fraction of a day for the rectangle width

colorup : color

the color of the rectangle where close >= open

colordown : color

the color of the rectangle where close < open

alpha : float

the rectangle alpha level

Returns ret : tuple

returns (lines, patches) where lines is a list of lines added and patches is a list of the rectangle patches added

`matplotlib.finance.candlestick_ohlc(ax, quotes, width=0.2, colorup='k', colordown='r', alpha=1.0)`

Plot the time, open, high, low, close as a vertical line ranging from low to high. Use a rectangular bar to represent the open-close span. If close >= open, use colorup to color the bar, otherwise use colordown

Parameters

- **ax** : Axes
  
an Axes instance to plot to

- **quotes** : sequence of (time, open, high, low, close, ...) sequences
  
  As long as the first 5 elements are these values, the record can be as long as you want (e.g., it may store volume).

  time must be in float days format - see date2num

- **width** : float
  
  fraction of a day for the rectangle width

- **colorup** : color
  
  the color of the rectangle where close >= open

- **colordown** : color
  
  the color of the rectangle where close < open

- **alpha** : float
  
  the rectangle alpha level
Returns `ret`: tuple
returns (lines, patches) where lines is a list of lines added and patches is a list
of the rectangle patches added

```python
matplotlib.finance.fetch_historical_yahoo(ticker, date1, date2, cachename=None, dividends=False)
```
Fetch historical data for ticker between date1 and date2. date1 and date2 are date or datetime in-
stances, or (year, month, day) sequences.

**Parameters**

ticker: str
ticker
date1: sequence of form (year, month, day), datetime, or `date`
start date
date2: sequence of form (year, month, day), datetime, or `date`
end date
cachename: str
cachename is the name of the local file cache. If None, will default to the
md5 hash or the url (which incorporates the ticker and date range)
dividends: bool
set dividends=True to return dividends instead of price data. With this option
set, parse functions will not work

Returns `file_handle`: file handle
a file handle is returned

**Examples**

```python
>>> fh = fetch_historical_yahoo('^GSPC', (2000, 1, 1), (2001, 12, 31))
```

```python
matplotlib.finance.index_bar(ax, vals, facecolor='b', edgecolor='l', width=4, alpha=1.0)
```
Add a bar collection graph with height vals (-1 is missing).

**Parameters**

ax: Axes
an Axes instance to plot to
vals: sequence
a sequence of values
facecolor: color
the color of the bar face
decolor: color
the color of the bar edges
width : int
the bar width in points

alpha : float
bar transparency

Returns ret : barCollection
The barCollection added to the axes

matplotlib.finance.md5(x)

matplotlib.finance.parse_yahoo_historical_ochl(fh, adjusted=True, asobject=False)
Parse the historical data in file handle fh from yahoo finance.

Parameters adjusted : bool
If True (default) replace open, close, high, low prices with their adjusted values. The adjustment is by a scale factor, $S = \frac{\text{adjusted close}}{\text{close}}$. Adjusted prices are actual prices multiplied by S.

Volume is not adjusted as it is already backward split adjusted by Yahoo. If you want to compute dollars traded, multiply volume by the adjusted close, regardless of whether you choose adjusted = True|False.

asobject : bool or None
If False (default for compatibility with earlier versions) return a list of tuples containing

d, open, close, high, low, volume

If None (preferred alternative to False), return a 2-D ndarray corresponding to the list of tuples.

Otherwise return a numpy recarray with

date, year, month, day, d, open, close, high, low, volume, adjusted_close

where d is a floating poing representation of date, as returned by date2num, and date is a python standard library datetime.date instance.

The name of this kwarg is a historical artifact. Formerly, True returned a cbook Bunch holding 1-D ndarrays. The behavior of a numpy recarray is very similar to the Bunch.

matplotlib.finance.parse_yahoo_historical_ohlc(fh, adjusted=True, asobject=False)
Parse the historical data in file handle fh from yahoo finance.

Parameters adjusted : bool
If True (default) replace open, high, low, close prices with their adjusted values. The adjustment is by a scale factor, $S = \frac{\text{adjusted close}}{\text{close}}$. Adjusted prices are actual prices multiplied by S.
Volume is not adjusted as it is already backward split adjusted by Yahoo. If you want to compute dollars traded, multiply volume by the adjusted close, regardless of whether you choose adjusted = True|False.

**asobject** : bool or None

If False (default for compatibility with earlier versions) return a list of tuples containing

\[d, \text{open}, \text{high}, \text{low}, \text{close}, \text{volume}\]

If None (preferred alternative to False), return a 2-D ndarray corresponding to the list of tuples.

Otherwise return a numpy recarray with

\[\text{date, year, month, day, d, open, high, low, close, volume, adjusted\_close}\]

where \(d\) is a floating poing representation of date, as returned by date2num, and \(\text{date}\) is a python standard library datetime.date instance.

The name of this kwarg is a historical artifact. Formerly, True returned a cbook Bunch holding 1-D ndarrays. The behavior of a numpy recarray is very similar to the Bunch.

```python
matplotlib.finance.plot_day_summary2_ochl(ax, opens, closes, highs, lows, ticksize=4, colorup='k', colordown='r')
```

Represent the time, open, close, high, low, as a vertical line ranging from low to high. The left tick is the open and the right tick is the close.

**Parameters**

**ax** : Axes

an Axes instance to plot to

**opens** : sequence

sequence of opening values

**closes** : sequence

sequence of closing values

**highs** : sequence

sequence of high values

**lows** : sequence

sequence of low values

**ticksize** : int

size of open and close ticks in points

**colorup** : color

the color of the lines where close >= open

**colordown** : color
the color of the lines where close < open

**Returns ret :** list

a list of lines added to the axes

matplotlib.finance.plot_day_summary2_ohlc(ax, opens, highs, lows, closes, ticksize=4, colorup='k', colordown='r')

Represent the time, open, high, low, close as a vertical line ranging from low to high. The left tick is the open and the right tick is the close. opens, highs, lows and closes must have the same length. NOTE: this code assumes if any value open, high, low, close is missing (-1) they all are missing

**Parameters ax :** Axes

an Axes instance to plot to

**opens :** sequence

sequence of opening values

**highs :** sequence

sequence of high values

**lows :** sequence

sequence of low values

**closes :** sequence

sequence of closing values

**ticksize :** int

size of open and close ticks in points

**colorup :** color

the color of the lines where close >= open

**colordown :** color

the color of the lines where close < open

**Returns ret :** list

a list of lines added to the axes

matplotlib.finance.plot_day_summary_oclh(ax, quotes, ticksize=3, colorup='k', colordown='r')

Plots day summary

Represent the time, open, close, high, low as a vertical line ranging from low to high. The left tick is the open and the right tick is the close.

**Parameters ax :** Axes

an Axes instance to plot to

**quotes :** sequence of (time, open, close, high, low, ...) sequences
data to plot. time must be in float date format - see date2num

ticksiz e: int
open/close tick marker in points
colorup : color
the color of the lines where close >= open
colordown : color
the color of the lines where close < open

Returns lines : list
list of tuples of the lines added (one tuple per quote)

matplotlib.finance.plot_day_summary_ohlc(ax, quotes, ticksize=3, colorup='k', colordown='r')

Plots day summary
Represent the time, open, high, low, close as a vertical line ranging from low to high. The left tick is the open and the right tick is the close.

Parameters ax : Axes
an Axes instance to plot to
quotes : sequence of (time, open, high, low, close, ...) sequences
data to plot. time must be in float date format - see date2num
ticksiz e : int
open/close tick marker in points
colorup : color
the color of the lines where close >= open
colordown : color
the color of the lines where close < open

Returns lines : list
list of tuples of the lines added (one tuple per quote)

matplotlib.finance.quotes_historical_yahoo_ochl(ticker, date1, date2, asobject=False, 
adjusted=True, cachename=None)

Get historical data for ticker between date1 and date2.
See parse_yahoo_historical() for explanation of output formats and the asobject and adjusted kwargs.

Parameters ticker : str
stock ticker
**date1**: sequence of form (year, month, day), `datetime`, or `date`
start date

**date2**: sequence of form (year, month, day), `datetime`, or `date`
end date

**cachename**: `str` or `None`
is the name of the local file cache. If `None`, will default to the md5 hash or the url (which incorporates the ticker and date range)

### Examples

```python
>>> sp = f.quotes_historical_yahoo_ochl('^GSPC', d1, d2,
   asobject=True, adjusted=True)
>>> returns = (sp.open[1:] - sp.open[:-1])/sp.open[1:]
>>> [n,bins,patches] = hist(returns, 100)
>>> mu = mean(returns)
>>> sigma = std(returns)
>>> x = normpdf(bins, mu, sigma)
>>> plot(bins, x, color='red', lw=2)
```

`matplotlib.finance.quotes_historical_yahoo_ohlc(ticker, date1, date2, asobject=False, adjusted=True, cachename=None)`

Get historical data for ticker between `date1` and `date2`.

See `parse_yahoo_historical()` for explanation of output formats and the `asobject` and `adjusted` kwargs.

**Parameters**

**ticker**: `str`
stock ticker

**date1**: sequence of form (year, month, day), `datetime`, or `date`
start date

**date2**: sequence of form (year, month, day), `datetime`, or `date`
end date

**cachename**: `str` or `None`
is the name of the local file cache. If `None`, will default to the md5 hash or the url (which incorporates the ticker and date range)

### Examples

```python
>>> sp = f.quotes_historical_yahoo_ochl('^GSPC', d1, d2,
   asobject=True, adjusted=True)
>>> returns = (sp.open[1:] - sp.open[:-1])/sp.open[1:]
>>> [n,bins,patches] = hist(returns, 100)
>>> mu = mean(returns)
>>> sigma = std(returns)
>>> x = normpdf(bins, mu, sigma)
>>> plot(bins, x, color='red', lw=2)
```
```python
>>> mu = mean(returns)
>>> sigma = std(returns)
>>> x = normpdf(bins, mu, sigma)
>>> plot(bins, x, color='red', lw=2)
```

```python
matplotlib.finance.volume_overlay(ax, opens, closes, volumes, colorup='k', colordown='r', width=4, alpha=1.0)
```

Add a volume overlay to the current axes. The opens and closes are used to determine the color of the bar. -1 is missing. If a value is missing on one it must be missing on all

**Parameters**

- **ax**: Axes
  - an Axes instance to plot to
- **opens**: sequence
  - a sequence of opens
- **closes**: sequence
  - a sequence of closes
- **volumes**: sequence
  - a sequence of volumes
- **width**: int
  - the bar width in points
- **colorup**: color
  - the color of the lines where close >= open
- **colordown**: color
  - the color of the lines where close < open
- **alpha**: float
  - bar transparency

**Returns**

- **ret**: barCollection
  - The barCollection added to the axes

```python
matplotlib.finance.volume_overlay2(ax, closes, volumes, colorup='k', colordown='r', width=4, alpha=1.0)
```

Add a volume overlay to the current axes. The closes are used to determine the color of the bar. -1 is missing. If a value is missing on one it must be missing on all

nb: first point is not displayed - it is used only for choosing the right color

**Parameters**

- **ax**: Axes
  - an Axes instance to plot to
- **closes**: sequence
  - a sequence of closes
`volumes` : sequence
    a sequence of volumes

`width` : int
    the bar width in points

`colorup` : color
    the color of the lines where close >= open

`colordown` : color
    the color of the lines where close < open

`alpha` : float
    bar transparency

**Returns** `ret` : `barCollection`

The `barCollection` added to the axes

```python
matplotlib.finance.volume_overlay3(ax, quotes, colorup='k', colordown='r', width=4, alpha=1.0)
```

Add a volume overlay to the current axes. quotes is a list of (d, open, high, low, close, volume) and close-open is used to determine the color of the bar

**Parameters**
- `ax` : Axes
  an Axes instance to plot to
- `quotes` : sequence of (time, open, high, low, close, ...) sequences
  data to plot. time must be in float date format - see date2num
- `width` : int
  the bar width in points
- `colorup` : color
  the color of the lines where close1 >= close0
- `colordown` : color
  the color of the lines where close1 < close0
- `alpha` : float
  bar transparency

**Returns** `ret` : `barCollection`

The `barCollection` added to the axes

31.1. `matplotlib.finance`
32.1 matplotlib.font_manager

A module for finding, managing, and using fonts across platforms.

This module provides a single FontManager instance that can be shared across backends and platforms. The findfont() function returns the best TrueType (TTF) font file in the local or system font path that matches the specified FontProperties instance. The FontManager also handles Adobe Font Metrics (AFM) font files for use by the PostScript backend.

The design is based on the W3C Cascading Style Sheet, Level 1 (CSS1) font specification. Future versions may implement the Level 2 or 2.1 specifications.

Experimental support is included for using fontconfig on Unix variant platforms (Linux, OS X, Solaris). To enable it, set the constant USE_FONTCONFIG in this file to True. Fontconfig has the advantage that it is the standard way to look up fonts on X11 platforms, so if a font is installed, it is much more likely to be found.

class matplotlib.font_manager.FontEntry(fname='', name='', style='normal', variant='normal', weight='normal', stretch='normal', size='medium')

Bases: object

A class for storing Font properties. It is used when populating the font lookup dictionary.

class matplotlib.font_manager.FontManager(size=None, weight='normal')

Bases: object

On import, the FontManager singleton instance creates a list of TrueType fonts based on the font properties: name, style, variant, weight, stretch, and size. The findfont() method does a nearest neighbor search to find the font that most closely matches the specification. If no good enough match is found, a default font is returned.

findfont(prop, fontext='ttf', directory=None, fallback_to_default=True, rebuild_if_missing=True)

Search the font list for the font that most closely matches the FontProperties prop.

findfont() performs a nearest neighbor search. Each font is given a similarity score to the target font properties. The first font with the highest score is returned. If no matches below a certain threshold are found, the default font (usually DejaVu Sans) is returned.
directory, is specified, will only return fonts from the given directory (or subdirectory of that directory).

The result is cached, so subsequent lookups don’t have to perform the O(n) nearest neighbor search.

If `fallback_to_default` is True, will fallback to the default font family (usually “DejaVu Sans” or “Helvetica”) if the first lookup hard-fails.

See the W3C Cascading Style Sheet, Level 1 documentation for a description of the font finding algorithm.

**static get_default_size()**

Return the default font size.

**get_default_weight()**

Return the default font weight.

**score_family(families, family2)**

Returns a match score between the list of font families in `families` and the font family name `family2`.

An exact match at the head of the list returns 0.0.

A match further down the list will return between 0 and 1.

No match will return 1.0.

**score_size(size1, size2)**

Returns a match score between `size1` and `size2`.

If `size2` (the size specified in the font file) is ‘scalable’, this function always returns 0.0, since any font size can be generated.

Otherwise, the result is the absolute distance between `size1` and `size2`, normalized so that the usual range of font sizes (6pt - 72pt) will lie between 0.0 and 1.0.

**score_stretch(stretch1, stretch2)**

Returns a match score between `stretch1` and `stretch2`.

The result is the absolute value of the difference between the CSS numeric values of `stretch1` and `stretch2`, normalized between 0.0 and 1.0.

**score_style(style1, style2)**

Returns a match score between `style1` and `style2`.

An exact match returns 0.0.

A match between ‘italic’ and ‘oblique’ returns 0.1.

No match returns 1.0.

**score_variant(variant1, variant2)**

Returns a match score between `variant1` and `variant2`.

An exact match returns 0.0, otherwise 1.0.
score_weight\((weight1, weight2)\)

Returns a match score between \(weight1\) and \(weight2\).

The result is the absolute value of the difference between the CSS numeric values of \(weight1\) and \(weight2\), normalized between 0.0 and 1.0.

set_default_weight\((weight)\)

Set the default font weight. The initial value is ‘normal’.

update_fonts\((filenames)\)

Update the font dictionary with new font files. Currently not implemented.

class matplotlib.font_manager.FontProperties\((family=None, style=None, variant=None, weight=None, stretch=None, size=None, fname=None, _init=None)\)

Bases: object

A class for storing and manipulating font properties.

The font properties are those described in the W3C Cascading Style Sheet, Level 1 font specification. The six properties are:

- **family**: A list of font names in decreasing order of priority. The items may include a generic font family name, either ‘serif’, ‘sans-serif’, ‘cursive’, ‘fantasy’, or ‘monospace’. In that case, the actual font to be used will be looked up from the associated rcParam in matplotlibrc.
- **style**: Either ‘normal’, ‘italic’ or ‘oblique’.
- **variant**: Either ‘normal’ or ‘small-caps’.

The default font property for TrueType fonts (as specified in the default matplotlibrc file) is:

| sans-serif, normal, normal, normal, normal, scalable. |

Alternatively, a font may be specified using an absolute path to a .ttf file, by using the `fname` kwarg.

The preferred usage of font sizes is to use the relative values, e.g., ‘large’, instead of absolute font sizes, e.g., 12. This approach allows all text sizes to be made larger or smaller based on the font manager’s default font size.

This class will also accept a fontconfig pattern, if it is the only argument provided. See the documentation on fontconfig patterns. This support does not require fontconfig to be installed. We are merely borrowing its pattern syntax for use here.
Note that matplotlib’s internal font manager and fontconfig use a different algorithm to lookup fonts, so the results of the same pattern may be different in matplotlib than in other applications that use fontconfig.

copy()
Return a deep copy of self

get_family()
Return a list of font names that comprise the font family.

get_file()
Return the filename of the associated font.

g_get_fontconfig_pattern()
Get a fontconfig pattern suitable for looking up the font as specified with fontconfig’s fc-match utility.

See the documentation on fontconfig patterns.

This support does not require fontconfig to be installed or support for it to be enabled. We are merely borrowing its pattern syntax for use here.

g_get_name()
Return the name of the font that best matches the font properties.

g_get_size()
Return the font size.

g_get_size_in_points()

g_get_slant()
Return the font style. Values are: ‘normal’, ‘italic’ or ‘oblique’.

g_get_stretch()

g_get_style()
Return the font style. Values are: ‘normal’, ‘italic’ or ‘oblique’.

g_get_variant()
Return the font variant. Values are: ‘normal’ or ‘small-caps’.

g_get_weight()

set_family(family)
Change the font family. May be either an alias (generic name is CSS parlance), such as: ‘serif’, ‘sans-serif’, ‘cursive’, ‘fantasy’, or ‘monospace’, a real font name or a list of real font names. Real font names are not supported when text.usetex is True.
**set_file(file)**
Set the filename of the fontfile to use. In this case, all other properties will be ignored.

**set_fontconfig_pattern(pattern)**
Set the properties by parsing a fontconfig pattern.

See the documentation on fontconfig patterns.

This support does not require fontconfig to be installed or support for it to be enabled. We are merely borrowing its pattern syntax for use here.

**set_name(family)**
Change the font family. May be either an alias (generic name is CSS parlance), such as: ‘serif’, ‘sans-serif’, ‘cursive’, ‘fantasy’, or ‘monospace’, a real font name or a list of real font names. Real font names are not supported when text.usetex is True.

**set_size(size)**

**set_slant(style)**
Set the font style. Values are: ‘normal’, ‘italic’ or ‘oblique’.

**set_stretch(stretch)**

**set_style(style)**
Set the font style. Values are: ‘normal’, ‘italic’ or ‘oblique’.

**set_variant(variant)**
Set the font variant. Values are: ‘normal’ or ‘small-caps’.

**set_weight(weight)**

**matplotlib.font_manager.OSXInstalledFonts(directions=None, fontext=’ttf’)**
Get list of font files on OS X - ignores font suffix by default.

**class matplotlib.font_manager.TempCache**
Bases: object

A class to store temporary caches that are (a) not saved to disk and (b) invalidated whenever certain font-related rcParams—namely the family lookup lists—are changed or the font cache is reloaded. This avoids the expensive linear search through all fonts every time a font is looked up.

**get(prop)**

make_rcparams_key()

set(prop, value)

matplotlib.font_manager.afmFontProperty(fontpath, font)
A function for populating a FontKey instance by extracting information from the AFM font file.
font is a class: AFM instance.

matplotlib.font_manager.createFontList(fontfiles, fontext='ttf')
A function to create a font lookup list. The default is to create a list of TrueType fonts. An AFM font list can optionally be created.

matplotlib.font_manager.findSystemFonts(fontpaths=None, fontext='ttf')
Search for fonts in the specified font paths. If no paths are given, will use a standard set of system paths, as well as the list of fonts tracked by fontconfig if fontconfig is installed and available. A list of TrueType fonts are returned by default with AFM fonts as an option.

matplotlib.font_manager.findfont(prop, **kw)

matplotlib.font_manager.get_fontconfig_fonts(fontext='ttf')
List the font filenames known to fc-list having the given extension.

matplotlib.font_manager.get_fontext_synonyms(fontext)
Return a list of file extensions that are synonyms for the given file extension fileext.

matplotlib.font_manager.is_opentype_cff_font(filename)
Returns True if the given font is a Postscript Compact Font Format Font embedded in an OpenType wrapper. Used by the PostScript and PDF backends that can not subset these fonts.

matplotlib.font_manager.list_fonts(directory, extensions)
Return a list of all fonts matching any of the extensions, possibly upper-cased, found recursively under the directory.

matplotlib.font_manager.pickle_dump(data, filename)
Equivalent to pickle.dump(data, open(filename, 'w')) but closes the file to prevent filehandle leakage.

matplotlib.font_manager.pickle_load(filename)
Equivalent to pickle.load(open(filename, 'r')) but closes the file to prevent filehandle leakage.

matplotlib.font_manager.ttfFontProperty(font)
A function for populating the FontKey by extracting information from the TrueType font file.
font is a FT2Font instance.

matplotlib.font_manager.ttfdict_to_fnames(d)
flatten a ttfdict to all the filenames it contains

matplotlib.font_manager.weight_as_number(weight)
Return the weight property as a numeric value. String values are converted to their corresponding numeric value.
matplotlib.font_manager.win32FontDirectory()

Return the user-specified font directory for Win32. This is looked up from the registry key:

```
\HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Explorer\Shell Folders\Fonts
```

If the key is not found, $WINDIR/Fonts will be returned.

matplotlib.font_manager.win32InstalledFonts(directory=None, fontext='ttf')

Search for fonts in the specified font directory, or use the system directories if none given. A list of TrueType font filenames are returned by default, or AFM fonts if fontext == ‘afm’.

### 32.2 matplotlib.fontconfig_pattern

A module for parsing and generating fontconfig patterns.

See the fontconfig pattern specification for more information.

class matplotlib.fontconfig_pattern.FontconfigPatternParser

Bases: object

A simple pyparsing-based parser for fontconfig-style patterns.

See the fontconfig pattern specification for more information.

parse(pattern)

Parse the given fontconfig pattern and return a dictionary of key/value pairs useful for initializing a font_manager.FontProperties object.

matplotlib.fontconfig_pattern.family_escape()

Return the string obtained by replacing the leftmost non-overlapping occurrences of pattern in string by the replacement repl.

matplotlib.fontconfig_pattern.family_unescape()

Return the string obtained by replacing the leftmost non-overlapping occurrences of pattern in string by the replacement repl.

matplotlib.fontconfig_pattern.generate_fontconfig_pattern(d)

Given a dictionary of key/value pairs, generates a fontconfig pattern string.

matplotlib.fontconfig_pattern.value_escape()

Return the string obtained by replacing the leftmost non-overlapping occurrences of pattern in string by the replacement repl.

matplotlib.fontconfig_pattern.value_unescape()

Return the string obtained by replacing the leftmost non-overlapping occurrences of pattern in string by the replacement repl.
33.1 matplotlib.gridspec

`gridspec` is a module which specifies the location of the subplot in the figure.

- **GridSpec** specifies the geometry of the grid that a subplot will be placed. The number of rows and number of columns of the grid need to be set. Optionally, the subplot layout parameters (e.g., left, right, etc.) can be tuned.

- **SubplotSpec** specifies the location of the subplot in the given `GridSpec`.

```python
class matplotlib.gridspec.GridSpec(nrows, ncols, left=None, bottom=None, right=None, top=None, wspace=None, hspace=None, width_ratios=None, height_ratios=None)
```

Bases: `matplotlib.gridspec.GridSpecBase`

A class that specifies the geometry of the grid that a subplot will be placed. The location of grid is determined by similar way as the SubplotParams.

The number of rows and number of columns of the grid need to be set. Optionally, the subplot layout parameters (e.g., left, right, etc.) can be tuned.

- **get_subplot_params**(fig=None)
  return a dictionary of subplot layout parameters. The default parameters are from rcParams unless a figure attribute is set.

- **locally_modified_subplot_params**()

- **tight_layout**(fig, renderer=None, pad=1.08, h_pad=None, w_pad=None, rect=None)
  Adjust subplot parameters to give specified padding.

  Parameters:

  - **pad** [float] padding between the figure edge and the edges of subplots, as a fraction of the fontsize.

  - **h_pad**, **w_pad** [float] padding (height/width) between edges of adjacent subplots. Defaults to `pad_inches`.
rect [if rect is given, it is interpreted as a rectangle] (left, bottom, right, top) in the normalized figure coordinate that the whole subplots area (including labels) will fit into. Default is (0, 0, 1, 1).

update(**kwargs)
    Update the current values. If any kwarg is None, default to the current value, if set, otherwise to rc.

class matplotlib.gridspec.GridSpecBase(nrows, ncols, height_ratios=None, width_ratios=None)
Bases: object
A base class of GridSpec that specifies the geometry of the grid that a subplot will be placed.
The number of rows and number of columns of the grid need to be set. Optionally, the ratio of heights and widths of rows and columns can be specified.

get_geometry()
    get the geometry of the grid, e.g., 2,3

get_grid_positions(fig)
    return lists of bottom and top position of rows, left and right positions of columns.

get_height_ratios()

get_subplot_params(fig=None)

get_width_ratios()

new_subplotspec(loc, rowspan=1, colspan=1)
    create and return a SubplotSpec instance.

set_height_ratios(height_ratios)

set_width_ratios(width_ratios)

class matplotlib.gridspec.GridSpecFromSubplotSpec(nrows, ncols, subplot_spec, subplot_spec, wspace=None, hspace=None, height_ratios=None, width_ratios=None)
Bases: matplotlib.gridspec.GridSpecBase
GridSpec whose subplot layout parameters are inherited from the location specified by a given SubplotSpec.
The number of rows and number of columns of the grid need to be set. An instance of SubplotSpec is also needed to be set from which the layout parameters will be inherited. The wspace and hspace of the layout can be optionally specified or the default values (from the figure or rcParams) will be used.
get_subplot_params(fig=None)
    return a dictionary of subplot layout parameters.

get_topmost_subplotspec()
    get the topmost SubplotSpec instance associated with the subplot

class matplotlib.gridspec.SubplotSpec(gridspec, num1, num2=None)
    Bases: object
    specifies the location of the subplot in the given GridSpec.
    The subplot will occupy the num1-th cell of the given gridspec. If num2 is provided, the subplot will span between num1-th cell and num2-th cell.
    The index starts from 0.

get_geometry()
    get the subplot geometry, e.g., 2,2,3. Unlike SubplotParams, index is 0-based

get_gridspec()

get_position(fig, return_all=False)
    update the subplot position from fig.subplotpars

get_topmost_subplotspec()
    get the topmost SubplotSpec instance associated with the subplot
34.1 matplotlib.image

The image module supports basic image loading, rescaling and display operations.

```python
class matplotlib.image.AxesImage(ax, cmap=None, norm=None, interpolation=None, origin=None, extent=None, filternorm=1, filterrad=4.0, resample=False, **kwargs)
```
Bases: matplotlib.image._ImageBase

Interpolation and cmap default to their rc settings.

cmap is a colors.Colormap instance norm is a colors.Normalize instance to map luminance to 0-1

Extent is data axes (left, right, bottom, top) for making image plots registered with data plots. Default is to label the pixel centers with the zero-based row and column indices.

Additional kwargs are matplotlib.artist properties

```python
def get_cursor_data(event)
    Get the cursor data for a given event

def get_extent()
    Get the image extent: left, right, bottom, top

def get_window_extent(renderer=None)

make_image(renderer, magnification=1.0, unsampled=False)
```

```python
def set_extent(extent)
    extent is data axes (left, right, bottom, top) for making image plots
    This updates ax.dataLim, and, if autoscaling, sets viewLim to tightly fit the image, regardless of dataLim. Autoscaling state is not changed, so following this with ax.autoscale_view will redo the autoscaling in accord with dataLim.
```

```python
class matplotlib.image.BboxImage(bbox, cmap=None, norm=None, interpolation=None, origin=None, filternorm=1, filterrad=4.0, resample=False, interp_at_native=True, **kwargs)
```
Bases: matplotlib.image._ImageBase
The Image class whose size is determined by the given bbox.

cmap is a colors.Colormap instance norm is a colors.Normalize instance to map luminance to 0-1

interp_at_native is a flag that determines whether or not interpolation should still be applied when
the image is displayed at its native resolution. A common use case for this is when displaying an
image for annotational purposes; it is treated similarly to Photoshop (interpolation is only used when
displaying the image at non-native resolutions).

kwargs are an optional list of Artist keyword args

contains(mouseevent)
    Test whether the mouse event occurred within the image.

get_transform()

get_window_extent(renderer=None)

make_image(renderer, magnification=1.0, unsampled=False)

class matplotlib.image.FigureImage(fig, cmap=None, norm=None, offsetx=0, offsety=0, origin=None, **kwargs)
    Bases: matplotlib.image._ImageBase
    cmap is a colors.Colormap instance norm is a colors.Normalize instance to map luminance to 0-1
    kwargs are an optional list of Artist keyword args

    get_extent()
        Get the image extent: left, right, bottom, top

    make_image(renderer, magnification=1.0, unsampled=False)

    set_data(A)
        Set the image array.

    zorder = 0

class matplotlib.image.NonUniformImage(ax, **kwargs)
    Bases: matplotlib.image.AxesImage
    kwargs are identical to those for AxesImage, except that ‘nearest’ and ‘bilinear’ are the only supported
    ‘interpolation’ options.

    get_extent()

    make_image(renderer, magnification=1.0, unsampled=False)

    set_array(*args)
set_cmap(cmap)

set_data(x, y, A)
Set the grid for the pixel centers, and the pixel values.

\[ x \text{ and } y \text{ are monotonic 1-D ndarrays of lengths } N \text{ and } M, \text{ respectively, specifying pixel centers} \]

\[ A \text{ is an (M,N) ndarray or masked array of values to be colormapped, or a (M,N,3) RGB array, or a (M,N,4) RGBA array.} \]

set_filternorm(s)

set_filterrad(s)

set_interpolation(s)

set_norm(norm)

class matplotlib.image.PcolorImage(ax, x=None, y=None, A=None, cmap=None, norm=None, **kwargs)

Bases: matplotlib.image.AxesImage
Make a pcolor-style plot with an irregular rectangular grid.
This uses a variation of the original irregular image code, and it is used by pcolorfast for the corresponding grid type.
cmap defaults to its rc setting
cmap is a colors.Colormap instance norm is a colors.Normalize instance to map luminance to 0-1
Additional kwargs are matplotlib.artist properties
get_cursor_data(event)
   Get the cursor data for a given event
make_image(renderer, magnification=1.0, unsampled=False)

set_array(*args)

set_data(x, y, A)
Set the grid for the rectangle boundaries, and the data values.

\[ x \text{ and } y \text{ are monotonic 1-D ndarrays of lengths } N+1 \text{ and } M+1, \text{ respectively, specifying rectangle boundaries. If None, they will be created as uniform arrays from 0 through } N \text{ and 0 through } M, \text{ respectively.} \]

\[ A \text{ is an (M,N) ndarray or masked array of values to be colormapped, or a (M,N,3) RGB array, or a (M,N,4) RGBA array.} \]
Composite a number of RGBA images into one. The images are composited in the order in which they appear in the images list.

**Parameters**

- **images**: list of Images
  - Each must have a `make_image` method. For each image, `can_composite` should return `True`, though this is not enforced by this function. Each image must have a purely affine transformation with no shear.

- **renderer**: RendererBase instance

- **magnification**: float
  - The additional magnification to apply for the renderer in use.

**Returns**

- **tuple**: image, offset_x, offset_y
  - Returns the tuple:
    - `image`: A numpy array of the same type as the input images.
    - `offset_x`, `offset_y`: The offset of the image (left, bottom) in the output figure.

---

Read an image from a file into an array.

- **fname**: A string containing a path to a filename, or a Python file-like object. If using a file object, it must be opened in binary mode.

- **format**: None
  - If `format` is provided, will try to read file of that type, otherwise the format is deduced from the filename. If nothing can be deduced, PNG is tried.

Return value is a `numpy.array`. For grayscale images, the return array is MxN. For RGB images, the return value is MxNx3. For RGBA images the return value is MxNx4.

Matplotlib can only read PNGs natively, but if PIL is installed, it will use it to load the image and return an array (if possible) which can be used with `imshow()`. Note, URL strings may not be compatible with PIL. Check the PIL documentation for more information.

Save an array as in image file.

- **fname**: A string containing a path to a filename, or a Python file-like object. If `format` is `None` and `fname` is a string, the output format is deduced from the extension of the filename.

- **arr**: An MxN (luminance), MxNx3 (RGB) or MxNx4 (RGBA) array.

**Keyword arguments:**
**vmin/vmax:** [None | scalar] `vmin` and `vmax` set the color scaling for the image by fixing the values that map to the colormap color limits. If either `vmin` or `vmax` is None, that limit is determined from the `arr min/max` value.

**cmap:** `cmap` is a `colors.Colormap` instance, e.g., `cm.jet`. If None, default to the rc image.cmap value.

**format:** One of the file extensions supported by the active backend. Most backends support png, pdf, ps, eps and svg.

**origin** [‘upper’ | ‘lower’] Indicates where the [0,0] index of the array is in the upper left or lower left corner of the axes. Defaults to the rc image.origin value.

**dpi** The DPI to store in the metadata of the file. This does not affect the resolution of the output image.

```python
cmatplotlib.image.pil_to_array(pilImage)
```
Load a PIL image and return it as a numpy array. For grayscale images, the return array is MxN. For RGB images, the return value is MxNx3. For RGBA images the return value is MxNx4.

```python
cmatplotlib.image.thumbnail(infile, thumbfile, scale=0.1, interpolation=’bilinear’, preview=False)
```
make a thumbnail of image in `infile` with output filename `thumbfile`.

- `infile` the image file – must be PNG or Pillow-readable if you have Pillow installed
- `thumbfile` the thumbnail filename
- `scale` the scale factor for the thumbnail
- `interpolation` the interpolation scheme used in the resampling
- `preview` if True, the default backend (presumably a user interface backend) will be used which will cause a figure to be raised if `show()` is called. If it is False, a pure image backend will be used depending on the extension, ‘png’->FigureCanvasAgg, ‘pdf’->FigureCanvasPdf, ‘svg’->FigureCanvasSVG

See examples/misc/image_thumbnail.py.

Return value is the figure instance containing the thumbnail.
LEGEND AND LEGEND_HANDLER

35.1 matplotlib.legend

The legend module defines the Legend class, which is responsible for drawing legends associated with axes and/or figures.

**Important:** It is unlikely that you would ever create a Legend instance manually. Most users would normally create a legend via the `legend()` function. For more details on legends there is also a `legend guide`.

The Legend class can be considered as a container of legend handles and legend texts. Creation of corresponding legend handles from the plot elements in the axes or figures (e.g., lines, patches, etc.) are specified by the handler map, which defines the mapping between the plot elements and the legend handlers to be used (the default legend handlers are defined in the `legend_handler` module). Note that not all kinds of artist are supported by the legend yet by default but it is possible to extend the legend handler’s capabilities to support arbitrary objects. See the `legend guide` for more information.

```python
class matplotlib.legend.DraggableLegend(legend, use_blit=False, update='loc')
    Bases: matplotlib.offsetbox.DraggableOffsetBox

    update [If “loc”, update loc parameter of] legend upon finalizing. If “bbox”, update bbox_to_anchor parameter.

    artist_picker(legend, evt)

    finalize_offset()
```

```python
class matplotlib.legend.Legend(parent, handles, labels, loc=None, numpoints=None, markerscale=None, markerfirst=True, scatterpoints=None, scatteryoffsets=None, prop=None, fontsize=None, borderpad=None, labelsizing=None, handlelength=None, handleheight=None, handletextpad=None, borderaxespad=None, columnspace=None, ncol=1, mode=None, fancybox=None, shadow=None, title=None, facecolor=None, edgecolor=None, bbox_to_anchor=None, bbox_transform=None, frameon=None, handler_map=None)
```
Bases: `matplotlib.artist.Artist`

Place a legend on the axes at location `loc`. Labels are a sequence of strings and `loc` can be a string or an integer specifying the legend location.

The location codes are:

```
'best'    : 0, (only implemented for axes legends)
'upper right' : 1,
'upper left'  : 2,
'lower left'  : 3,
'lower right' : 4,
'right'      : 5,
'center left' : 6,
'center right': 7,
'lower center': 8,
'upper center': 9,
'center'     : 10,
```

`loc` can be a tuple of the normalized coordinate values with respect its parent.

- `parent`: the artist that contains the legend
- `handles`: a list of artists (lines, patches) to be added to the legend
- `labels`: a list of strings to label the legend

Optional keyword arguments:
<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>loc</td>
<td>Location code string, or tuple (see below).</td>
</tr>
<tr>
<td>prop</td>
<td>the font property</td>
</tr>
<tr>
<td>fontsize</td>
<td>the font size (used only if prop is not specified)</td>
</tr>
<tr>
<td>markerscale</td>
<td>the relative size of legend markers vs. original</td>
</tr>
<tr>
<td>markerfirst</td>
<td>If True (default), marker is to left of the label.</td>
</tr>
<tr>
<td>numpoints</td>
<td>the number of points in the legend for line</td>
</tr>
<tr>
<td>scatterpoints</td>
<td>the number of points in the legend for scatter plot</td>
</tr>
<tr>
<td>scatteryoffsets</td>
<td>a list of yoffsets for scatter symbols in legend</td>
</tr>
<tr>
<td>frameon</td>
<td>If True, draw the legend on a patch (frame).</td>
</tr>
<tr>
<td>fancybox</td>
<td>If True, draw the frame with a round fancybox.</td>
</tr>
<tr>
<td>shadow</td>
<td>If True, draw a shadow behind legend.</td>
</tr>
<tr>
<td>framealpha</td>
<td>Transparency of the frame.</td>
</tr>
<tr>
<td>edgecolor</td>
<td>Frame edgecolor.</td>
</tr>
<tr>
<td>facecolor</td>
<td>Frame facecolor.</td>
</tr>
<tr>
<td>ncol</td>
<td>number of columns</td>
</tr>
<tr>
<td>borderpad</td>
<td>the fractional whitespace inside the legend border</td>
</tr>
<tr>
<td>labelspacing</td>
<td>the vertical space between the legend entries</td>
</tr>
<tr>
<td>handlelength</td>
<td>the length of the legend handles</td>
</tr>
<tr>
<td>handleheight</td>
<td>the height of the legend handles</td>
</tr>
<tr>
<td>handletextpad</td>
<td>the pad between the legend handle and text</td>
</tr>
<tr>
<td>borderaxespad</td>
<td>the pad between the axes and legend border</td>
</tr>
<tr>
<td>columnsspacing</td>
<td>the spacing between columns</td>
</tr>
<tr>
<td>title</td>
<td>the legend title</td>
</tr>
<tr>
<td>bbox_to_anchor</td>
<td>the bbox that the legend will be anchored.</td>
</tr>
<tr>
<td>bbox_transform</td>
<td>the transform for the bbox. transAxes if None.</td>
</tr>
</tbody>
</table>

The pad and spacing parameters are measured in font-size units. e.g., a fontsize of 10 points and a handlelength=5 implies a handlelength of 50 points. Values from rcParams will be used if None.

Users can specify any arbitrary location for the legend using the `bbox_to_anchor` keyword argument. `bbox_to_anchor` can be an instance of BboxBase(or its derivatives) or a tuple of 2 or 4 floats. See `set_bbox_to_anchor()` for more detail.

The legend location can be specified by setting `loc` with a tuple of 2 floats, which is interpreted as the lower-left corner of the legend in the normalized axes coordinate.

```python
codes = {'best': 0, 'upper right': 1, 'upper left': 2, 'lower left': 3, 'lower right': 4, 'right': 5, 'center left': 6, 'center right': 7, 'center': 10}
contains(event)
```
If draggable is on, you can drag the legend on the canvas with the mouse. The DraggableLegend helper instance is returned if draggable is on.

The update parameter control which parameter of the legend changes when dragged. If update is “loc”, the loc parameter of the legend is changed. If “bbox”, the bbox_to_anchor parameter is changed.

\[\text{draw}(\text{artist}, \text{renderer}, *\text{args}, **\text{kwargs})\]
\[\text{Draw everything that belongs to the legend}\]

\[\text{draw_frame}(b)\]
\[\text{b is a boolean. Set draw frame to b}\]

\[\text{get_bbox_to_anchor}()\]
\[\text{return the bbox that the legend will be anchored}\]

\[\text{get_children}()\]
\[\text{return a list of child artists}\]

\textit{classmethod get_default_handler_map()}
\[\text{A class method that returns the default handler map.}\]

\[\text{get_frame}()\]
\[\text{return the Rectangle instance used to frame the legend}\]

\[\text{get_frame_on}()\]
\[\text{Get whether the legend box patch is drawn}\]

\textit{static get_legend_handler(legend_handler_map, orig_handle)}
\[\text{return a legend handler from legend_handler_map that corresponds to orig_handler.}\]

\[\text{legend_handler_map} \text{ should be a dictionary object (that is returned by the get_legend_handler_map method).}\]

It first checks if the \texttt{orig_handle} itself is a key in the \texttt{legend_handler_map} and return the associated value. Otherwise, it checks for each of the classes in its method-resolution-order. If no matching key is found, it returns None.

\[\text{get_legend_handler_map}()\]
\[\text{return the handler map.}\]

\[\text{get_lines}()\]
\[\text{return a list of lines.Line2D instances in the legend}\]

\[\text{get_patches}()\]
\[\text{return a list of patch instances in the legend}\]

\[\text{get_texts}()\]
\[\text{return a list of text.Text instance in the legend}\]

\[\text{get_title}()\]
\[\text{return Text instance for the legend title}\]

\[\text{get_window_extent}(*\text{args}, **\text{kwargs})\]
\[\text{return a extent of the legend}\]
set_bbox_to_anchor(bbox, transform=None)
set the bbox that the legend will be anchored.

bbox can be a BBoxBase instance, a tuple of [left, bottom, width, height] in the given transform (normalized axes coordinate if None), or a tuple of [left, bottom] where the width and height will be assumed to be zero.

classmethod set_default_handler_map(handler_map)
A class method to set the default handler map.

set_frame_on(b)
Set whether the legend box patch is drawn

ACCEPTS: [ True | False ]

set_title(title, prop=None)
set the legend title. Fontproperties can be optionally set with prop parameter.

classmethod update_default_handler_map(handler_map)
A class method to update the default handler map.

zorder = 5

35.2 matplotlib.legend_handler

This module defines default legend handlers.

It is strongly encouraged to have read the legend guide before this documentation.

Legend handlers are expected to be a callable object with a following signature.

legend_handler(legend, orig_handle, fontsize, handlebox)

Where legend is the legend itself, orig_handle is the original plot, fontsize is the fontsize in pixels, and handlebox is a OffsetBox instance. Within the call, you should create relevant artists (using relevant properties from the legend and/or orig_handle) and add them into the handlebox. The artists needs to be scaled according to the fontsize (note that the size is in pixel, i.e., this is dpi-scaled value).

This module includes definition of several legend handler classes derived from the base class (HandlerBase) with the following method.

def legend_artist(self, legend, orig_handle, fontsize, handlebox):

class matplotlib.legend_handler.HandlerBase(xpad=0.0, ypad=0.0, update_func=None)
A Base class for default legend handlers.

The derived classes are meant to override create_artists method, which has a following signature.

def create_artists(self, legend, orig_handle, xdescent, ydescent, width, height, fontsize, trans):
The overridden method needs to create artists of the given transform that fits in the given dimension (xdescent, ydescent, width, height) that are scaled by fontsize if necessary.

adjust_drawing_area(legend, orig_handle, xdescent, ydescent, width, height, fontsize)

create_artists(legend, orig_handle, xdescent, ydescent, width, height, fontsize, trans)

legend_artist(legend, orig_handle, fontsize, handlebox)
   Return the artist that this HandlerBase generates for the given original artist/handle.

   Parameters
   legend : matplotlib.legend.Legend instance
      The legend for which these legend artists are being created.

   orig_handle : matplotlib.artist.Artist or similar
      The object for which these legend artists are being created.

   fontsize : float or int
      The fontsize in pixels. The artists being created should be scaled according to the given fontsize.

   handlebox : matplotlib.offsetbox.OffsetBox instance
      The box which has been created to hold this legend entry’s artists. Artists created in the legend_artist method must be added to this handlebox inside this method.

update_prop(legend_handle, orig_handle, legend)

class matplotlib.legend_handler.HandlerCircleCollection(yoffs=None, sizes=None, **kw)
   Handler for CircleCollections

   create_collection(orig_handle, sizes, offsets, transOffset)

class matplotlib.legend_handler.HandlerErrorbar(xerr_size=0.5, yerr_size=None, marker_pad=0.3, numpoints=None, **kw)
   Handler for Errorbars

   create_artists(legend, orig_handle, xdescent, ydescent, width, height, fontsize, trans)

   get_err_size(legend, xdescent, ydescent, width, height, fontsize)

class matplotlib.legend_handler.HandlerLine2D(marker_pad=0.3, numpoints=None, **kw)
   Handler for Line2D instances.

   create_artists(legend, orig_handle, xdescent, ydescent, width, height, fontsize, trans)
class matplotlib.legend_handler.HandlerLineCollection(marker_pad=0.3, numpoints=None, **kw)

Handler for LineCollection instances.

create_artists(legend, orig_handle, xdescent, ydescent, width, height, fontsize, trans)

get_numpoints(legend)

class matplotlib.legend_handler.HandlerNpoints(marker_pad=0.3, numpoints=None, **kw)

get_numpoints(legend)

get_xdata(legend, xdescent, ydescent, width, height, fontsize)

class matplotlib.legend_handler.HandlerNpointsYoffsets(numpoints=None, yoffsets=None, **kw)

get_ydata(legend, xdescent, ydescent, width, height, fontsize)

class matplotlib.legend_handler.HandlerPatch(patch_func=None, **kw)

Handler for Patch instances.

The HandlerPatch class optionally takes a function patch_func who’s responsibility is to create the legend key artist. The patch_func should have the signature:

```python
def patch_func(legend=legend, orig_handle=orig_handle, 
               xdescent=xdescent, ydescent=ydescent, 
               width=width, height=height, fontsize=fontsize)
```

Subsequently the created artist will have its update_prop method called and the appropriate transform will be applied.

create_artists(legend, orig_handle, xdescent, ydescent, width, height, fontsize, trans)

class matplotlib.legend_handler.HandlerPathCollection(yoffsets=None, sizes=None, **kw)

Handler for PathCollections, which are used by scatter

create_collection(orig_handle, sizes, offsets, transOffset)

class matplotlib.legend_handler.HandlerPolyCollection(xpad=0.0, ypad=0.0, update_func=None)

Handler for PolyCollection used in fill_between and stackplot.

create_artists(legend, orig_handle, xdescent, ydescent, width, height, fontsize, trans)
class matplotlib.legend_handler.HandlerRegularPolyCollection(yoffsets=None, sizes=None, **kw)

Handler for RegularPolyCollections.

create_artists(legend, orig_handle, xdescent, ydescent, width, height, fontsize, trans)

create_collection(orig_handle, sizes, offsets, transOffset)

get_numpoints(legend)

get_sizes(legend, orig_handle, xdescent, ydescent, width, height, fontsize)

update_prop(legend_handle, orig_handle, legend)

class matplotlib.legend_handler.HandlerStem(marker_pad=0.3, numpoints=None, bottom=None, yoffsets=None, **kw)

Handler for Errorbars

create_artists(legend, orig_handle, xdescent, ydescent, width, height, fontsize, trans)

get_ydata(legend, xdescent, ydescent, width, height, fontsize)

class matplotlib.legend_handler.HandlerTuple(**kwargs)

Handler for Tuple

create_artists(legend, orig_handle, xdescent, ydescent, width, height, fontsize, trans)

matplotlib.legend_handler.update_from_first_child(tgt, src)
36.1 matplotlib.lines

This module contains all the 2D line class which can draw with a variety of line styles, markers and colors.

```python
class matplotlib.lines.Line2D(xdata, ydata, linewidth=None, linestyle=None, color=None, marker=None, markersize=None, markeredgewidth=None, markeredgecolor=None, markerfacecolor=None, markerfacecoloralt='none', fillstyle=None, antialiased=None, dash_capstyle=None, solid_capstyle=None, dash_joinstyle=None, solid_joinstyle=None, pickradius=5, drawstyle=None, markevery=None, **kwargs)
```

Bases: `matplotlib.artist.Artist`

A line - the line can have both a solid linestyle connecting all the vertices, and a marker at each vertex. Additionally, the drawing of the solid line is influenced by the drawstyle, e.g., one can create “stepped” lines in various styles.

Create a `Line2D` instance with x and y data in sequences `xdata`, `ydata`.

The kwargs are `Line2D` properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[ (Path, Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>[‘butt’</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>[‘miter’</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>drawstyle</td>
<td>[‘default’</td>
</tr>
</tbody>
</table>
### Table 36.1 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>figure</td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>['full'</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with <code>%%s</code> conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['solid'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>marker</td>
<td>A valid marker style</td>
</tr>
<tr>
<td>markeredgecolor or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>[None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function <code>fn(artist, event)</code></td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a <code>matplotlib.transforms.Transform</code> instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

See `set_linestyle()` for a description of the line styles, `set_marker()` for a description of the markers, and `set_drawstyle()` for a description of the draw styles.

### axes

The `Axes` instance the artist resides in, or `None`.

### contains(mouseevent)

Test whether the mouse event occurred on the line. The pick radius determines the precision of the location test (usually within five points of the value). Use `get_pickradius()` or `set_pickradius()` to view or modify it.

Returns `True` if any values are within the radius along with `{"ind": pointlist}`, where `pointlist` is the set of points within the radius.

TODO: sort returned indices by distance

### draw(artist, renderer, *args, **kwargs)
draw the Line with renderer unless visibility is False

drawStyleKeys = ['default', 'steps-mid', 'steps-pre', 'steps-post', 'steps']

drawStyles = {'default': '_draw_lines', 'steps-mid': '_draw_steps_mid', 'steps-pre': '_draw_steps_pre', 'steps-post': '_draw_steps_post', 'steps': '_draw_steps_pre'}

fillStyles = ('full', 'left', 'right', 'bottom', 'top', 'none')

filled_markers = ('o', 'v', '^', '<', '>', '8', 's', 'p', '*', 'h', 'H', 'D', 'd', 'P', 'X')

get_aa()  
    alias for get_antialiased

g_get_antialiased()

get_c()  
    alias for get_color

g_get_color()

g_get_dash_capstyle()  
    Get the cap style for dashed linestyles

g_get_dash_joinstyle()  
    Get the join style for dashed linestyles

g_get_data(ori=True)  
    Return the xdata, ydata.  
    If ori is True, return the original data.

g_get_drawstyle()

g_get_fillstyle()  
    return the marker fillstyle

g_get_linestyle()

g_get_linewidth()

g_get_ls()  
    alias for get_linestyle

g_get_lw()  
    alias for get_linewidth

g_get_marker()
get_markeredgecolor()

get_markeredgewidth()

get_markerfacecolor()

get_markerfacecoloralt()

get_markersize()

get_markevery()
    return the markevery setting

get_mec()
    alias for get_markeredgecolor

get_mew()
    alias for get_markeredgewidth

get_mfc()
    alias for get_markerfacecolor

get_mfcalt(alt=False)
    alias for get_markerfacecoloralt

get_ms()
    alias for get_markersize

get_path()
    Return the Path object associated with this line.

get_pickradius()
    return the pick radius used for containment tests

get_solid_capstyle()
    Get the cap style for solid linestyles

get_solid_joinstyle()
    Get the join style for solid linestyles

get_window_extent(renderer)

get_xdata(orig=True)
    Return the xdata.
    If orig is True, return the original data, else the processed data.

get_xydata()
    Return the xy data as a Nx2 numpy array.
get_ydata\(\text{orig=True}\)

Return the ydata.

If \text{orig} is \text{True}, return the original data, else the processed data.

is_dashed()

return True if line is dashstyle

lineStyles = {‘-’: ‘_draw_solid’, ‘-.’: ‘_draw_dash_dot’, ‘.’: ‘_draw_dotted’, ‘None’: ‘_draw_nothing’}


recache\(\text{always=False}\)

recache_always()

set_aa\(val\)

alias for set_antialiased

set_antialiased\(b\)

True if line should be drawin with antialiased rendering

ACCEPTS: [True | False]

set_c\(val\)

alias for set_color

set_color\(color\)

Set the color of the line

ACCEPTS: any matplotlib color

set_dash_capstyle\(s\)

Set the cap style for dashed linestyles

ACCEPTS: [‘butt’ | ‘round’ | ‘projecting’]

set_dash_joinstyle\(s\)

Set the join style for dashed linestyles ACCEPTS: [‘miter’ | ‘round’ | ‘bevel’]

set_dashes\(seq\)

Set the dash sequence, sequence of dashes with on off ink in points. If seq is empty or if seq =
\(\text{None, None}\), the linestyle will be set to solid.

ACCEPTS: sequence of on/off ink in points

set_data\(*args\)

Set the x and y data

ACCEPTS: 2D array (rows are x, y) or two 1D arrays

set_drawstyle\(\text{drawstyle}\)

Set the drawstyle of the plot
‘default’ connects the points with lines. The steps variants produce step-plots. ‘steps’ is equivalent to ‘steps-pre’ and is maintained for backward-compatibility.

**ACCEPTS:** ['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post']

**set_fillstyle**(fs)
Set the marker fill style; ‘full’ means fill the whole marker. ‘none’ means no filling; other options are for half-filled markers.

**ACCEPTS:** ['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']

**set_linestyle**(ls)
Set the linestyle of the line (also accepts drawstyles, e.g., 'steps--')

<table>
<thead>
<tr>
<th>linestyle</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'-' or 'solid'</td>
<td>solid line</td>
</tr>
<tr>
<td>'--' or 'dashed'</td>
<td>dashed line</td>
</tr>
<tr>
<td>'-.' or 'dashdot'</td>
<td>dash-dotted line</td>
</tr>
<tr>
<td>':' or 'dotted'</td>
<td>dotted line</td>
</tr>
<tr>
<td>'None'</td>
<td>draw nothing</td>
</tr>
<tr>
<td>' '</td>
<td>draw nothing</td>
</tr>
<tr>
<td>''</td>
<td>draw nothing</td>
</tr>
</tbody>
</table>

‘steps’ is equivalent to ‘steps-pre’ and is maintained for backward-compatibility.

Alternatively a dash tuple of the following form can be provided:

```
(offset, onoffseq),
```

where onoffseq is an even length tuple of on and off ink in points.

**ACCEPTS:** ['solid' | 'dashed', 'dashdot', 'dotted' | (offset, onoffdash-seq) | ' ' | ' ' | ' -.' | ':' | 'None' | ' ' | ' ']

See also:

**set_drawstyle()** To set the drawing style (stepping) of the plot.

**Parameters** ls : {'-', '--', '-.', ':'} and more see description

The line style.

**set_linewidth**(w)
Set the line width in points

**ACCEPTS:** float value in points

**set_ls**(val)
alias for set_linestyle

**set_lw**(val)
alias for set_linewidth

**set_marker**(marker)
Set the line marker
ACCEPST: *A valid marker style*

**Parameters marker: marker style**

See *markers* for full description of possible argument

**set_markeredgecolor**(ec)
Set the marker edge color

ACCEPST: any matplotlib color

**set_markeredgewidth**(ew)
Set the marker edge width in points

ACCEPST: float value in points

**set_markerfacecolor**(fc)
Set the marker face color.

ACCEPST: any matplotlib color

**set_markerfacecoloralt**(fc)
Set the alternate marker face color.

ACCEPST: any matplotlib color

**set_markersize**(sz)
Set the marker size in points

ACCEPST: float

**set_markevery**(every)
Set the markevery property to subsample the plot when using markers.

e.g., if every=5, every 5-th marker will be plotted.

ACCEPST: [None | int | length-2 tuple of int | slice | list/array of int | float | length-2 tuple of float]

**Parameters every: None | int | length-2 tuple of int | slice | list/array of int | float | length-2 tuple of float**

Which markers to plot.

- every=None, every point will be plotted.
- every=N, every N-th marker will be plotted starting with marker 0.
- every=(start, N), every N-th marker, starting at point start, will be plotted.
- every=slice(start, end, N), every N-th marker, starting at point start, upto but not including point end, will be plotted.
- every=[i, j, m, n], only markers at points i, j, m, and n will be plotted.
- every=0.1, (i.e. a float) then markers will be spaced at approximately equal distances along the line; the distance along the line between markers is determined by multiplying the display-coordinate distance of the axes bounding-box diagonal by the value of every.
• \textit{every}=(0.5, 0.1) (i.e. a length-2 tuple of float), the same functionality as \textit{every}=0.1 is exhibited but the first marker will be 0.5 multiplied by the display-coordinate-diagonal-distance along the line.

\textbf{Notes}

Setting the \textit{markevery} property will only show markers at actual data points. When using float arguments to set the \textit{markevery} property on irregularly spaced data, the markers will likely not appear evenly spaced because the actual data points do not coincide with the theoretical spacing between markers.

When using a start offset to specify the first marker, the offset will be from the first data point which may be different from the first the visible data point if the plot is zoomed in.

If zooming in on a plot when using float arguments then the actual data points that have markers will change because the distance between markers is always determined from the display-coordinates axes-bounding-box-diagonal regardless of the actual axes data limits.

\texttt{set_mec(val)}
\begin{verbatim}
alias for set_markeredgecolor
\end{verbatim}

\texttt{set_mew(val)}
\begin{verbatim}
alias for set_markeredgewidth
\end{verbatim}

\texttt{set_mfc(val)}
\begin{verbatim}
alias for set_markerfacecolor
\end{verbatim}

\texttt{set_mfcalt(val)}
\begin{verbatim}
alias for set_markerfacecoloralt
\end{verbatim}

\texttt{set_ms(val)}
\begin{verbatim}
alias for set_markersize
\end{verbatim}

\texttt{set_picker(p)}
\begin{verbatim}
Sets the event picker details for the line.
ACCEPTS: float distance in points or callable pick function fn(artist, event)
\end{verbatim}

\texttt{set_pickradius(d)}
\begin{verbatim}
Sets the pick radius used for containment tests
ACCEPTS: float distance in points
\end{verbatim}

\texttt{set_solid_capstyle(s)}
\begin{verbatim}
Set the cap style for solid linestyles
ACCEPTS: ['butt' | 'round' | 'projecting']
\end{verbatim}

\texttt{set_solid_joinstyle(s)}
\begin{verbatim}
Set the join style for solid linestyles ACCEPTS: ['miter' | 'round' | 'bevel']
\end{verbatim}

\texttt{set_transform(t)}
\begin{verbatim}
set the Transformation instance used by this artist
ACCEPTS: a matplotlib.transforms.Transform instance
\end{verbatim}
**set_xdata(x)**
Set the data np.array for x
ACCEPTS: 1D array

**set_ydata(y)**
Set the data np.array for y
ACCEPTS: 1D array

**update_from(other)**
copy properties from other to self

validCap = ('butt', 'round', 'projecting')

validJoin = ('miter', 'round', 'bevel')

zorder = 2

class matplotlib.lines.VertexSelector(line)
Bases: object

Manage the callbacks to maintain a list of selected vertices for *matplotlib.lines.Line2D*. Derived classes should override *process_selected()* to do something with the picks.

Here is an example which highlights the selected verts with red circles:

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.lines as lines

class HighlightSelected(lines.VertexSelector):
    def __init__(self, line, fmt='ro', **kwargs):
        lines.VertexSelector.__init__(self, line)
        self.markers, = self.axes.plot([], [], fmt, **kwargs)

    def process_selected(self, ind, xs, ys):
        self.markers.set_data(xs, ys)
        self.canvas.draw()

fig = plt.figure()
ax = fig.add_subplot(111)
x, y = np.random.rand(2, 30)
line, = ax.plot(x, y, 'bs-', picker=5)
selector = HighlightSelected(line)
plt.show()
```

Initialize the class with a *matplotlib.lines.Line2D* instance. The line should already be added to some *matplotlib.axes.Axes* instance and should have the picker property set.

**onpick**(event)
When the line is picked, update the set of selected indicies.
**process_selected**(*ind*, *xs*, *ys*)

Default “do nothing” implementation of the `process_selected()` method.

*ind* are the indices of the selected vertices. *xs* and *ys* are the coordinates of the selected vertices.

**matplotlib.lines.segment_hits**(*cx*, *cy*, *x*, *y*, *radius*)

Determine if any line segments are within radius of a point. Returns the list of line segments that are within that radius.
37.1 matplotlib.markers

This module contains functions to handle markers. Used by both the marker functionality of plot and scatter.

All possible markers are defined here:

<table>
<thead>
<tr>
<th>marker</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;.&quot;</td>
<td>point</td>
</tr>
<tr>
<td>&quot;,&quot;</td>
<td>pixel</td>
</tr>
<tr>
<td>&quot;o&quot;</td>
<td>circle</td>
</tr>
<tr>
<td>&quot;v&quot;</td>
<td>triangle_down</td>
</tr>
<tr>
<td>&quot;^&quot;</td>
<td>triangle_up</td>
</tr>
<tr>
<td>&quot;&lt;&quot;</td>
<td>triangle_left</td>
</tr>
<tr>
<td>&quot;&gt;&quot;</td>
<td>triangle_right</td>
</tr>
<tr>
<td>&quot;1&quot;</td>
<td>tri_down</td>
</tr>
<tr>
<td>&quot;2&quot;</td>
<td>tri_up</td>
</tr>
<tr>
<td>&quot;3&quot;</td>
<td>tri_left</td>
</tr>
<tr>
<td>&quot;4&quot;</td>
<td>tri_right</td>
</tr>
<tr>
<td>&quot;8&quot;</td>
<td>octagon</td>
</tr>
<tr>
<td>&quot;s&quot;</td>
<td>square</td>
</tr>
<tr>
<td>&quot;p&quot;</td>
<td>pentagon</td>
</tr>
<tr>
<td>&quot;P&quot;</td>
<td>plus (filled)</td>
</tr>
<tr>
<td>&quot;H&quot;</td>
<td>star</td>
</tr>
<tr>
<td>&quot;h&quot;</td>
<td>hexagon1</td>
</tr>
<tr>
<td>&quot;H&quot;</td>
<td>hexagon2</td>
</tr>
<tr>
<td>&quot;+&quot;</td>
<td>plus</td>
</tr>
<tr>
<td>&quot;x&quot;</td>
<td>x</td>
</tr>
<tr>
<td>&quot;X&quot;</td>
<td>x (filled)</td>
</tr>
<tr>
<td>&quot;D&quot;</td>
<td>diamond</td>
</tr>
<tr>
<td>&quot;d&quot;</td>
<td>thin_diamond</td>
</tr>
<tr>
<td>&quot;$&quot;</td>
<td>vline</td>
</tr>
<tr>
<td>&quot;_&quot;</td>
<td>hline</td>
</tr>
<tr>
<td>TICKLEFT</td>
<td>tickleft</td>
</tr>
</tbody>
</table>

Continued on next page
Table 37.1 – continued from previous page

<table>
<thead>
<tr>
<th>marker</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TICKRIGHT</td>
<td>tickright</td>
</tr>
<tr>
<td>TICKUP</td>
<td>tickup</td>
</tr>
<tr>
<td>TICKDOWN</td>
<td>tickdown</td>
</tr>
<tr>
<td>CARETLEFT</td>
<td>caretleft (centered at tip)</td>
</tr>
<tr>
<td>CARETRIGHT</td>
<td>caretright (centered at tip)</td>
</tr>
<tr>
<td>CARETUP</td>
<td>caretup (centered at tip)</td>
</tr>
<tr>
<td>CARETDOWN</td>
<td>caretdown (centered at tip)</td>
</tr>
<tr>
<td>CARETLEFTBASE</td>
<td>caretleft (centered at base)</td>
</tr>
<tr>
<td>CARETRIGHTBASE</td>
<td>caretright (centered at base)</td>
</tr>
<tr>
<td>CARETUPBASE</td>
<td>caretup (centered at base)</td>
</tr>
<tr>
<td>&quot;None&quot;,&quot; &quot; or &quot;&quot;</td>
<td>nothing</td>
</tr>
<tr>
<td>'$...$'</td>
<td>render the string using mathtext.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>verts</th>
<th>a list of (x, y) pairs used for Path vertices. The center of the marker is located at (0,0) and the size is normalized.</th>
</tr>
</thead>
<tbody>
<tr>
<td>path</td>
<td>a Path instance.</td>
</tr>
<tr>
<td>(numsides, style, angle)</td>
<td>The marker can also be a tuple (numsides, style, angle), which will create a custom, regular symbol.</td>
</tr>
</tbody>
</table>

For backward compatibility, the form (verts, 0) is also accepted, but it is equivalent to just verts for giving a raw set of vertices that define the shape.

None is the default which means ‘nothing’, however this table is referred to from other docs for the valid inputs from marker inputs and in those cases None still means ‘default’.

```python
class matplotlib.markers.MarkerStyle(marker=None, fillstyle=None)
    Bases: object

    Parameters marker : string or array_like, optional, default: None
        See the descriptions of possible markers in the module docstring.

    fillstyle : string, optional, default: ‘full’
```
Attributes

<table>
<thead>
<tr>
<th>markers</th>
<th>(list of known markers)</th>
</tr>
</thead>
<tbody>
<tr>
<td>fillstyles</td>
<td>(list of known fillstyles)</td>
</tr>
<tr>
<td>filled_markers</td>
<td>(list of known filled markers.)</td>
</tr>
</tbody>
</table>


- `get_alt_path()`
- `get_alt_transform()`
- `get_capstyle()`
- `get_fillstyle()`
- `get_joinstyle()`
- `get_marker()`
- `get_path()`
- `get_snap_threshold()`
- `get_transform()`
- `is_filled()`


`set_fillstyle(fillstyle)`
Sets fillstyle

Parameters

- `fillstyle` : string amongst known fillstyles

`set_marker(marker)`
CHAPTER

THIRTYEIGHT

MATHTEXT
38.1 matplotlib.mathtext

`mathtext` is a module for parsing a subset of the TeX math syntax and drawing them to a matplotlib backend.

For a tutorial of its usage see `Writing mathematical expressions`. This document is primarily concerned with implementation details.

The module uses `pyparsing` to parse the TeX expression.

The Bakoma distribution of the TeX Computer Modern fonts, and STIX fonts are supported. There is experimental support for using arbitrary fonts, but results may vary without proper tweaking and metrics for those fonts.

```python
class matplotlib.mathtext.Accent(c, state, math=True)
    Bases: matplotlib.mathtext.Char
    The font metrics need to be dealt with differently for accents, since they are already offset correctly from the baseline in TrueType fonts.
    grow()
    render(x, y)
        Render the character to the canvas.
    shrink()

class matplotlib.mathtext.AutoHeightChar(c, height, depth, state, always=False, factor=None)
    Bases: matplotlib.mathtext.Hlist
    `AutoHeightChar` will create a character as close to the given height and depth as possible. When using a font with multiple height versions of some characters (such as the BaKoMa fonts), the correct glyph will be selected, otherwise this will always just return a scaled version of the glyph.

class matplotlib.mathtext.AutoWidthChar(c, width, state, always=False, char_class=<class 'matplotlib.mathtext.Char'>)
    Bases: matplotlib.mathtext.Hlist
    `AutoWidthChar` will create a character as close to the given width as possible. When using a font with multiple width versions of some characters (such as the BaKoMa fonts), the correct glyph will be selected, otherwise this will always just return a scaled version of the glyph.

class matplotlib.mathtext.BakomaFonts(*args, **kwargs)
    Bases: matplotlib.mathtext.TruetypeFonts
    Use the Bakoma TrueType fonts for rendering.
    Symbols are strewn about a number of font files, each of which has its own proprietary 8-bit encoding.
    alias = "\\"
```
get_sized_alternatives_for_symbol(fontname, sym)

target = ‘]’

class matplotlib.mathtext.Box(width, height, depth)
    Bases: matplotlib.mathtext.Node
    Represents any node with a physical location.
    grow()

    render(x1, y1, x2, y2)

    shrink()

class matplotlib.mathtext.Char(c, state, math=True)
    Bases: matplotlib.mathtext.Node
    Represents a single character. Unlike TeX, the font information and metrics are stored with each
Char to make it easier to lookup the font metrics when needed. Note that TeX boxes have a width,
height, and depth, unlike Type1 and TrueType which use a full bounding box and an advance in the
x-direction. The metrics must be converted to the TeX way, and the advance (if different from width)
must be converted into a Kern node when the Char is added to its parent Hlist.

    get_kerning(next)
        Return the amount of kerning between this and the given character. Called when characters are
strung together into Hlist to create Kern nodes.

    grow()

    is_slanted()

    render(x, y)
        Render the character to the canvas

    shrink()

class matplotlib.mathtext.ComputerModernFontConstants
    Bases: matplotlib.mathtext.FontConstantsBase
    delta = 0.075

    delta_integral = 0.3

    delta_slanted = 0.3
```
script_space = 0.075

sub1 = 0.2

sub2 = 0.3

subdrop = 0.2

sup1 = 0.45

class matplotlib.mathtext.DejaVuFonts(*args, **kwargs)
    Bases: matplotlib.mathtext.UnicodeFonts
    use_cmex = False

class matplotlib.mathtext.DejaVuSansFontConstants
    Bases: matplotlib.mathtext.FontConstantsBase

class matplotlib.mathtext.DejaVuSansFonts(*args, **kwargs)
    Bases: matplotlib.mathtext.DejaVuFonts
    A font handling class for the DejaVu Sans fonts
    If a glyph is not found it will fallback to Stix Sans

class matplotlib.mathtext.DejaVuSerifFontConstants
    Bases: matplotlib.mathtext.FontConstantsBase

class matplotlib.mathtext.DejaVuSerifFonts(*args, **kwargs)
    Bases: matplotlib.mathtext.DejaVuFonts
    A font handling class for the DejaVu Serif fonts
    If a glyph is not found it will fallback to Stix Serif

class matplotlib.mathtext.Error(msg)
    Helper class to raise parser errors.

class matplotlib.mathtext.Fill
    Bases: matplotlib.mathtext.Glue

class matplotlib.mathtext.Filll
    Bases: matplotlib.mathtext.Glue

class matplotlib.mathtext.FontConstantsBase
    Bases: object
    A set of constants that controls how certain things, such as sub- and superscripts are laid out. These are all metrics that can’t be reliably retrieved from the font metrics in the font itself.
\[\delta = 0.025\]

\[\delta_{\text{integral}} = 0.1\]

\[\delta_{\text{slanted}} = 0.2\]

\[\text{script\_space} = 0.05\]

\[\text{sub1} = 0.3\]

\[\text{sub2} = 0.5\]

\[\text{subdrop} = 0.4\]

\[\text{sup1} = 0.7\]

class matplotlib.mathtext.Fonts(
default_font_prop, mathtext_backend)

Bases: object

An abstract base class for a system of fonts to use for mathtext.

The class must be able to take symbol keys and font file names and return the character metrics. It also delegates to a backend class to do the actual drawing.

\textit{default_font_prop}: A FontProperties object to use for the default non-math font, or the base font for Unicode (generic) font rendering.

\textit{mathtext_backend}: A subclass of MathTextBackend used to delegate the actual rendering.

def destroy()
    Fix any cyclical references before the object is about to be destroyed.

def get_kern(font1, fontclass1, sym1, fontsize1, font2, fontclass2, sym2, fontsize2, dpi)
    Get the kerning distance for font between sym1 and sym2.

\textit{fontX}: one of the TeX font names:

| tt, it, rm, cal, sf, bf or default/regular (non-math) |

\textit{fontclassX}: TODO

\textit{symX}: a symbol in raw TeX form. e.g., ‘1’, ‘x’ or ‘sigma’

\textit{fontsizeX}: the fontsize in points

\textit{dpi}: the current dots-per-inch

def get_metrics(font, font_class, sym, fontsize, dpi, math=True)

\textit{font}: one of the TeX font names:
font_class: TODO

sym: a symbol in raw TeX form. e.g., ‘1’, ‘x’ or ‘sigma’

fontsize: font size in points

dpi: current dots-per-inch

math: whether sym is a math character

Returns an object with the following attributes:

- advance: The advance distance (in points) of the glyph.
- height: The height of the glyph in points.
- width: The width of the glyph in points.
- xmin, xmax, ymin, ymax - the ink rectangle of the glyph
- iceberg - the distance from the baseline to the top of the glyph. This corresponds to TeX’s definition of “height”.

get_results(box)

Get the data needed by the backend to render the math expression. The return value is backend-specific.

get_sized_alternatives_for_symbol(fontname, sym)

Override if your font provides multiple sizes of the same symbol. Should return a list of symbols matching sym in various sizes. The expression renderer will select the most appropriate size for a given situation from this list.

get_underline_thickness(font, fontsize, dpi)

Get the line thickness that matches the given font. Used as a base unit for drawing lines such as in a fraction or radical.

get_used_characters()

Get the set of characters that were used in the math expression. Used by backends that need to subset fonts so they know which glyphs to include.

get_xheight(font, fontsize, dpi)

Get the xheight for the given font and fontsize.

render_glyph(ox, oy, facename, font_class, sym, fontsize, dpi)

Draw a glyph at

- ox, oy: position
- facename: One of the TeX face names
- font_class:
- sym: TeX symbol name or single character
- fontsize: fontsize in points
- `dpi`: The dpi to draw at.

```python
render_rect_filled(x1, y1, x2, y2)
```
Draw a filled rectangle from \((x1, y1)\) to \((x2, y2)\).

```python
set_canvas_size(w, h, d)
```
Set the size of the buffer used to render the math expression. Only really necessary for the bitmap backends.

```python
class matplotlib.mathtext.Glue(glue_type, copy=False)

Bases: matplotlib.mathtext.Node
```
Most of the information in this object is stored in the underlying `GlueSpec` class, which is shared between multiple glue objects. (This is a memory optimization which probably doesn’t matter anymore, but it’s easier to stick to what TeX does.)

```python
grow()
```

```python
shrink()
```

```python
class matplotlib.mathtext.GlueSpec(width=0.0, stretch=0.0, stretch_order=0, shrink=0.0, shrink_order=0)

Bases: object
```
See `Glue`.

```python
copy()
```

```python
classmethod factory(glue_type)
```

```python
class matplotlib.mathtext.HCentered(elements)

Bases: matplotlib.mathtext.Hlist
```
A convenience class to create an `Hlist` whose contents are centered within its enclosing box.

```python
class matplotlib.mathtext.HBox(width)

Bases: matplotlib.mathtext.Box
```
A box with only width (zero height and depth).

```python
class matplotlib.mathtext.HList(elements, w=0.0, m='additional', do_kern=True)

Bases: matplotlib.mathtext.List
```
A horizontal list of boxes.

```python
hpack(w=0.0, m='additional')
```
The main duty of `hpack()` is to compute the dimensions of the resulting boxes, and to adjust the glue if one of those dimensions is pre-specified. The computed sizes normally enclose all of the material inside the new box; but some items may stick out if negative glue is used, if the box is overfull, or if a `\vbox` includes other boxes that have been shifted left.

```python
- `w`: specifies a width
```
• $m$: is either 'exactly' or 'additional'.

Thus, hpack($w$, 'exactly') produces a box whose width is exactly $w$, while hpack($w$, 'additional') yields a box whose width is the natural width plus $w$. The default values produce a box with the natural width.

**kern()**

Insert *Kern* nodes between *Char* nodes to set kerning. The *Char* nodes themselves determine the amount of kerning they need (in *get_kerning()*), and this function just creates the linked list in the correct way.

**class matplotlib.mathtext.Hrule(state, thickness=None)**

Bases: matplotlib.mathtext.Rule

Convenience class to create a horizontal rule.

**class matplotlib.mathtext.Kern(width)**

Bases: matplotlib.mathtext.Node

A *Kern* node has a width field to specify a (normally negative) amount of spacing. This spacing correction appears in horizontal lists between letters like A and V when the font designer said that it looks better to move them closer together or further apart. A kern node can also appear in a vertical list, when its *width* denotes additional spacing in the vertical direction.

**depth = 0**

**grow()**

**height = 0**

**shrink()**

**class matplotlib.mathtext.List(elements)**

Bases: matplotlib.mathtext.Box

A list of nodes (either horizontal or vertical).

**grow()**

**shrink()**

**class matplotlib.mathtext.MathTextParser(output)**

Bases: object

Create a MathTextParser for the given backend *output*.

**get_depth(texstr, dpi=120, fontsize=14)**

Returns the offset of the baseline from the bottom of the image in pixels.

*texstr* A valid mathtext string, e.g., r‘IQ: $\sigma_i = 15$’
\textit{dpi} The dots-per-inch to render the text

\textit{fontsize} The font size in points

\texttt{parse}(s, dpi=72, prop=None)

Parse the given math expression \textit{s} at the given \textit{dpi}. If \textit{prop} is provided, it is a \texttt{FontProperties} object specifying the “default” font to use in the math expression, used for all non-math text.

The results are cached, so multiple calls to \texttt{parse()} with the same expression should be fast.

\texttt{to_mask}(\textit{texstr}, dpi=120, fontsize=14)

\textit{texstr} A valid mathtext string, e.g., r’IQ: $\sigma_i = 15$’

\textit{dpi} The dots-per-inch to render the text

\textit{fontsize} The font size in points

Returns a tuple (\textit{array}, \textit{depth})

- \textit{array} is an NxM uint8 alpha ubyte mask array of rasterized tex.
- \textit{depth} is the offset of the baseline from the bottom of the image in pixels.

\texttt{to_png}(\textit{filename}, texstr, color='black', dpi=120, fontsize=14)

Writes a tex expression to a PNG file.

Returns the offset of the baseline from the bottom of the image in pixels.

\textit{filename} A writable filename or fileobject

\textit{texstr} A valid mathtext string, e.g., r’IQ: $\sigma_i = 15$’

\textit{color} A valid matplotlib color argument

\textit{dpi} The dots-per-inch to render the text

\textit{fontsize} The font size in points

Returns the offset of the baseline from the bottom of the image in pixels.

\texttt{to_rgba}(\textit{texstr}, color='black', dpi=120, fontsize=14)

\textit{texstr} A valid mathtext string, e.g., r’IQ: $\sigma_i = 15$’

\textit{color} Any matplotlib color argument

\textit{dpi} The dots-per-inch to render the text

\textit{fontsize} The font size in points

Returns a tuple (\textit{array}, \textit{depth})

- \textit{array} is an NxM uint8 alpha ubyte mask array of rasterized tex.
- \textit{depth} is the offset of the baseline from the bottom of the image in pixels.

\texttt{exception} \texttt{matplotlib.mathtext.MathTextWarning}

Bases: \texttt{Warning}
class matplotlib.mathtext.MathtextBackend
Bases: object

The base class for the mathtext backend-specific code. The purpose of MathtextBackend subclasses is to interface between mathtext and a specific matplotlib graphics backend.

Subclasses need to override the following:

- `render_glyph()`
- `render_rect_filled()`
- `get_results()`

And optionally, if you need to use a FreeType hinting style:

- `get_hinting_type()`

`get_hinting_type()`
Get the FreeType hinting type to use with this particular backend.

`get_results(box)`
Return a backend-specific tuple to return to the backend after all processing is done.

`render_glyph(ox, oy, info)`
Draw a glyph described by info to the reference point (ox, oy).

`render_rect_filled(xl, yl, x2, y2)`
Draw a filled black rectangle from (xl, yl) to (x2, y2).

`set_canvas_size(w, h, d)`
Dimension the drawing canvas

class matplotlib.mathtext.MathtextBackendAgg
Bases: matplotlib.mathtext.MathtextBackend

Render glyphs and rectangles to an FTImage buffer, which is later transferred to the Agg image by the Agg backend.

`get_hinting_type()`

`get_results(box, used_characters)`

`render_glyph(ox, oy, info)`

`render_rect_filled(xl, yl, x2, y2)`

`set_canvas_size(w, h, d)`

class matplotlib.mathtext.MathtextBackendBitmap
Bases: matplotlib.mathtext.MathtextBackendAgg

38.1. matplotlib.mathtext

1269
get_results(box, used_characters)

class matplotlib.mathtext.MathtextBackendCairo
    Bases: matplotlib.mathtext.MathtextBackend
    Store information to write a mathtext rendering to the Cairo backend.
    get_results(box, used_characters)

    render_glyph(ox, oy, info)

    render_rect_filled(x1, y1, x2, y2)

class matplotlib.mathtext.MathtextBackendPath
    Bases: matplotlib.mathtext.MathtextBackend
    Store information to write a mathtext rendering to the text path machinery.
    get_results(box, used_characters)

    render_glyph(ox, oy, info)

    render_rect_filled(x1, y1, x2, y2)

class matplotlib.mathtext.MathtextBackendPdf
    Bases: matplotlib.mathtext.MathtextBackend
    Store information to write a mathtext rendering to the PDF backend.
    get_results(box, used_characters)

    render_glyph(ox, oy, info)

    render_rect_filled(x1, y1, x2, y2)

class matplotlib.mathtext.MathtextBackendPs
    Bases: matplotlib.mathtext.MathtextBackend
    Store information to write a mathtext rendering to the PostScript backend.
    get_results(box, used_characters)

    render_glyph(ox, oy, info)

    render_rect_filled(x1, y1, x2, y2)
class matplotlib.mathtext.MathTextBackendSvg
    Bases: matplotlib.mathtext.MathTextBackend

    Store information to write a mathtext rendering to the SVG backend.

    get_results(box, used_characters)

    render_glyph(ox, oy, info)

    render_rect_filled(x1, y1, x2, y2)

class matplotlib.mathtext.NegFil
    Bases: matplotlib.mathtext.Glue

class matplotlib.mathtext.NegFill
    Bases: matplotlib.mathtext.Glue

class matplotlib.mathtext.NegFilll
    Bases: matplotlib.mathtext.Glue

class matplotlib.mathtext.Node
    Bases: object
    A node in the TeX box model

    get_kerning(next)

    grow()
        Grows one level larger. There is no limit to how big something can get.

    render(x, y)

    shrink()
        Shrinks one level smaller. There are only three levels of sizes, after which things will no longer
        get smaller.

class matplotlib.mathtext.Parser
    Bases: object

    This is the pyparsing-based parser for math expressions. It actually parses full strings containing math
    expressions, in that raw text may also appear outside of pairs of $.

    The grammar is based directly on that in TeX, though it cuts a few corners.

class State(font_output, font, font_class, fontsize, dpi)
    Bases: object

    Stores the state of the parser.

    States are pushed and popped from a stack as necessary, and the “current” state is always at the
    top of the stack.
copy()

font

accent(s, loc, toks)

auto_delim(s, loc, toks)

binom(s, loc, toks)

c_over_c(s, loc, toks)

customspace(s, loc, toks)

dupbeg(s, loc, toks)

dupend(s, loc, toks)

dupmark(s, loc, toks)

dupsep(s, loc, toks)

duplicate(s, loc, toks)

duplicate_beg(s, loc, toks)

duplicate_end(s, loc, toks)

duplicate_mark(s, loc, toks)

duplicate_sep(s, loc, toks)

euclidian(s, loc, toks)

get_state()

Get the current State of the parser.

group(s, loc, toks)

is_between_brackets(s, loc)

is_dropsup(nucleus)

is_overunder(nucleus)

is_slanted(nucleus)

main(s, loc, toks)
math\(s, \text{loc}, \text{toks}\)

math_string\(s, \text{loc}, \text{toks}\)

non_math\(s, \text{loc}, \text{toks}\)

operatorname\(s, \text{loc}, \text{toks}\)

overline\(s, \text{loc}, \text{toks}\)

parse\(s, \text{fonts}, \text{fontsize}, \text{dpi}\)
  Parse expression \(s\) using the given \text{fonts} for output, at the given \text{fontsize} and \text{dpi}.
  Returns the parse tree of \text{Node} instances.

pop_state()
  Pop a \text{State} off of the stack.

push_state()
  Push a new \text{State} onto the stack which is just a copy of the current state.

required_group\(s, \text{loc}, \text{toks}\)

simple_group\(s, \text{loc}, \text{toks}\)

snowflake\(s, \text{loc}, \text{toks}\)

space\(s, \text{loc}, \text{toks}\)

sqrt\(s, \text{loc}, \text{toks}\)

stackrel\(s, \text{loc}, \text{toks}\)

start_group\(s, \text{loc}, \text{toks}\)

subsuper\(s, \text{loc}, \text{toks}\)

symbol\(s, \text{loc}, \text{toks}\)

unknown_symbol\(s, \text{loc}, \text{toks}\)
class matplotlib.mathtext.Rule(width, height, depth, state)
    Bases: matplotlib.mathtext.Box

    A Rule node stands for a solid black rectangle; it has width, depth, and height fields just as in an Hlist. However, if any of these dimensions is inf, the actual value will be determined by running the rule up to the boundary of the innermost enclosing box. This is called a “running dimension.” The width is never running in an Hlist; the height and depth are never running in a Vlist.

    render(x, y, w, h)

class matplotlib.mathtext.STIXFontConstants
    Bases: matplotlib.mathtext.FontConstantsBase

    delta = 0.05

    delta_integral = 0.3

    delta_slanted = 0.3

    script_space = 0.1

    sub2 = 0.6

    sup1 = 0.8

class matplotlib.mathtext.STIXSansFontConstants
    Bases: matplotlib.mathtext.FontConstantsBase

    delta_integral = 0.3

    delta_slanted = 0.6

    script_space = 0.05

    sup1 = 0.8

class matplotlib.mathtext.Ship
    Bases: object

    Once the boxes have been set up, this sends them to output. Since boxes can be inside of boxes inside of boxes, the main work of Ship is done by two mutually recursive routines, hlist_out() and vlist_out(), which traverse the Hlist nodes and Vlist nodes inside of horizontal and vertical boxes. The global variables used in TeX to store state as it processes have become member variables here.
static clamp(value)

hlist_out(box)

vlist_out(box)

class matplotlib.mathtext.SsGlue
   Bases: matplotlib.mathtext.Glue

class matplotlib.mathtext.StandardPsFonts(default_font_prop)
   Bases: matplotlib.mathtext.Fonts
   Use the standard postscript fonts for rendering to backend_ps
   Unlike the other font classes, BakomaFont and UnicodeFont, this one requires the Ps backend.
   basepath = '/tmp/mpl_clean/lib/python3.6/site-packages/matplotlib/mpl-data/fonts/afm'

   fontmap = {'cal': 'pzcmi8a', 'rm': 'pncr8a', 'tt': 'pcrr8a', 'it': 'pncri8a', 'sf': 'phvr8a', 'bf': 'pncb8a', 'itc': 'pncit8a', 'bfi': 'pncbi8a', 'itc': 'pncit8a', None: 'psyr'}

   get_kern(font1, fontclass1, sym1, fontsize1, font2, fontclass2, sym2, fontsize2, dpi)

   get_underline_thickness(font, fontsize, dpi)

   get_xheight(font, fontsize, dpi)

class matplotlib.mathtext.StixFonts(*args, **kwargs)
   Bases: matplotlib.mathtext.UnicodeFonts
   A font handling class for the STIX fonts.
   In addition to what UnicodeFonts provides, this class:
   • supports “virtual fonts” which are complete alpha numeric character sets with different font styles at special Unicode code points, such as “Blackboard”.
   • handles sized alternative characters for the STIXSizeX fonts.
   cm_fallback = False

   get_sized_alternatives_for_symbol(fontname, sym)

   use_cmex = False

class matplotlib.mathtext.StixSansFonts(*args, **kwargs)
   Bases: matplotlib.mathtext.StixFonts
A font handling class for the STIX fonts (that uses sans-serif characters by default).

**class matplotlib.mathtext.SubSuperCluster**

Bases: *matplotlib.mathtext.Hlist*

*SubSuperCluster* is a sort of hack to get around that fact that this code do a two-pass parse like TeX. This lets us store enough information in the hlist itself, namely the nucleus, sub- and super-script, such that if another script follows that needs to be attached, it can be reconfigured on the fly.

**class matplotlib.mathtext.TruetypeFonts**(default_font_prop, mathtext_backend)

Bases: *matplotlib.mathtext.Fonts*

A generic base class for all font setups that use Truetype fonts (through FT2Font).

*destroy()*

*get_kern*(font1, fontclass1, sym1, fontsize1, font2, fontclass2, sym2, fontsize2, dpi)*

*get_underline_thickness*(font, fontsize, dpi)*

*get_xheight*(fontname, fontsize, dpi)*

**class matplotlib.mathtext.UnicodeFonts**( *args, **kwargs)*

Bases: *matplotlib.mathtext.TruetypeFonts*

An abstract base class for handling Unicode fonts.

While some reasonably complete Unicode fonts (such as DejaVu) may work in some situations, the only Unicode font I’m aware of with a complete set of math symbols is STIX.

This class will “fallback” on the Bakoma fonts when a required symbol can not be found in the font.

*get_sized_alternatives_for_symbol*(fontname, sym)*

*use_cmex = True*

**class matplotlib.mathtext.VCentered**(elements)

Bases: *matplotlib.mathtext.Hlist*

A convenience class to create a *Vlist* whose contents are centered within its enclosing box.

**class matplotlib.mathtext.Vbox**(height, depth)

Bases: *matplotlib.mathtext.Box*

A box with only height (zero width).

**class matplotlib.mathtext.Vlist**(elements, h=0.0, m=‘additional’)

Bases: *matplotlib.mathtext.List*

A vertical list of boxes.
vp\texttt{ack}(h=0.0, m='additional', l=\text{inf})

The main duty of \texttt{vpack()} is to compute the dimensions of the resulting boxes, and to adjust the glue if one of those dimensions is pre-specified.

- \(h\): specifies a height
- \(m\): is either ‘exactly’ or ‘additional’.
- \(l\): a maximum height

Thus, \texttt{vpack(h, 'exactly')} produces a box whose height is exactly \(h\), while \texttt{vpack(h, 'additional')} yields a box whose height is the natural height plus \(h\). The default values produce a box with the natural width.

\texttt{class matplotlib.mathtext.Vrule(state)}

\texttt{Bases: matplotlib.mathtext.Rule}

Convenience class to create a vertical rule.

\texttt{matplotlib.mathtext.get_unicode_index(symbol[, bool]) \rightarrow integer}

Return the integer index (from the Unicode table) of symbol. \texttt{symbol} can be a single unicode character, a TeX command (i.e. \texttt{r'pi')}), or a Type1 symbol name (i.e. ‘phi’). If math is False, the current symbol should be treated as a non-math symbol.

\texttt{matplotlib.mathtext.math_to_image(s, filename_or_obj, prop=None, dpi=None, format=None)}

Given a math expression, renders it in a closely-clipped bounding box to an image file.

- \texttt{s} A math expression. The math portion should be enclosed in dollar signs.
- \texttt{filename_or_obj} A filepath or writable file-like object to write the image data to.
- \texttt{prop} If provided, a FontProperties() object describing the size and style of the text.
- \texttt{dpi} Override the output dpi, otherwise use the default associated with the output format.
- \texttt{format} The output format, e.g., ‘svg’, ‘pdf’, ‘ps’ or ‘png’. If not provided, will be deduced from the filename.

\texttt{matplotlib.mathtext.unichr_safe(index)}

Return the Unicode character corresponding to the index, or the replacement character if this is a narrow build of Python and the requested character is outside the BMP.
39.1 matplotlib.mlab

Numerical python functions written for compatibility with MATLAB commands with the same names.

39.1.1 MATLAB compatible functions

- **cohere()**  Coherence (normalized cross spectral density)
- **csd()**  Cross spectral density using Welch’s average periodogram
- **detrend()**  Remove the mean or best fit line from an array
- **find()**  Return the indices where some condition is true; numpy.nonzero is similar but more general.
- **griddata()**  Interpolate irregularly distributed data to a regular grid.
- **prctile()**  Find the percentiles of a sequence
- **prepca()**  Principal Component Analysis
- **psd()**  Power spectral density using Welch’s average periodogram
- **rk4()**  A 4th order runge kutta integrator for 1D or ND systems
- **specgram()**  Spectrogram (spectrum over segments of time)

39.1.2 Miscellaneous functions

Functions that don’t exist in MATLAB, but are useful anyway:

- **cohere_pairs()**  Coherence over all pairs. This is not a MATLAB function, but we compute coherence a lot in my lab, and we compute it for a lot of pairs. This function is optimized to do this efficiently by caching the direct FFTs.
- **rk4()**  A 4th order Runge-Kutta ODE integrator in case you ever find yourself stranded without scipy (and the far superior scipy.integrate tools)
- **contiguous_regions()**  Return the indices of the regions spanned by some logical mask
- **cross_from_below()**  Return the indices where a 1D array crosses a threshold from below
cross_from_above() Return the indices where a 1D array crosses a threshold from above
complex_spectrum() Return the complex-valued frequency spectrum of a signal
magnitude_spectrum() Return the magnitude of the frequency spectrum of a signal
angle_spectrum() Return the angle (wrapped phase) of the frequency spectrum of a signal
phase_spectrum() Return the phase (unwrapped angle) of the frequency spectrum of a signal
detrend_mean() Remove the mean from a line.
demean() Remove the mean from a line. This function is the same as detrend_mean() except for the default axis.
detrend_linear() Remove the best fit line from a line.
detrend_none() Return the original line.
stride_windows() Get all windows in an array in a memory-efficient manner
stride_repeat() Repeat an array in a memory-efficient manner
apply_window() Apply a window along a given axis

39.1.3 record array helper functions

A collection of helper methods for numpy record arrays

See misc Examples
rec2txt() Pretty print a record array
rec2csv() Store record array in CSV file
csv2rec() Import record array from CSV file with type inspection
rec_append_fields() Adds field(s)/array(s) to record array
rec_drop_fields() Drop fields from record array
rec_join() Join two record arrays on sequence of fields
recs_join() A simple join of multiple recarrays using a single column as a key
rec_groupby() Summarize data by groups (similar to SQL GROUP BY)
rec_summarize() Helper code to filter rec array fields into new fields

For the rec viewer functions (e.g, rec2csv), there are a bunch of Format objects you can pass into the functions that will do things like color negative values red, set percent formatting and scaling, etc.

Example usage:

```python
r = csv2rec('somefile.csv', checkrows=0)
formatd = dict(
    weight = FormatFloat(2),
    change = FormatPercent(2),
)```

```
cost  = FormatThousands(2),
)

rec2excel(r, 'test.xls', formatd=formatd)
rec2csv(r, 'test.csv', formatd=formatd)
scroll = rec2gtk(r, formatd=formatd)

win = gtk.Window()
win.set_size_request(600,800)
win.add(scroll)
win.show_all()
gtk.main()

class matplotlib.mlab.FormatBool
    Bases: matplotlib.mlab.FormatObj

    fromstr(s)

    toval(x)

class matplotlib.mlab.FormatDate(fmt)
    Bases: matplotlib.mlab.FormatObj

    fromstr(x)

    toval(x)

class matplotlib.mlab.FormatDatetime(fmt='%Y-%m-%d %H:%M:%S')
    Bases: matplotlib.mlab.FormatDate

    fromstr(x)

class matplotlib.mlab.FormatFloat(precision=4, scale=1.0)
    Bases: matplotlib.mlab.FormatFormatStr

    fromstr(s)

    toval(x)

class matplotlib.mlab.FormatFormatStr(fmt)
    Bases: matplotlib.mlab.FormatObj

    tostr(x)

class matplotlib.mlab.FormatInt
    Bases: matplotlib.mlab.FormatObj
```
fromstr(s)

tostr(x)

toval(x)

class matplotlib.mlab.FormatMillions(precision=4)
    Bases: matplotlib.mlab.FormatFloat

class matplotlib.mlab.FormatObj
    Bases: object
        fromstr(s)

tostr(x)

toval(x)

class matplotlib.mlab.FormatPercent(precision=4)
    Bases: matplotlib.mlab.FormatFloat

class matplotlib.mlab.FormatString
    Bases: matplotlib.mlab.FormatObj
        tostr(x)

class matplotlib.mlab.FormatThousands(precision=4)
    Bases: matplotlib.mlab.FormatFloat

class matplotlib.mlab.GaussianKDE(dataset, bw_method=None)
    Bases: object
    Representation of a kernel-density estimate using Gaussian kernels.

    Parameters
    ----------
    dataset : array_like
        Datapoints to estimate from. In case of univariate data this is a 1-D array, 
        otherwise a 2-D array with shape (# of dims, # of data).

    bw_method : str, scalar or callable, optional
        The method used to calculate the estimator bandwidth. This can be ‘scott’, 
        ‘silverman’, a scalar constant or a callable. If a scalar, this will be used di-
        rectly as kde.factor. If a callable, it should take a GaussianKDE instance 
        as only parameter and return a scalar. If None (default), ‘scott’ is used.
Attributes

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dataset</td>
<td>(ndarray) The dataset with which <code>gaussian_kde</code> was initialized.</td>
</tr>
<tr>
<td>dim</td>
<td>(int) Number of dimensions.</td>
</tr>
<tr>
<td>num_dp</td>
<td>(int) Number of datapoints.</td>
</tr>
<tr>
<td>factor</td>
<td>(float) The bandwidth factor, obtained from <code>kde.covariance_factor</code>, with which the covariance matrix is multiplied.</td>
</tr>
<tr>
<td>covariance</td>
<td>(ndarray) The covariance matrix of <code>dataset</code>, scaled by the calculated bandwidth <code>kde.factor</code>.</td>
</tr>
<tr>
<td>inv_cov</td>
<td>(ndarray) The inverse of <code>covariance</code>.</td>
</tr>
</tbody>
</table>

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>kde.evaluate(points)</code></td>
<td>(ndarray) Evaluate the estimated pdf on a provided set of points.</td>
</tr>
<tr>
<td><code>kde(points)</code></td>
<td>(ndarray) Same as <code>kde.evaluate(points)</code></td>
</tr>
</tbody>
</table>

```
covariance_factor()
```

evaluate(points)

Evaluate the estimated pdf on a set of points.

Parameters:

<table>
<thead>
<tr>
<th>Points</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>points</td>
<td>(# of dimensions, # of points)-array</td>
</tr>
<tr>
<td></td>
<td>Alternatively, a (# of dimensions,) vector can be passed in and treated as a single point.</td>
</tr>
</tbody>
</table>

Returns:

<table>
<thead>
<tr>
<th>Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>values</td>
<td>(# of points,) array</td>
</tr>
<tr>
<td></td>
<td>The values at each point.</td>
</tr>
</tbody>
</table>

Raises:

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ValueError</td>
<td>if the dimensionality of the input points is different than the dimensionality of the KDE.</td>
</tr>
</tbody>
</table>

```
scotts_factor()
```

```
silverman_factor()
```

class matplotlib.mlab.PCA(a, standardize=True)

Bases: object

compute the SVD of a and store data for PCA. Use project to project the data onto a reduced set of dimensions

Parameters:

<table>
<thead>
<tr>
<th>a</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>np.ndarray</td>
</tr>
<tr>
<td>standardize</td>
<td>bool</td>
</tr>
<tr>
<td></td>
<td>A numobservations x numdims array</td>
</tr>
</tbody>
</table>

39.1. matplotlib.mlab
True if input data are to be standardized. If False, only centering will be carried out.

Notes

The factor loadings are in the $W_t$ factor, i.e., the factor loadings for the first principal component are given by $W_t[0]$. This row is also the first eigenvector.

Attributes

<table>
<thead>
<tr>
<th>a</th>
<th>A centered unit sigma version of input a.</th>
</tr>
</thead>
<tbody>
<tr>
<td>numrows, numcols</td>
<td>The dimensions of a.</td>
</tr>
<tr>
<td>mu</td>
<td>A numdims array of means of a. This is the vector that points to the origin of PCA space.</td>
</tr>
<tr>
<td>sigma</td>
<td>A numdims array of standard deviation of a.</td>
</tr>
<tr>
<td>fracs</td>
<td>The proportion of variance of each of the principal components.</td>
</tr>
<tr>
<td>s</td>
<td>The actual eigenvalues of the decomposition.</td>
</tr>
<tr>
<td>Wt</td>
<td>The weight vector for projecting a numdims point or array into PCA space.</td>
</tr>
<tr>
<td>Y</td>
<td>A projected into PCA space.</td>
</tr>
</tbody>
</table>

center($x$)

center and optionally standardize the data using the mean and sigma from training set $a$

project($x$, minfrac=0.0)

project $x$ onto the principle axes, dropping any axes where fraction of variance < minfrac

matplotlib.mlab.amap(function, sequence[, sequence, ...]) → array.

Works like map(), but it returns an array. This is just a convenient shorthand for numpy.array(map(. ..)).

matplotlib.mlab.angle_spectrum($x$, Fs=None, window=None, pad_to=None, sides=None)

Compute the angle of the frequency spectrum (wrapped phase spectrum) of $x$. Data is padded to a length of pad_to and the windowing function window is applied to the signal.

Parameters $x$ : 1-D array or sequence

Array or sequence containing the data

Fs : scalar

The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.

window : callable or ndarray

A function or a vector of length NFFT. To create window vectors see $\text{window} \_\text{hanning}$(), $\text{window} \_\text{none}$(), $\text{numpy} \_\text{blackman}$(), $\text{numpy} \_\text{hamming}$(), $\text{numpy} \_\text{bartlett}$(), $\text{scipy} \_\text{signal}()$, $\text{scipy} \_\text{signal} \_\text{get} \_\text{window}$(), etc. The default is $\text{window} \_\text{hanning}$(). If a function is
passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

sides : [‘default’ | ‘onesided’ | ‘twosided’]

Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. ‘onesided’ forces the return of a one-sided spectrum, while ‘twosided’ forces two-sided.

pad_to : integer

The number of points to which the data segment is padded when performing the FFT. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the n parameter in the call to fft(). The default is None, which sets pad_to equal to the length of the input signal (i.e. no padding).

Returns spectrum : 1-D array

The values for the angle spectrum in radians (real valued)

freqs : 1-D array

The frequencies corresponding to the elements in spectrum

See also:

complex_spectrum() This function returns the angle value of complex_spectrum().
magnitude_spectrum() angle_spectrum() returns the magnitudes of the corresponding frequencies.
phase_spectrum() phase_spectrum() returns the unwrapped version of this function.
specgram() specgram() can return the angle spectrum of segments within the signal.

matplotlib.mlab.apply_window(x, window, axis=0, return_window=None)

Apply the given window to the given 1D or 2D array along the given axis.

Parameters x : 1D or 2D array or sequence

Array or sequence containing the data.

window : function or array.

Either a function to generate a window or an array with length x.shape[axis]

axis : integer

The axis over which to do the repetition. Must be 0 or 1. The default is 0

return_window : bool

If true, also return the 1D values of the window that was applied
matplotlib.mlab.base_repr(number, base=2, padding=0)
Return the representation of a number in any given base.

matplotlib.mlab.binary_repr(number, max_length=1025)
Return the binary representation of the input number as a string.
This is more efficient than using base_repr() with base 2.
Increase the value of max_length for very large numbers. Note that on 32-bit machines, 2**1023 is the largest integer power of 2 which can be converted to a Python float.

matplotlib.mlab.bivariate_normal(X, Y, sigmax=1.0, sigmay=1.0, mux=0.0, muy=0.0, sigmaxy=0.0)
Bivariate Gaussian distribution for equal shape X, Y.
See bivariate normal at mathworld.

matplotlib.mlab.center_matrix(M, dim=0)
Return the matrix M with each row having zero mean and unit std.
If dim = 1 operate on columns instead of rows. (dim is opposite to the numpy axis kwarg.)

matplotlib.mlab.cohere(x, y, NFFT=256, Fs=2, detrend=<function detrend_none>, window=<function window_hanning>, noverlap=0, pad_to=None, sides='default', scale_by_freq=None)
The coherence between x and y. Coherence is the normalized cross spectral density:
\[ C_{xy} = \frac{|P_{xy}|^2}{P_{xx}P_{yy}} \]  

Parameters x, y
Array or sequence containing the data

Fs : scalar
The sampling frequency (samples per time unit). It is used to calculate the
Fourier frequencies, freqs, in cycles per time unit. The default value is 2.

window : callable or ndarray
A function or a vector of length NFFT. To create window vectors see
window_hanning(), window_none(), numpy.blackman(), numpy.
hamming(), numpy.bartlett(), scipy.signal(), scipy.signal.
get_window(), etc. The default is window_hanning(). If a function is
passed as the argument, it must take a data segment as an argument and re-
turn the windowed version of the segment.

sides : ['default' | 'onesided' | 'twosided']
Specifies which sides of the spectrum to return. Default gives the default
behavior, which returns one-sided for real data and both for complex data.
‘onesided’ forces the return of a one-sided spectrum, while ‘twosided’ forces
two-sided.

pad_to : integer
The number of points to which the data segment is padded when performing the FFT. This can be different from \textit{NFFT}, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the \textit{n} parameter in the call to \texttt{fft()}. The default is \texttt{None}, which sets \textit{pad_to} equal to \textit{NFFT}.

\textbf{NFFT} : integer

The number of data points used in each block for the FFT. A power 2 is most efficient. The default value is 256. This should \textit{NOT} be used to get zero padding, or the scaling of the result will be incorrect. Use \textit{pad_to} for this instead.

\textbf{detrend} : \{‘default’, ‘constant’, ‘mean’, ‘linear’, ‘none’\} or callable

The function applied to each segment before \texttt{fft-ing}, designed to remove the mean or linear trend. Unlike in MATLAB, where the \textit{detrend} parameter is a vector, in matplotlib it is a function. The \texttt{pylab} module defines \texttt{detrend\_none()}, \texttt{detrend\_mean()}, and \texttt{detrend\_linear()}, but you can use a custom function as well. You can also use a string to choose one of the functions. ‘default’, ‘constant’, and ‘mean’ call \texttt{detrend\_mean()}. ‘linear’ calls \texttt{detrend\_linear()}. ‘none’ calls \texttt{detrend\_none()}.

\textbf{scale\_by\_freq} : boolean, optional

Specifies whether the resulting density values should be scaled by the scaling frequency, which gives density in units of Hz$^{-1}$. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

\textbf{noverlap} : integer

The number of points of overlap between blocks. The default value is 0 (no overlap).

\textbf{Returns} The return value is the tuple \((Cxy, f)\), where \(f\) are the frequencies of the coherence vector. For \texttt{cohere}, scaling the individual densities by the sampling frequency has no effect, since the factors cancel out.

\textbf{See also:} \texttt{psd()}, \texttt{csd()}

\texttt{matplotlib.mlab.cohere\_pairs}(\(X\), \(ij\), \textit{NFFT}=256, \(Fs=2\), \textit{detrend=<function detrend\_none>}, \textit{window=<function window\_hanning>}, \textit{noverlap}=0, \textit{prefer\_SpeedOverMemory}=\texttt{True}, \textit{progressCallback=<function donothing\_callback>}, \textit{returnPxx}=\texttt{False})

Compute the coherence and phase for all pairs \(ij\), in \(X\).

\(X\) is a \texttt{numSamples \* numCols} array
$ij$ is a list of tuples. Each tuple is a pair of indexes into the columns of $X$ for which you want to compute coherence. For example, if $X$ has 64 columns, and you want to compute all nonredundant pairs, define $ij$ as:

```python
ij = []
for i in range(64):
    for j in range(i+1, 64):
        ij.append( (i,j) )
```

$\text{preferSpeedOverMemory}$ is an optional bool. Defaults to true. If False, limits the caching by only making one, rather than two, complex cache arrays. This is useful if memory becomes critical. Even when $\text{preferSpeedOverMemory}$ is False, $\text{cohere_pairs()}$ will still give significant performance gains over calling $\text{cohere()}$ for each pair, and will use substantially less memory than if $\text{preferSpeedOverMemory}$ is True. In my tests with a 43000,64 array over all nonredundant pairs, $\text{preferSpeedOverMemory} = \text{True}$ delivered a 33% performance boost on a 1.7GHZ Athlon with 512MB RAM compared with $\text{preferSpeedOverMemory} = \text{False}$. But both solutions were more than 10x faster than naively crunching all possible pairs through $\text{cohere()}$.

**Returns** $\text{Cxy}$ : dictionary of $(i, j)$ tuples -> coherence vector for that pair. i.e., $\text{Cxy}[(i, j)] = \text{cohere}(X[:,i], X[:,j])$. Number of dictionary keys is $\text{len}(ij)$.

$\text{Phase}$ : dictionary of phases of the cross spectral density at each frequency for each pair. Keys are $(i, j)$.

$\text{freqs}$ : vector of frequencies, equal in length to either the coherence or phase vectors for any $(i, j)$ key.

e.g., to make a coherence Bode plot:

```python
subplot(211)
plot( ffreqs, Cxy[(12,19)])
subplot(212)
plot( ffreqs, Phase[(12,19)])
```

For a large number of pairs, $\text{cohere_pairs()}$ can be much more efficient than just calling $\text{cohere()}$ for each pair, because it caches most of the intensive computations. If $N$ is the number of pairs, this function is $O(N)$ for most of the heavy lifting, whereas calling cohere for each pair is $O(N^2)$. However, because of the caching, it is also more memory intensive, making 2 additional complex arrays with approximately the same number of elements as $X$.

See test/cohere_pairs_test.py in the src tree for an example script that shows that this $\text{cohere_pairs()}$ and
Matplotlib, Release 2.0.2

cohere() give the same results for a given pair.

See also:

psd() For information about the methods used to compute $P_{xy}$, $P_{xx}$ and $P_{yy}$.

**matplotlib.mlab.complex_spectrum**($x, Fs=None, window=None, pad_to=None, sides=None$)

Compute the complex-valued frequency spectrum of $x$. Data is padded to a length of $pad_to$ and the windowing function $window$ is applied to the signal.

**Parameters**

- **x**: 1-D array or sequence
  
  Array or sequence containing the data

- **Fs**: scalar
  
  The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.

- **window**: callable or ndarray
  
  A function or a vector of length $NFFT$. To create window vectors see window_hanning(), window_none(), numpy.blackman(), numpy.hanning(), numpy.bartlett(), scipy.signal(), scipy.signal.get_window(), etc. The default is window_hanning(). If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

- **sides**: [‘default’ | ‘onesided’ | ‘twosided’]
  
  Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. ‘onesided’ forces the return of a one-sided spectrum, while ‘twosided’ forces two-sided.

- **pad_to**: integer
  
  The number of points to which the data segment is padded when performing the FFT. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the $n$ parameter in the call to fft(). The default is None, which sets $pad_to$ equal to the length of the input signal (i.e. no padding).

**Returns**

- **spectrum**: 1-D array
  
  The values for the complex spectrum (complex valued)

- **freqs**: 1-D array
  
  The frequencies corresponding to the elements in spectrum

See also:

**magnitude_spectrum()** magnitude_spectrum() returns the absolute value of this function.
angle_spectrum() angle_spectrum() returns the angle of this function.
phase_spectrum() phase_spectrum() returns the phase (unwrapped angle) of this function.
specgram() specgram() can return the complex spectrum of segments within the signal.

matplotlib.mlab.contiguous_regions(mask)
return a list of (ind0, ind1) such that mask[ind0:ind1].all() is True and we cover all such regions

matplotlib.mlab.cross_from_above(x, threshold)
return the indices into x where x crosses some threshold from below, e.g., the i’s where:

\[
x[i-1]>\text{threshold and } x[i] \leq \text{threshold}
\]

See also:
cross_from_below()

matplotlib.mlab.cross_from_below(x, threshold)
return the indices into x where x crosses some threshold from below, e.g., the i’s where:

\[
x[i-1]<\text{threshold and } x[i] \geq \text{threshold}
\]

Example code:

```python
import matplotlib.pyplot as plt

import matplotlib.pyplot as plt

import matplotlib.pyplot as plt

t = np.arange(0.0, 2.0, 0.1)
s = np.sin(2*np.pi*t)

fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(t, s, '-o')
ax.axhline(0.5)
ax.axhline(-0.5)

ind = cross_from_below(s, 0.5)
ax.vlines(t[ind], -1, 1)

ind = cross_from_above(s, -0.5)
ax.vlines(t[ind], -1, 1)

plt.show()
```

See also:
cross_from_above()

matplotlib.mlab.csd(x, y, NFFT=None, Fs=None, detrend=None, window=None, noverlap=None, pad_to=None, sides=None, scale_by_freq=None)
Compute the cross-spectral density.

Call signature:
The cross spectral density $P_{xy}$ by Welch’s average periodogram method. The vectors $x$ and $y$ are divided into $NFFT$ length segments. Each segment is detrended by function `detrend` and windowed by function `window`. `noverlap` gives the length of the overlap between segments. The product of the direct FFTs of $x$ and $y$ are averaged over each segment to compute $P_{xy}$, with a scaling to correct for power loss due to windowing.

If len($x$) < $NFFT$ or len($y$) < $NFFT$, they will be zero padded to $NFFT$.

**Parameters** $x, y$ : 1-D arrays or sequences

Arrays or sequences containing the data

$Fs$ : scalar

The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.

`window` : callable or ndarray

A function or a vector of length $NFFT$. To create window vectors see `window_hanning()`, `window_none()`, `numpy.blackman()`, `numpy.hamming()`, `numpy.bartlett()`, `scipy.signal.hanning()`, `scipy.signal.get_window()`, etc. The default is `window_hanning()`. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

`sides` : [‘default’, ‘onesided’, ‘twosided’]

Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. ‘onesided’ forces the return of a one-sided spectrum, while ‘twosided’ forces two-sided.

`pad_to` : integer

The number of points to which the data segment is padded when performing the FFT. This can be different from $NFFT$, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the $n$ parameter in the call to `fft()`. The default is None, which sets `pad_to` equal to $NFFT$

$NFFT$ : integer

The number of data points used in each block for the FFT. A power 2 is most efficient. The default value is 256. This should NOT be used to get zero padding, or the scaling of the result will be incorrect. Use `pad_to` for this instead.

`detrend` : {‘default’, ‘constant’, ‘mean’, ‘linear’, ‘none’} or callable
The function applied to each segment before fft-ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the `detrend` parameter is a vector, in matplotlib it is a function. The `pylab` module defines `detrend_none()`, `detrend_mean()`, and `detrend_linear()`, but you can use a custom function as well. You can also use a string to choose one of the functions. ‘default’, ‘constant’, and ‘mean’ call `detrend_mean()`. ‘linear’ calls `detrend_linear()`. ‘none’ calls `detrend_none()`.

**scale_by_freq** : boolean, optional

Specifies whether the resulting density values should be scaled by the scaling frequency, which gives density in units of Hz^{-1}. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

**noverlap** : integer

The number of points of overlap between segments. The default value is 0 (no overlap).

**Returns Pxy** : 1-D array

The values for the cross spectrum $P_{xy}$ before scaling (real valued)

**freqs** : 1-D array

The frequencies corresponding to the elements in $P_{xy}$

**See also:**

`psd()` is the equivalent to setting y=x.

**References**


```python
matplotlib.mlab.csv2rec(fname, comments='#', skiprows=0, checkrows=0, delimiter=’,’, converterd=’None’, names=’None’, missing=’’, missingd=’None’, use_mrecords=’False’, dayfirst=’False’, yearfirst=’False’)
```

Load data from comma/space/tab delimited file in `fname` into a numpy record array and return the record array.

If `names` is `None`, a header row is required to automatically assign the recarray names. The headers will be lower cased, spaces will be converted to underscores, and illegal attribute name characters removed. If `names` is not `None`, it is a sequence of names to use for the column names. In this case, it is assumed there is no header row.

- **fname**: can be a filename or a file handle. Support for gzipped files is automatic, if the filename ends in `.gz`
- **comments**: the character used to indicate the start of a comment in the file, or `None` to switch off the removal of comments
- **skiprows**: is the number of rows from the top to skip
• **checkrows**: is the number of rows to check to validate the column data type. When set to zero all rows are validated.

• **converterd**: if not None, is a dictionary mapping column number or munged column name to a converter function.

• **names**: if not None, is a list of header names. In this case, no header will be read from the file

• **missingd**: is a dictionary mapping munged column names to field values which signify that the field does not contain actual data and should be masked, e.g., ‘0000-00-00’ or ‘unused’

• **missing**: a string whose value signals a missing field regardless of the column it appears in

• **use_mrecords**: if True, return an mrecords.fromrecords record array if any of the data are missing

• **dayfirst**: default is False so that MM-DD-YY has precedence over DD-MM-YY. See http://labix.org/python-dateutil#head-b95ce2094d189a89f80f5ae52a05b4ab7b41af47 for further information.

• **yearfirst**: default is False so that MM-DD-YY has precedence over YY-MM-DD. See http://labix.org/python-dateutil#head-b95ce2094d189a89f80f5ae52a05b4ab7b41af47 for further information.

If no rows are found, None is returned – see examples/loadrec.py

```python
matplotlib.mlab.csvformat_factory(format)
```

```python
matplotlib.mlab.demean(x, axis=0)
```

Return x minus its mean along the specified axis.

**Parameters**

- **x**: array or sequence
  
  Array or sequence containing the data. Can have any dimensionality

- **axis**: integer
  
  The axis along which to take the mean. See numpy.mean for a description of this argument.

**See also:**

delinear()

denone() delinear() and denone() are other detrend algorithms.

detrend_mean() This function is the same as detrend_mean() except for the default axis.

```python
matplotlib.mlab.detrend(x, key=None, axis=None)
```

Return x with its trend removed.

**Parameters**

- **x**: array or sequence
  
  Array or sequence containing the data.

- **key**: [‘default’ | ‘constant’ | ‘mean’ | ‘linear’ | ‘none’] or function
  
  Specifies the detrend algorithm to use. ‘default’ is ‘mean’, which is the same as detrend_mean(). ‘constant’ is the same. ‘linear’ is the same as
detrend_linear(). ‘none’ is the same as detrend_none(). The default is ‘mean’. See the corresponding functions for more details regarding the algorithms. Can also be a function that carries out the detrend operation.

axis : integer
The axis along which to do the detrending.

See also:

detrend_mean() detrend_mean() implements the ‘mean’ algorithm.
detrend_linear() detrend_linear() implements the ‘linear’ algorithm.
detrend_none() detrend_none() implements the ‘none’ algorithm.

Matplotlib.mlab.detrend_linear(y)
Return x minus best fit line; ‘linear’ detrending.

Parameters y : 0-D or 1-D array or sequence
 Array or sequence containing the data

axis : integer
The axis along which to take the mean. See numpy.mean for a description of this argument.

See also:

delinear() This function is the same as delinear() except for the default axis.

detrend_mean()
detrend_none() detrend_mean() and detrend_none() are other detrend algorithms.
detrend() detrend() is a wrapper around all the detrend algorithms.

Matplotlib.mlab.detrend_mean(x, axis=None)
Return x minus the mean(x).

Parameters x : array or sequence
 Array or sequence containing the data Can have any dimensionality

axis : integer
The axis along which to take the mean. See numpy.mean for a description of this argument.

See also:

demean() This function is the same as demean() except for the default axis.
detrend_linear()
detrend_none() detrend_linear() and detrend_none() are other detrend algorithms.
**detrend()** is a wrapper around all the detrend algorithms.

```python
matplotlib.mlab.detrend_none(x, axis=None)
```

Return x: no detrending.

**Parameters**
- **x**: any object
  - An object containing the data
- **axis**: integer
  - This parameter is ignored. It is included for compatibility with detrend_mean

**See also:**
- **detrend_none()** This function is the same as detrend_none() except for the default axis, which has no effect.
- **detrend_mean()**
- **detrend_linear()** detrend_mean() and detrend_linear() are other detrend algorithms.
- **detrend()** detrend() is a wrapper around all the detrend algorithms.

```python
matplotlib.mlab.dist(x, y)
```

Return the distance between two points.

```python
matplotlib.mlab.dist_point_to_segment(p, s0, s1)
```

Get the distance of a point to a segment.

- **p, s0, s1** are xy sequences

This algorithm from [http://geomalgorithms.com/a02-_lines.html](http://geomalgorithms.com/a02-_lines.html)

```python
matplotlib.mlab.distances_along_curve(X)
```

Computes the distance between a set of successive points in \( N \) dimensions.

Where \( X \) is an \( M \times N \) array or matrix. The distances between successive rows is computed. Distance is the standard Euclidean distance.

```python
matplotlib.mlab.donething_callback(*args)
```

```python
matplotlib.mlab.entropy(y, bins)
```

Return the entropy of the data in \( y \) in units of nat.

\[
- \sum p_i \ln(p_i)
\] (39.2)

where \( p_i \) is the probability of observing \( y \) in the \( i^{th} \) bin of \( bins \). \( bins \) can be a number of bins or a range of bins; see numpy.histogram().

Compare \( S \) with analytic calculation for a Gaussian:

```python
x = mu + sigma * randn(200000)
Sanalytic = 0.5 * ( 1.0 + \log(2*pi*sigma**2.0) )
```
matplotlib.mlab.exp_safe(x)
Compute exponentials which safely underflow to zero.
Slow, but convenient to use. Note that numpy provides proper floating point exception handling with access to the underlying hardware.

matplotlib.mlab.fftsurr(x, detrend=<function detrend_none>, window=<function window_none>)
Compute an FFT phase randomized surrogate of x.

matplotlib.mlab.find(condition)
Return the indices where ravel(condition) is true

matplotlib.mlab.frange([start], stop[, step, keywords]) → array of floats
Return a numpy ndarray containing a progression of floats. Similar to numpy.arange(), but defaults to a closed interval.
frange(x0, x1) returns [x0, x0+1, x0+2, ..., x1]: start defaults to 0, and the endpoint is included. This behavior is different from that of range() and numpy.arange(). This is deliberate, since frange() will probably be more useful for generating lists of points for function evaluation, and endpoints are often desired in this use. The usual behavior of range() can be obtained by setting the keyword closed = 0, in this case, frange() basically becomes :func:numpy.arange'.
When step is given, it specifies the increment (or decrement). All arguments can be floating point numbers.
frange(x0,x1,d) returns [x0,x0+d,x0+2d,...,xfin] where xfin <= x1.
frange() can also be called with the keyword npts. This sets the number of points the list should contain (and overrides the value step might have been given). numpy.arange() doesn’t offer this option.

Examples:

```python
>>> frange(3)
array([ 0., 1., 2., 3.])
>>> frange(3,closed=0)
array([ 0., 1., 2.])
>>> frange(1,6,2)
array([1, 3, 5]) or 1,3,5,7, depending on floating point vagueries
>>> frange(1,6.5,npts=5)
array([ 1. , 2.375, 3.75 , 5.125, 6.5 ])
```

matplotlib.mlab.get_formatd(r, formatd=None)
build a formatd guaranteed to have a key for every dtype name

matplotlib.mlab.get_sparse_matrix(M, N, frac=0.1)
Return a M x N sparse matrix with frac elements randomly filled.

matplotlib.mlab.get_xyz_where(Z, Cond)
Z and Cond are M x N matrices. Z are data and Cond is a boolean matrix where some condition is satisfied. Return value is (x, y, z) where x and y are the indices into Z and z are the values of Z at those indices. x, y, and z are 1D arrays.
Interpolates from a nonuniformly spaced grid to some other grid. Fits a surface of the form \( z = f(x, y) \) to the data in the (usually) nonuniformly spaced vectors \((x, y, z)\), then interpolates this surface at the points specified by \((x_i, y_i)\) to produce \(z_i\).

**Parameters**
- \(x, y, z\) : 1d array_like
  Coordinates of grid points to interpolate from.
- \(x_i, y_i\) : 1d or 2d array_like
  Coordinates of grid points to interpolate to.
- **interp** : string key from \{'nn', 'linear'\}
  Interpolation algorithm, either \('nn'\) for natural neighbor, or \('linear'\) for linear interpolation.

**Returns**
- 2d float array
  Array of values interpolated at \((x_i, y_i)\) points. Array will be masked is any of \((x_i, y_i)\) are outside the convex hull of \((x, y)\).

**Notes**

If \(\text{interp}\) is \('nn'\) (the default), uses natural neighbor interpolation based on Delaunay triangulation. This option is only available if the mpl_toolkits.natgrid module is installed. This can be downloaded from https://github.com/matplotlib/natgrid. The \((x_i, y_i)\) grid must be regular and monotonically increasing in this case.

If \(\text{interp}\) is \('linear'\), linear interpolation is used via matplotlib.tri.LinearTriInterpolator.

Instead of using `griddata`, more flexible functionality and other interpolation options are available using a matplotlib.tri.Triangulation and a matplotlib.tri.TriInterpolator.

### matplotlib.mlab.identity

**Parameters**
- \(n, rank=2, dtype='l', typecode=None\)
  Returns the identity matrix of shape \((n, n, ..., n)\) (rank \(r\)).

For ranks higher than 2, this object is simply a multi-index Kronecker delta:

\[
\begin{array}{ll}
\text{id}[i_0, i_1, ..., i_R] = & 1 \quad \text{if } i_0 = i_1 = \ldots = i_R, \\
& 0 \quad \text{otherwise}.
\end{array}
\]

Optionally a \(\text{dtype}\) (or typecode) may be given (it defaults to \('l'\)).

Since rank defaults to 2, this function behaves in the default case (when only \(n\) is given) like numpy. `identity(n)` – but surprisingly, it is much faster.

### matplotlib.mlab.inside_poly

**Parameters**
- \(points, verts\)
  \(points\) is a sequence of \(x, y\) points. \(verts\) is a sequence of \(x, y\) vertices of a polygon.

Return value is a sequence of indices into points for the points that are inside the polygon.
**Matplotlib, Release 2.0.2**

**matplotlib.mlab.is_closed_polygon(X)**
Tests whether first and last object in a sequence are the same. These are presumably coordinates on a polygonal curve, in which case this function tests if that curve is closed.

**matplotlib.mlab.ispower2(n)**
Returns the log base 2 of $n$ if $n$ is a power of 2, zero otherwise.

Note the potential ambiguity if $n == 1$: $2**0 == 1$, interpret accordingly.

**matplotlib.mlab.isvector(X)**
Like the MATLAB function with the same name, returns True if the supplied numpy array or matrix $X$ looks like a vector, meaning it has a one non-singleton axis (i.e., it can have multiple axes, but all must have length 1, except for one of them).

If you just want to see if the array has 1 axis, use X.ndim == 1.

**matplotlib.mlab.linorm(a)**
Return the $l1$ norm of $a$, flattened out.

Implemented as a separate function (not a call to norm() for speed).

**matplotlib.mlab.l2norm(a)**
Return the $l2$ norm of $a$, flattened out.

Implemented as a separate function (not a call to norm() for speed).

**matplotlib.mlab.less_simple_linear_interpolation(x, y, xi, extrap=False)**
This function provides simple (but somewhat less so than cbook.simple_linear_interpolation()) linear interpolation. simple_linear_interpolation() will give a list of point between a start and an end, while this does true linear interpolation at an arbitrary set of points.

This is very inefficient linear interpolation meant to be used only for a small number of points in relatively non-intensive use cases. For real linear interpolation, use scipy.

**matplotlib.mlab.log2(x, ln2=0.6931471805599453)**
Return the log($x$) in base 2.

This is a _slow_ function but which is guaranteed to return the correct integer value if the input is an integer exact power of 2.

**matplotlib.mlab.logspace(xmin, xmax, N)**
Return N values logarithmically spaced between xmin and xmax.

**matplotlib.mlab.longest_contiguous_ones(x)**
Return the indices of the longest stretch of contiguous ones in $x$, assuming $x$ is a vector of zeros and ones. If there are two equally long stretches, pick the first.

**matplotlib.mlab.longest_ones(x)**
alias for longest_contiguous_ones

**matplotlib.mlab.magnitude_spectrum(x, Fs=None, window=None, pad_to=None, sides=None)**
Compute the magnitude (absolute value) of the frequency spectrum of $x$. Data is padded to a length of pad_to and the windowing function window is applied to the signal.
Parameters $x$ : 1-D array or sequence

Array or sequence containing the data

$Fs$ : scalar

The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, $freqs$, in cycles per time unit. The default value is 2.

$window$ : callable or ndarray

A function or a vector of length $NFFT$. To create window vectors see $\text{window}_\text{hanning}()$, $\text{window}_\text{none}()$, $\text{numpy}_\text{blackman}()$, $\text{numpy}_\text{hamming}()$, $\text{numpy}_\text{bartlett}()$, $\text{scipy}_\text{signal}()$, $\text{scipy}_\text{signal}_\text{get}_\text{window}()$, etc. The default is $\text{window}_\text{hanning}()$. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

$sides$ : [‘default’ | ‘onesided’ | ‘twosided’ ]

Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. ‘onesided’ forces the return of a one-sided spectrum, while ‘twosided’ forces two-sided.

$pad\_to$ : integer

The number of points to which the data segment is padded when performing the FFT. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the $n$ parameter in the call to $\text{fft}()$. The default is None, which sets $pad\_to$ equal to the length of the input signal (i.e. no padding).

Returns $spectrum$ : 1-D array

The values for the magnitude spectrum (real valued)

$freqs$ : 1-D array

The frequencies corresponding to the elements in $spectrum$

See also:

$\text{psd()}$ $\text{psd()}$ returns the power spectral density.

$\text{complex\_spectrum()}$ This function returns the absolute value of $\text{complex\_spectrum()}$.

$\text{angle\_spectrum()}$ $\text{angle\_spectrum()}$ returns the angles of the corresponding frequencies.

$\text{phase\_spectrum()}$ $\text{phase\_spectrum()}$ returns the phase (unwrapped angle) of the corresponding frequencies.

$\text{specgram()}$ $\text{specgram()}$ can return the magnitude spectrum of segments within the signal.

$\text{matplotlib.mlab.movav}(x, n)$

Compute the $len(n)$ moving average of $x$. 

39.1. $\text{matplotlib.mlab}$ 1299
matplotlib.mlab.norm_flat\(a, p=2\)

\[
\text{norm}(a, p=2) \rightarrow 1-p \text{ norm of } a.\text{flat}
\]

Return the l-p norm of \(a\), considered as a flat array. This is NOT a true matrix norm, since arrays of arbitrary rank are always flattened.

\(p\) can be a number or the string ‘Infinity’ to get the L-infinity norm.

matplotlib.mlab.normpdf\(x, *\text{args}\)

Return the normal pdf evaluated at \(x\); \text{args} provides \(\mu, \sigma\)

matplotlib.mlab.offset_line\(y, yerr\)

Offsets an array \(y\) by +/- an error and returns a tuple \((y - \text{err}, y + \text{err})\).

The error term can be:

- A scalar. In this case, the returned tuple is obvious.
- A vector of the same length as \(y\). The quantities \(y +/- \text{err}\) are computed component-wise.
- A tuple of length 2. In this case, \(\text{yerr}[0]\) is the error below \(y\) and \(\text{yerr}[1]\) is error above \(y\). For example:

```python
from pylab import *

x = linspace(0, 2*pi, num=100, endpoint=True)
y = sin(x)
y_minus, y_plus = mlab.offset_line(y, 0.1)
plot(x, y)
fill_between(x, ym, y2=yp)
show()
```

matplotlib.mlab.path_length\(X\)

Computes the distance travelled along a polygonal curve in \(N\) dimensions.

Where \(X\) is an \(M \times N\) array or matrix. Returns an array of length \(M\) consisting of the distance along the curve at each point (i.e., the rows of \(X\)).

matplotlib.mlab.phase_spectrum\(x, Fs=None, window=None, pad_to=None, sides=None\)

Compute the phase of the frequency spectrum (unwrapped angle spectrum) of \(x\). Data is padded to a length of \(pad\_to\) and the windowing function \(window\) is applied to the signal.

**Parameters**

- \(x\) : 1-D array or sequence
  - Array or sequence containing the data
- \(Fs\) : scalar
  - The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, \(freqs\), in cycles per time unit. The default value is 2.
- \(window\) : callable or ndarray
  - A function or a vector of length \(N\_FFT\). To create window vectors see \(\text{window\_hanning()}\), \(\text{window\_none()}\), \(\text{numpy\_blackman()}\), \(\text{numpy\_hamming()}\), \(\text{numpy\_bartlett()}\), \(\text{scipy\_signal()}\), \(\text{scipy\_signal\_get\_window()}\), etc. The default is \(\text{window\_hanning()}\). If a function is
passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

**sides** : [ ‘default’ | ‘onesided’ | ‘twosided’ ]

Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. ‘onesided’ forces the return of a one-sided spectrum, while ‘twosided’ forces two-sided.

**pad_to** : integer

The number of points to which the data segment is padded when performing the FFT. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the n parameter in the call to fft(). The default is None, which sets pad_to equal to the length of the input signal (i.e. no padding).

**Returns spectrum** : 1-D array

The values for the phase spectrum in radians (real valued)

**freqs** : 1-D array

The frequencies corresponding to the elements in spectrum

See also:

- `complex_spectrum()` This function returns the angle value of `complex_spectrum()`.
- `magnitude_spectrum()` returns the magnitudes of the corresponding frequencies.
- `angle_spectrum()` returns the wrapped version of this function.
- `specgram()` can return the phase spectrum of segments within the signal.

(matplotlib.mlab.polygon_below(xmin, xs, ys)

Given a sequence of xs and ys, return the vertices of a polygon that has a horizontal base at xmin and an upper bound at the ys. xmin is a scalar.

Intended for use with `matplotlib.axes.Axes.fill()`, e.g.:

```python
xv, yv = polygon_below(0, x, y)
ax.fill(xv, yv)
```

(matplotlib.mlab.polygon_between(x, ylower, yupper)

Given a sequence of x, ylower and yupper, return the polygon that fills the regions between them. ylower or yupper can be scalar or iterable. If they are iterable, they must be equal in length to x.

Return value is x, y arrays for use with `matplotlib.axes.Axes.fill()`.

(matplotlib.mlab.prctile(x, p=[0.0, 25.0, 50.0, 75.0, 100.0])

Return the percentiles of x. p can either be a sequence of percentile values or a scalar. If p is a
sequence, the $i$th element of the return sequence is the $p^{(i)}$-th percentile of $x$. If $p$ is a scalar, the largest value of $x$ less than or equal to the $p$ percentage point in the sequence is returned.

**matplotlib.mlab.prctile_rank**(x, p)

Return the rank for each element in $x$, return the rank 0..len(p). e.g., if $p = (25, 50, 75)$, the return value will be a len(x) array with values in [0,1,2,3] where 0 indicates the value is less than the 25th percentile, 1 indicates the value is $\geq$ the 25th and $< 50$th percentile, ... and 3 indicates the value is above the 75th percentile cutoff.

$p$ is either an array of percentiles in [0..100] or a scalar which indicates how many quantiles of data you want ranked.

**matplotlib.mlab.psd**(x, NFFT=None, Fs=2, detrend=mlab.detrend_none, window=None, noverlap=None, pad_to=None, sides='default', scale_by_freq=None)

Compute the power spectral density.

Call signature:

```python
psd(x, NFFT=256, Fs=2, detrend=mlab.detrend_none,
    window=mlab.window_hanning, noverlap=0, pad_to=None,
    sides='default', scale_by_freq=None)
```

The power spectral density $P_{xx}$ by Welch’s average periodogram method. The vector $x$ is divided into $NFFT$ length segments. Each segment is detrended by function detrend and windowed by function window. noverlap gives the length of the overlap between segments. The $|fft(i)|^2$ of each segment $i$ are averaged to compute $P_{xx}$.

If len(x) $< NFFT$, it will be zero padded to $NFFT$.

**Parameters**

- **x**: 1-D array or sequence
  
  Array or sequence containing the data

- **Fs**: scalar
  
  The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.

- **window**: callable or ndarray
  
  A function or a vector of length NFFT. To create window vectors see `window_hanning()`, `window_none()`, `numpy.blackman()`, `numpy.hamming()`, `numpy.bartlett()`, `scipy.signal`, `scipy.signal.get_window()`, etc. The default is `window_hanning()`. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

- **sides**: [‘default’ | ‘onesided’ | ‘twosided’]
  
  Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. ‘onesided’ forces the return of a one-sided spectrum, while ‘twosided’ forces two-sided.

- **pad_to**: integer
The number of points to which the data segment is padded when performing the FFT. This can be different from NFFT, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the n parameter in the call to fft(). The default is None, which sets pad_to equal to NFFT.

NFFT : integer
The number of data points used in each block for the FFT. A power 2 is most efficient. The default value is 256. This should NOT be used to get zero padding, or the scaling of the result will be incorrect. Use pad_to for this instead.

detrend : {'default', 'constant', 'mean', 'linear', 'none'} or callable
The function applied to each segment before fft-ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the detrend parameter is a vector, in matplotlib it is a function. The pylab module defines detrend_none(), detrend_mean(), and detrend_linear(), but you can use a custom function as well. You can also use a string to choose one of the functions. ‘default’, ‘constant’, and ‘mean’ call detrend_mean(). ‘linear’ calls detrend_linear(). ‘none’ calls detrend_none().

scale_by_freq : boolean, optional
Specifies whether the resulting density values should be scaled by the scaling frequency, which gives density in units of Hz^-1. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

noverlap : integer
The number of points of overlap between segments. The default value is 0 (no overlap).

Returns Pxx : 1-D array
The values for the power spectrum P_{xx} (real valued)

freqs : 1-D array
The frequencies corresponding to the elements in Pxx

See also:

specgram() specgram() differs in the default overlap; in not returning the mean of the segment periodograms; and in returning the times of the segments.
magnitude_spectrum() magnitude_spectrum() returns the magnitude spectrum.
csd() csd() returns the spectral density between two signals.
References


`matplotlib.mlab.quad2cubic(q0x, q0y, q1x, q1y, q2x, q2y)`

Converts a quadratic Bezier curve to a cubic approximation.

The inputs are the x and y coordinates of the three control points of a quadratic curve, and the output is a tuple of x and y coordinates of the four control points of the cubic curve.

`matplotlib.mlab.rec2csv(r, fname, delimiter=',', formatd=None, missing='', missingd=None, withheader=True)`

Save the data from numpy recarray `r` into a comma/space/tab-delimited file. The record array dtype names will be used for column headers.

`fname`: can be a filename or a file handle. Support for gzipped files is automatic, if the filename ends in `.gz`

`withheader`: if withheader is False, do not write the attribute names in the first row

for formatd type FormatFloat, we override the precision to store full precision floats in the CSV file

See also:

`csv2rec()` For information about `missing` and `missingd`, which can be used to fill in masked values into your CSV file.

`matplotlib.mlab.rec2txt(r, header=None, padding=3, precision=3, fields=None)`

Returns a textual representation of a record array.

**Parameters r: numpy recarray**

- `header`: list
  column headers

- `padding`: space between each column

- `precision`: number of decimal places to use for floats.
  Set to an integer to apply to all floats. Set to a list of integers to apply precision individually. Precision for non-floats is simply ignored.

- `fields`: list
  If not None, a list of field names to print. fields can be a list of strings like `[‘field1’, ‘field2’]` or a single comma separated string like `‘field1,field2’`

**Examples**

For `precision=[0,2,3]`, the output is
matplotlib.mlab.rec_append_fields(rec, names, arrs, dtypes=None)
Return a new record array with field names populated with data from arrays in arrs. If appending a single field, then names, arrs and dtypes do not have to be lists. They can just be the values themselves.

matplotlib.mlab.rec_drop_fields(rec, names)
Return a new numpy record array with fields in names dropped.

matplotlib.mlab.rec_groupby(r, groupby, stats)
r is a numpy record array
groupby is a sequence of record array attribute names that together form the grouping key. e.g., ('date', 'productcode')
stats is a sequence of (attr, func, outname) tuples which will call \( x = func(attr) \) and assign \( x \) to the record array output with attribute outname. For example:

\[
\text{stats} = (\ ('sales', \ \text{len}, \ 'numsales'), \ ('sales', \ \text{np.mean}, \ 'avgsale') )
\]

Return record array has dtype names for each attribute name in the groupby argument, with the associated group values, and for each outname name in the stats argument, with the associated stat summary output.

matplotlib.mlab.rec_join(key, r1, r2, jointype='inner', defaults=None, r1postfix='1', r2postfix='2')
Join record arrays r1 and r2 on key; key is a tuple of field names – if key is a string it is assumed to be a single attribute name. If r1 and r2 have equal values on all the keys in the key tuple, then their fields will be merged into a new record array containing the intersection of the fields of r1 and r2.

r1 (also r2) must not have any duplicate keys.

The jointype keyword can be 'inner', 'outer', 'leftouter'. To do a rightouter join just reverse r1 and r2.

The defaults keyword is a dictionary filled with \{column_name:default_value\} pairs.

The keywords r1postfix and r2postfix are postfixed to column names (other than keys) that are both in r1 and r2.

matplotlib.mlab.rec_keep_fields(rec, names)
Return a new numpy record array with only fields listed in names

matplotlib.mlab.rec_summarize(r, summaryfuncs)
r is a numpy record array
summaryfuncs is a list of (attr, func, outname) tuples which will apply func to the array r*[attr] and assign the output to a new attribute name *outname. The returned record array is identical to r, with extra arrays for each element in summaryfuncs.
matplotlib.mlab.recs_join(key, name, recs, jointype='outer', missing=0.0, postfixes=None)

Join a sequence of record arrays on single column key.

This function only joins a single column of the multiple record arrays

*key* is the column name that acts as a key

*name* is the name of the column that we want to join

*recs* is a list of record arrays to join

*jointype* is a string ‘inner’ or ‘outer’

*missing* is what any missing field is replaced by

*postfixes* if not None, a len recs sequence of postfixes

returns a record array with columns [rowkey, name0, name1, ... namen-1]. or if postfixes [PF0, PF1, ..., PFN-1] are supplied, [rowkey, namePF0, namePF1, ... namePFN-1].

Example:

```python
r = recs_join("date", "close", recs=[r0, r1], missing=0.0)
```

matplotlib.mlab.rk4(derivs, y0, t)

Integrate 1D or ND system of ODEs using 4-th order Runge-Kutta. This is a toy implementation which may be useful if you find yourself stranded on a system w/o scipy. Otherwise use scipy.integrate().

**Parameters**

* y0
  
  initial state vector

* t
  
  sample times

* derivs
  
  returns the derivative of the system and has the signature dy = derivs(yi, ti)

**Examples**

A 2D system:

```python
def derivs6(x, t):
    d1 = x[0] + 2*x[1]
    d2 = -3*x[0] + 4*x[1]
    return (d1, d2)
dt = 0.0005
t = arange(0.0, 2.0, dt)
y0 = (1,2)
yout = rk4(derivs6, y0, t)
```
A 1D system:

```python
alpha = 2
def derivs(x,t):
    return -alpha*x + exp(-t)

y0 = 1
yout = rk4(derivs, y0, t)
```

If you have access to scipy, you should probably be using the scipy.integrate tools rather than this function.

```python
matplotlib.mlab.rms_flat(a)
```

Return the root mean square of all the elements of a, flattened out.

```python
matplotlib.mlab.safe_isinf(x)
numpy.isinf() for arbitrary types
```

```python
matplotlib.mlab.safe_isnan(x)
numpy.isnan() for arbitrary types
```

```python
matplotlib.mlab.segments_intersect(s1, s2)
```

Return True if s1 and s2 intersect. s1 and s2 are defined as:

```plaintext
s1: (x1, y1), (x2, y2)
s2: (x3, y3), (x4, y4)
```

```python
matplotlib.mlab.slopes(x, y)
slopes() calculates the slope y'(x)
```

The slope is estimated using the slope obtained from that of a parabola through any three consecutive points.

This method should be superior to that described in the appendix of A CONSISTENTLY WELL BEHAVED METHOD OF INTERPOLATION by Russel W. Stineman (Creative Computing July 1980) in at least one aspect:

Circles for interpolation demand a known aspect ratio between x- and y-values. For many functions, however, the abscissa are given in different dimensions, so an aspect ratio is completely arbitrary.

The parabola method gives very similar results to the circle method for most regular cases but behaves much better in special cases.

Norbert Nemec, Institute of Theoretical Physics, University or Regensburg, April 2006 Norbert.Nemec atphysik.uni-regensburg.de

(inspired by a original implementation by Halldor Bjornsson, Icelandic Meteorological Office, March 2006 halldor at vedur.is)

```python
matplotlib.mlab.specgram(x, NFFT=None, Fs=None, detrend=None, window=None, noverlap=None, pad_to=None, sides=None, scale_by_freq=None, mode=None)
```

Compute a spectrogram.
Compute and plot a spectrogram of data in \( x \). Data are split into NFFT length segments and the spectrum of each section is computed. The windowing function window is applied to each segment, and the amount of overlap of each segment is specified with noverlap.

**Parameters**

\( x \) : array_like

1-D array or sequence.

\( Fs \) : scalar

The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.

\( window \) : callable or ndarray

A function or a vector of length \( NFFT \). To create window vectors see `window_hanning()`, `window_none()`, `numpy.blackman()`, `numpy.hanning()`, `numpy.bartlett()`, `scipy.signal()`, etc. The default is `window_hanning()`. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

\( sides \) : [ 'default' | 'onesided' | 'twosided' ]

Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. ‘onesided’ forces the return of a one-sided spectrum, while ‘twosided’ forces two-sided.

\( pad_to \) : integer

The number of points to which the data segment is padded when performing the FFT. This can be different from \( NFFT \), which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the \( n \) parameter in the call to `fft()`. The default is None, which sets \( pad_to \) equal to \( NFFT \).

\( NFFT \) : integer

The number of data points used in each block for the FFT. A power 2 is most efficient. The default value is 256. This should NOT be used to get zero padding, or the scaling of the result will be incorrect. Use \( pad_to \) for this instead.

\( detrend \) : {‘default’, ‘constant’, ‘mean’, ‘linear’, ‘none’} or callable

The function applied to each segment before fft-ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the \( detrend \) parameter is a vector, in matplotlib is it a function. The pylvab module defines `detrend_none()`, `detrend_mean()`, and `detrend_linear()`, but you can use a custom function as well. You can also use a string to choose one of the functions. ‘default’, ‘constant’, and ‘mean’ call `detrend_mean()`. ‘linear’ calls `detrend_linear()`. ‘none’ calls `detrend_none()`.
scale_by_freq : boolean, optional

Specifies whether the resulting density values should be scaled by the scaling frequency, which gives density in units of Hz^-1. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

nooverlap : int, optional

The number of points of overlap between blocks. The default value is 128.

mode : str, optional

What sort of spectrum to use, default is ‘psd’.

‘psd’ Returns the power spectral density.

‘complex’ Returns the complex-valued frequency spectrum.

‘magnitude’ Returns the magnitude spectrum.

‘angle’ Returns the phase spectrum without unwrapping.

‘phase’ Returns the phase spectrum with unwrapping.

Returns spectrum : array_like

2-D array, columns are the periodograms of successive segments.

freqs : array_like

1-D array, frequencies corresponding to the rows in spectrum.

t : array_like

1-D array, the times corresponding to midpoints of segments (i.e the columns in spectrum).

See also:

psd differs in the overlap and in the return values.
complex_spectrum similar, but with complex valued frequencies.
magnitude_spectrum similar single segment when mode is ‘magnitude’.
angle_spectrum similar to single segment when mode is ‘angle’.
phase_spectrum similar to single segment when mode is ‘phase’.

Notes

detrend and scale_by_freq only apply when mode is set to ‘psd’.

matplotlib.mlab.stineman_interp(xi, x, y, yp=None)

Given data vectors x and y, the slope vector yp and a new abscissa vector xi, the function stineman_interp() uses Stineman interpolation to calculate a vector yi corresponding to xi.
Here’s an example that generates a coarse sine curve, then interpolates over a finer abscissa:

```python
x = linspace(0,2*pi,20); y = sin(x); yp = cos(x)
x_i = linspace(0,2*pi,40); y_i = stineman_interp(x_i,x,y,yp);
plot(x,y,'o',x_i,y_i)
```

The interpolation method is described in the article A CONSISTENTLY WELL BEHAVED METHOD OF INTERPOLATION by Russell W. Stineman. The article appeared in the July 1980 issue of Creative Computing with a note from the editor stating that while they were:

not an academic journal but once in a while something serious and original comes in adding that this was “apparently a real solution” to a well known problem.

For \( yp = \text{None} \), the routine automatically determines the slopes using the `slopes()` routine.

\( x \) is assumed to be sorted in increasing order.

For values \( x_i[j] < x[0] \) or \( x_i[j] > x[-1] \), the routine tries an extrapolation. The relevance of the data obtained from this, of course, is questionable...

Original implementation by Halldor Bjornsson, Icelandic Meteorolocial Office, March 2006 halldor@vedur.is

Completely reworked and optimized for Python by Norbert Nemec, Institute of Theoretical Physics, University or Regensburg, April 2006 Norbert.Nemec@physik.uni-regensburg.de

```
matplotlib.mlab.stride_repeat(x, n, axis=0)
```

Repeat the values in an array in a memory-efficient manner. Array \( x \) is stacked vertically \( n \) times.

**Warning:** It is not safe to write to the output array. Multiple elements may point to the same piece of memory, so modifying one value may change others.

**Parameters**

- **x**: 1D array or sequence
  Array or sequence containing the data.
- **n**: integer
  The number of time to repeat the array.
- **axis**: integer
  The axis along which the data will run.

**References**

- stackoverflow: Repeat NumPy array without replicating data?

```
matplotlib.mlab.stride_windows(x, n, noverlap=None, axis=0)
```

Get all windows of \( x \) with length \( n \) as a single array, using strides to avoid data duplication.
**Warning:** It is not safe to write to the output array. Multiple elements may point to the same piece of memory, so modifying one value may change others.

**Parameters**

- **x**: 1D array or sequence
  
  Array or sequence containing the data.

- **n**: integer
  
  The number of data points in each window.

- **noverlap**: integer
  
  The overlap between adjacent windows. Default is 0 (no overlap)

- **axis**: integer
  
  The axis along which the windows will run.

**References**


```python
matplotlib.mlab.vector_lengths(X, P=2.0, axis=None)
```

Finds the length of a set of vectors in \( n \) dimensions. This is like the `numpy.norm()` function for vectors, but has the ability to work over a particular axis of the supplied array or matrix.

Computes \((\sum((x_i)^P))^{1/P}\) for each \({x_i}\) being the elements of \(X\) along the given axis. If \(axis\) is `None`, compute over all elements of \(X\).

```python
matplotlib.mlab.window_hanning(x)
```

Return \(x\) times the hanning window of \(\text{len}(x)\).

**See also:**

- `window_none()` [window_none()](https://matplotlib.org/api/mlab_api.html#matplotlib.mlab.window_none) is another window algorithm.

```python
matplotlib.mlab.window_none(x)
```

No window function; simply return \(x\).

**See also:**

- `window_hanning()` [window_hanning()](https://matplotlib.org/api/mlab_api.html#matplotlib.mlab.window_hanning) is another window algorithm.
40.1 matplotlib.offsetbox

The OffsetBox is a simple container artist. The child artist are meant to be drawn at a relative position to its parent. The [VH]Packer, DrawingArea and TextArea are derived from the OffsetBox.

The [VH]Packer automatically adjust the relative positions of their children, which should be instances of the OffsetBox. This is used to align similar artists together, e.g., in legend.

The DrawingArea can contain any Artist as a child. The DrawingArea has a fixed width and height. The position of children relative to the parent is fixed. The TextArea is contains a single Text instance. The width and height of the TextArea instance is the width and height of the its child text.

class matplotlib.offsetbox.AnchoredOffsetbox(loc, pad=0.4, borderpad=0.5, child=None, prop=None, frameon=True, bbox_to_anchor=None, bbox_transform=None, **kwargs)

Bases: matplotlib.offsetbox.OffsetBox

An offset box placed according to the legend location loc. AnchoredOffsetbox has a single child. When multiple children is needed, use other OffsetBox class to enclose them. By default, the offset box is anchored against its parent axes. You may explicitly specify the bbox_to_anchor.

loc is a string or an integer specifying the legend location. The valid location codes are:

<table>
<thead>
<tr>
<th>Location</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Upper right</td>
<td>1</td>
</tr>
<tr>
<td>Upper left</td>
<td>2</td>
</tr>
<tr>
<td>Lower left</td>
<td>3</td>
</tr>
<tr>
<td>Lower right</td>
<td>4</td>
</tr>
<tr>
<td>Right</td>
<td>5</td>
</tr>
<tr>
<td>Center left</td>
<td>6</td>
</tr>
<tr>
<td>Center right</td>
<td>7</td>
</tr>
<tr>
<td>Lower center</td>
<td>8</td>
</tr>
<tr>
<td>Upper center</td>
<td>9</td>
</tr>
<tr>
<td>Center</td>
<td>10</td>
</tr>
</tbody>
</table>

pad [pad around the child for drawing a frame. given in] fraction of fontsize.

borderpad : pad between offsetbox frame and the bbox_to_anchor,
child : OffsetBox instance that will be anchored.
prop : font property. This is only used as a reference for paddings.
frameon : draw a frame box if True.
bbox_to_anchor : bbox to anchor. Use self.axes.bbox if None.
bbox_transform : with which the bbox_to_anchor will be transformed.

draw(renderer)
   draw the artist

get_bbox_to_anchor()
   return the bbox that the legend will be anchored

get_child()
   return the child

get_children()
   return the list of children

get_extent(renderer)
   return the extent of the artist. The extent of the child added with the pad is returned

get_window_extent(renderer)
   get the bounding box in display space.

set_bbox_to_anchor(bbox, transform=None)
   set the bbox that the child will be anchored.

   bbox can be a Bbox instance, a list of [left, bottom, width, height], or a list of [left, bottom]
   where the width and height will be assumed to be zero. The bbox will be transformed to display
   coordinate by the given transform.

set_child(child)
   set the child to be anchored

update_frame(bbox, fontsize=None)

zorder = 5

class matplotlib.offsetbox.AnchoredText(s, loc, pad=0.4, borderpad=0.5, prop=None, **kwargs)

Bases: matplotlib.offsetbox.AnchoredOffsetbox

AnchoredOffsetbox with Text.

Parameters s : string

   Text.
loc : str
   Location code.
pad : float, optional
Pad between the text and the frame as fraction of the font size.

**borderpad** : float, optional
Pad between the frame and the axes (or bbox_to_anchor).

**prop** : matplotlib.font_manager.FontProperties
Font properties.

**Notes**

Other keyword parameters of AnchoredOffsetBox are also allowed.

### class matplotlib.offsetbox.AnnotationBbox

```
offsetbox, xy, xybox=None, xycoords='data',
boxcoords=None, frameon=True, pad=0.4,
annotation_clip=None, box_alignment=(0.5, 0.5), bboxprops=None, arrowprops=None,
```

Bases: matplotlib.artist.Artist, matplotlib.text._AnnotationBase

Annotation-like class, but with offsetbox instead of Text.

- **offsetbox** : OffsetBox instance
- **xycoords** [same as Annotation but can be a tuple of two] strings which are interpreted as x and y coordinates.
- **boxcoords** [similar to textcoords as Annotation but can be a] tuple of two strings which are interpreted as x and y coordinates.
- **box_alignment** [a tuple of two floats for a vertical and] horizontal alignment of the offset box w.r.t. the boxcoords. The lower-left corner is (0.0) and upper-right corner is (1.1).

other parameters are identical to that of Annotation.

**anncoords**

**contains**(event)

**draw**(renderer)

  Draw the Annotation object to the given renderer.

**get_children**( )

**get_fontsize**(s=None)

  return fontsize in points

**set_figure**(fig)

**set_fontsize**(s=None)

  set fontsize in points
update_positions(renderer)
    Update the pixel positions of the annotated point and the text.

xyann

zorder = 3

class matplotlib.offsetbox.AuxTransformBox(aux_transform)
    Bases: matplotlib.offsetbox.OffsetBox
    Offset Box with the aux_transform. Its children will be transformed with the aux_transform first then will be offseted. The absolute coordinate of the aux_transform is meaning as it will be automatically adjust so that the left-lower corner of the bounding box of children will be set to (0,0) before the offset transform.

    It is similar to drawing area, except that the extent of the box is not predetermined but calculated from the window extent of its children. Furthermore, the extent of the children will be calculated in the transformed coordinate.

    add_artist(a)
        Add any Artist to the container box

    draw(renderer)
        Draw the children

    get_extent(renderer)

    get_offset()
        return offset of the container.

    get_transform()
        Return the Transform applied to the children

    get_window_extent(renderer)
        get the bounding box in display space.

    set_offset(xy)
        set offset of the container.

        Accept : tuple of x,y coordinate in disokay units.

    set_transform(t)
        set_transform is ignored.

class matplotlib.offsetbox.DraggableAnnotation(annotation, use_blit=False)
    Bases: matplotlib.offsetbox.DraggableBase

    save_offset()

    update_offset(dx, dy)
class matplotlib.offsetbox.DraggableBase(ref_artist, use_blit=False)
    Bases: object

    helper code for a draggable artist (legend, offsetbox) The derived class must override following two method.

    def saveoffset(self): pass
    def update_offset(self, dx, dy): pass

    `saveoffset` is called when the object is picked for dragging and it is meant to save reference position
    of the artist.

    `update_offset` is called during the dragging. dx and dy is the pixel offset from the point where the
    mouse drag started.

    Optionally you may override following two methods.

    def artist_picker(self, artist, evt): return self.ref_artist.contains(evt)
    def finalize_offset(self): pass

    `artist_picker` is a picker method that will be used. `finalize_offset` is called when the mouse is re-
    leased. In current implementation of DraggableLegend and DraggableAnnotation, `update_offset` places
    the artists simply in display coordinates. And `finalize_offset` recalculate their position in
    the normalized axes coordinate and set a relevant attribute.

    `artist_picker(artist, evt)`

    `disconnect()`
    disconnect the callbacks

    `finalize_offset()`

    `on_motion(event)`

    `on_motion_blit(event)`

    `on_pick(event)`

    `on_release(event)`

    `save_offset()`

    `update_offset(dx, dy)`

class matplotlib.offsetbox.DraggableOffsetBox(ref_artist, offsetbox, use_blit=False)
    Bases: matplotlib.offsetbox.DraggableBase
get_loc_in_canvas()

save_offset()

update_offset(dx, dy)

class matplotlib.offsetbox.DrawingArea(width, height, xdescent=0.0, ydescent=0.0, clip=False)

Bases: matplotlib.offsetbox.OffsetBox

The DrawingArea can contain any Artist as a child. The DrawingArea has a fixed width and height. The position of children relative to the parent is fixed. The children can be clipped at the boundaries of the parent.

width, height : width and height of the container box. xdescent, ydescent : descent of the box in x- and y-direction. clip : Whether to clip the children

add_artist(a)

Add any Artist to the container box

clip_children

If the children of this DrawingArea should be clipped by DrawingArea bounding box.

draw(renderer)

Draw the children

get_extent(renderer)

Return with, height, xdescent, ydescent of box

get_offset()

return offset of the container.

get_transform()

Return the Transform applied to the children

get_window_extent(renderer)

get the bounding box in display space.

set_offset(xy)

set offset of the container.

Accept : tuple of x,y coordinate in disokay units.

set_transform(t)

set_transform is ignored.

class matplotlib.offsetbox.HPacker(pad=None, sep=None, width=None, height=None,

align='baseline', mode='fixed', children=None)

Bases: matplotlib.offsetbox.PackerBase

The HPacker has its children packed horizontally. It automatically adjusts the relative positions of children at draw time.

Parameters pad : float, optional
Boundary pad.

**sep** : float, optional

Spacing between items.

**width** : float, optional

**height** : float, optional

Width and height of the container box, calculated if **None**.

**align** : str

Alignment of boxes.

**mode** : str

Packing mode.

**Notes**

*pad* and *sep* need to be given in points and will be scale with the renderer dpi, while *width* and *height* need to be in pixels.

**get_extent_offsets**(*renderer*)

update offsets of children and return the extents of the box

**class** `matplotlib.offsetbox.OffsetBox`(*args, **kwargs*)

Bases: `matplotlib.artist.Artist`

The OffsetBox is a simple container artist. The child artist are meant to be drawn at a relative position to its parent.

**axes**

The *Axes* instance the artist resides in, or **None**.

**contains**(*mouseevent*)

**draw**(*renderer*)

Update the location of children if necessary and draw them to the given *renderer*.

**get_children**()

Return a list of artists it contains.

**get_extent**(*renderer*)

Return with, height, xdescent, ydescent of box

**get_extent_offsets**(*renderer*)

**get_offset**(*width, height, xdescent, ydescent, renderer*)

Get the offset

accepts extent of the box
get_visible_children()
    Return a list of visible artists it contains.

get_window_extent(renderer)
    get the bounding box in display space.

set_figure(fig)
    Set the figure
        accepts a class: Figure instance

set_height(height)
    Set the height
        accepts float

set_offset(xy)
    Set the offset
        accepts x, y, tuple, or a callable object.

set_width(width)
    Set the width
        accepts float

class matplotlib.offsetbox.OffsetImage(arr, zoom=1, cmap=None, norm=None, interpolation=None, origin=None, filternorm=1, filterrad=4.0, resample=False, dpi_cor=True, **kwargs)
    Bases: matplotlib.offsetbox.OffsetBox

draw(renderer)
    Draw the children

get_children()

get_data()

get_extent(renderer)

get_offset()
    return offset of the container.

get_window_extent(renderer)
    get the bounding box in display space.

get_zoom()

set_data(arr)
```python
set_zoom(zoom)

class matplotlib.offsetbox.PackerBase(pad=None, sep=None, width=None, height=None,
                                          align=None, mode=None, children=None)
Bases: matplotlib.offsetbox.OffsetBox

Parameters pad : float, optional
    Boundary pad.

    sep : float, optional
        Spacing between items.

    width : float, optional
    height : float, optional
        Width and height of the container box, calculated if None.

    align : str, optional
        Alignment of boxes. Can be one of top, bottom, left, right, center and
        baseline

    mode : str, optional
        Packing mode.

Notes

    pad and sep need to given in points and will be scale with the renderer dpi, while width and height
    need to be in pixels.

class matplotlib.offsetbox.PaddedBox(child, pad=None, draw_frame=False,
                                        patch_attrs=None)
Bases: matplotlib.offsetbox.OffsetBox

pad : boundary pad

Note: pad need to given in points and will be scale with the renderer dpi, while width and height
      need to be in pixels.

    draw(renderer)
        Update the location of children if necessary and draw them to the given renderer.

    draw_frame(renderer)

    get_extent_offsets(renderer)
        update offset of childrens and return the extents of the box

    update_frame(bbox, fontsize=None)
```

class `matplotlib.offsetbox.TextArea`(s, `textprops=None`, `multilinebaseline=None`, `minimumdescent=True`)

Bases: `matplotlib.offsetbox.OffsetBox`

The TextArea is contains a single Text instance. The text is placed at (0,0) with baseline+left alignment. The width and height of the TextArea instance is the width and height of the its child text.

**Parameters**

- `s`: str
  a string to be displayed.

- `textprops`: `FontProperties`, optional

- `multilinebaseline`: bool, optional
  If True, baseline for multiline text is adjusted so that it is (approximately) center-aligned with singleline text.

- `minimumdescent`: bool, optional
  If True, the box has a minimum descent of “p”.

**Methods**

- `draw(renderer)`
  Draw the children

- `get_extent(renderer)`

- `get_minimumdescent()`
  get minimumdescent.

- `get_multilinebaseline()`
  get multilinebaseline.

- `get_offset()`
  return offset of the container.

- `get_text()`
  Returns the string representation of this area’s text

- `get_window_extent(renderer)`
  get the bounding box in display space.

- `set_minimumdescent(t)`
  Set minimumdescent.
  If True, extent of the single line text is adjusted so that it has minimum descent of “p”

- `set_multilinebaseline(t)`
  Set multilinebaseline.
  If True, baseline for multiline text is adjusted so that it is (approximately) center-aligned with singleline text.

- `set_offset(xy)`
  set offset of the container.
  Accept : tuple of x,y coordinates in display units.
set_text(s)
Set the text of this area as a string.

set_transform(t)
set_transform is ignored.

class matplotlib.offsetbox.VPacker(pad=None, sep=None, width=None, height=None,
align='baseline', mode='fixed', children=None)
Bases: matplotlib.offsetbox.PackerBase

The VPacker has its children packed vertically. It automatically adjust the relative positions of children in the drawing time.

Parameters

pad : float, optional
  Boundary pad.

sep : float, optional
  Spacing between items.

width : float, optional

height : float, optional
  width and height of the container box, calculated if None.

align : str, optional
  Alignment of boxes.

mode : str, optional
  Packing mode.

Notes

pad and sep need to given in points and will be scale with the renderer dpi, while width and height need to be in pixels.

get_extent_offsets(renderer)
  update offset of childrens and return the extents of the box

matplotlib.offsetbox.bbox_artist(*args, **kwargs)
41.1 matplotlib.patches

```python
class matplotlib.patches.Arc(xy, width, height, angle=0.0, theta1=0.0, theta2=360.0, **kwargs)
```

Bases: `matplotlib.patches.Ellipse`

An elliptical arc. Because it performs various optimizations, it can not be filled.

The arc must be used in an `Axes` instance—it can not be added directly to a `Figure`—because it is optimized to only render the segments that are inside the axes bounding box with high resolution.

The following args are supported:

- `xy` center of ellipse
- `width` length of horizontal axis
- `height` length of vertical axis
- `angle` rotation in degrees (anti-clockwise)
- `theta1` starting angle of the arc in degrees
- `theta2` ending angle of the arc in degrees

If `theta1` and `theta2` are not provided, the arc will form a complete ellipse.

Valid kwargs are:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>agg_filter</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>alpha</code></td>
<td>float or None</td>
</tr>
<tr>
<td><code>animated</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>antialiased</code> or <code>aa</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>axes</code></td>
<td>an <code>Axes</code> instance</td>
</tr>
<tr>
<td><code>capstyle</code></td>
<td>[‘butt’</td>
</tr>
<tr>
<td><code>clip_box</code></td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td><code>clip_on</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>clip_path</code></td>
<td>[(Path, Transform)</td>
</tr>
</tbody>
</table>
Table 41.1 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgcolor or ec</td>
<td>mpl color spec, None, ‘none’, or ‘auto’</td>
</tr>
<tr>
<td>facecolor or fc</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>['/</td>
</tr>
<tr>
<td>joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['solid'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float or None for default</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

**draw**(artist, renderer, *args, **kwargs)

Ellipses are normally drawn using an approximation that uses eight cubic bezier splines. The error of this approximation is $1.89818e-6$, according to this unverified source:

Lancaster, Don. Approximating a Circle or an Ellipse Using Four Bezier Cubic Splines.


There is a use case where very large ellipses must be drawn with very high accuracy, and it is too expensive to render the entire ellipse with enough segments (either splines or line segments). Therefore, in the case where either radius of the ellipse is large enough that the error of the spline approximation will be visible (greater than one pixel offset from the ideal), a different technique is used.

In that case, only the visible parts of the ellipse are drawn, with each visible arc using a fixed number of spline segments (8). The algorithm proceeds as follows:

1. The points where the ellipse intersects the axes bounding box are located. (This is done by performing an inverse transformation on the axes bbox such that it is relative to the unit circle – this makes the intersection calculation much easier than doing rotated ellipse intersection directly).

This uses the “line intersecting a circle” algorithm from:
2. The angles of each of the intersection points are calculated.
3. Proceeding counterclockwise starting in the positive x-direction, each of the visible arc-segments between the pairs of vertices are drawn using the bezier arc approximation technique implemented in `matplotlib.path.Path.arc()`.

```python
class matplotlib.patches.Arrow(x, y, dx, dy, width=1.0, **kwargs)

Bases: matplotlib.patches.Patch

An arrow patch.

Draws an arrow, starting at (x, y), direction and length given by (dx, dy) the width of the arrow is scaled by width.

Valid kwargs are:
```
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or ec</td>
<td>mpl color spec, None, ‘none’, or ‘auto’</td>
</tr>
<tr>
<td>facecolor or fc</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[''</td>
</tr>
<tr>
<td>joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['solid'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float or None for default</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
</tbody>
</table>
```

Continued on next page
Table 41.2 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

```python
def get_patch_transform()
def get_path()
```

class matplotlib.patches.ArrowStyle

Bases: matplotlib.patches._Style

*ArrowStyle* is a container class which defines several arrowstyle classes, which is used to create an arrow path along a given path. These are mainly used with *FancyArrowPatch*.

A arrowstyle object can be either created as:

```python
ArrowStyle.Fancy(head_length=.4, head_width=.4, tail_width=.4)
```
or:

```python
ArrowStyle("Fancy", head_length=.4, head_width=.4, tail_width=.4)
```
or:

```python
ArrowStyle("Fancy, head_length=.4, head_width=.4, tail_width=.4")
```

The following classes are defined

<table>
<thead>
<tr>
<th>Class</th>
<th>Name</th>
<th>Attrs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Curve</td>
<td>-</td>
<td>None</td>
</tr>
<tr>
<td>CurveB</td>
<td>-&gt;</td>
<td>head_length=0.4,head_width=0.2</td>
</tr>
<tr>
<td>BracketB</td>
<td>-[</td>
<td>widthB=1.0,lengthB=0.2,angleB=None</td>
</tr>
<tr>
<td>Curve-FilledB</td>
<td>-</td>
<td>&gt;</td>
</tr>
<tr>
<td>CurveA</td>
<td>&lt;-</td>
<td>head_length=0.4,head_width=0.2</td>
</tr>
<tr>
<td>CurveAB</td>
<td>&lt;-&gt;</td>
<td>head_length=0.4,head_width=0.2</td>
</tr>
<tr>
<td>Curve-FilledA</td>
<td>&lt;</td>
<td>-</td>
</tr>
<tr>
<td>Curve-FilledAB</td>
<td>&lt;</td>
<td>&gt;-</td>
</tr>
<tr>
<td>BracketA</td>
<td>]-</td>
<td>widthA=1.0,lengthA=0.2,angleA=None</td>
</tr>
<tr>
<td>BracketAB</td>
<td>]-[</td>
<td>widthA=1.0,lengthA=0.2,angleA=None,widthB=1.0,lengthB=0.2,angleB=None</td>
</tr>
<tr>
<td>Fancy</td>
<td>fancy</td>
<td>head_length=0.4,head_width=0.4,tail_width=0.4</td>
</tr>
<tr>
<td>Simple</td>
<td>simple</td>
<td>head_length=0.5,head_width=0.5,tail_width=0.2</td>
</tr>
<tr>
<td>Wedge</td>
<td>wedge</td>
<td>tail_width=0.3,shrink_factor=0.5</td>
</tr>
<tr>
<td>BarAB</td>
<td>[</td>
<td>-</td>
</tr>
</tbody>
</table>
An instance of any arrow style class is a callable object, whose call signature is:

```
__call__(self, path, mutation_size, linewidth, aspect_ratio=1.)
```

and it returns a tuple of a `Path` instance and a boolean value. `path` is a `Path` instance along which the arrow will be drawn. `mutation_size` and `aspect_ratio` have the same meaning as in `BoxStyle`. `linewidth` is a line width to be stroked. This is meant to be used to correct the location of the head so that it does not overshoot the destination point, but not all classes support it.

```
-  <|-->
-> [-]  ]-
-[]  [-][
|--<  fancy
|--<  simple
<--  wedge
|--<  |-|
```

return the instance of the subclass with the given style name.

**class BarAB**

```
class BarAB(widthA=1.0, angleA=None, widthB=1.0, angleB=None)
```

Bases: `matplotlib.patches._Bracket`

An arrow with a bar(\) at both ends.

- **Parameters**
  - `widthA` : float, optional, default
    Width of the bracket
  - `angleA` : float, optional, default
    Angle between the bracket and the line
  - `widthB` : float, optional, default
width of the bracket

angleB : float, optional, default

Angle between the bracket and the line

class BracketA(widthA=1.0, lengthA=0.2, angleA=None)
Bases: matplotlib.patches._Bracket

An arrow with a bracket(]) at its end.

Parameters widthA : float, optional, default

Width of the bracket

lengthA : float, optional, default

Length of the bracket

angleA : float, optional, default

Angle between the bracket and the line

class BracketAB(widthA=1.0, lengthA=0.2, angleA=None, widthB=1.0, lengthB=0.2, angleB=None)
Bases: matplotlib.patches._Bracket

An arrow with a bracket(]) at both ends.

Parameters widthA : float, optional, default

Width of the bracket

lengthA : float, optional, default

Length of the bracket

angleA : float, optional, default

Angle between the bracket and the line

widthB : float, optional, default

Width of the bracket

lengthB : float, optional, default

Length of the bracket

angleB : float, optional, default

Angle between the bracket and the line

class BracketB(widthB=1.0, lengthB=0.2, angleB=None)
Bases: matplotlib.patches._Bracket

An arrow with a bracket([) at its end.

Parameters widthB : float, optional, default

Width of the bracket
**lengthB**: float, optional, default

Length of the bracket

**angleB**: float, optional, default

Angle between the bracket and the line

class Curve
Bases: matplotlib.patches._Curve

A simple curve without any arrow head.

class CurveA(*head_length=0.4, head_width=0.2*)
Bases: matplotlib.patches._Curve

An arrow with a head at its begin point.

**Parameters**

**head_length**: float, optional, default

Length of the arrow head

**head_width**: float, optional, default

Width of the arrow head

class CurveAB(*head_length=0.4, head_width=0.2*)
Bases: matplotlib.patches._Curve

An arrow with heads both at the begin and the end point.

**Parameters**

**head_length**: float, optional, default

Length of the arrow head

**head_width**: float, optional, default

Width of the arrow head

class CurveB(*head_length=0.4, head_width=0.2*)
Bases: matplotlib.patches._Curve

An arrow with a head at its end point.

**Parameters**

**head_length**: float, optional, default

Length of the arrow head

**head_width**: float, optional, default

Width of the arrow head

class CurveFilledA(*head_length=0.4, head_width=0.2*)
Bases: matplotlib.patches._Curve

An arrow with filled triangle head at the begin.

**Parameters**

**head_length**: float, optional, default

Length of the arrow head

**head_width**: float, optional, default

Width of the arrow head
Width of the arrow head

class CurveFilledAB($head\_length=0.4, head\_width=0.2$)
Bases: matplotlib.patches._Curve

An arrow with filled triangle heads at both ends.

**Parameters**

- **head_length**: float, optional, default
  Length of the arrow head

- **head_width**: float, optional, default
  Width of the arrow head

class CurveFilledB($head\_length=0.4, head\_width=0.2$)
Bases: matplotlib.patches._Curve

An arrow with filled triangle head at the end.

**Parameters**

- **head_length**: float, optional, default
  Length of the arrow head

- **head_width**: float, optional, default
  Width of the arrow head

class Fancy($head\_length=0.4, head\_width=0.4, tail\_width=0.4$)
Bases: matplotlib.patches._Base

A fancy arrow. Only works with a quadratic bezier curve.

**Parameters**

- **head_length**: float, optional, default
  Length of the arrow head

- **head_width**: float, optional, default
  Width of the arrow head

- **tail_width**: float, optional, default
  Width of the arrow tail

**transmute**(path, mutation_size, linewidth)

class Simple($head\_length=0.5, head\_width=0.5, tail\_width=0.2$)
Bases: matplotlib.patches._Base

A simple arrow. Only works with a quadratic bezier curve.

**Parameters**

- **head_length**: float, optional, default
  Length of the arrow head

- **head_width**: float, optional, default
  Width of the arrow head

- **tail_width**: float, optional, default
Width of the arrow tail

transmute\((path, mutation\_size, linewidth)\)

class Wedge\((tail\_width=0.3, shrink\_factor=0.5)\)
Bases: matplotlib.patches._Base

Wedge\(\) shape. Only works with a quadratic bezier curve. The begin point has a width of the tail_width and the end point has a width of 0. At the middle, the width is shrink_factor*tail_width.

Parameters tail_width : float, optional, default
Width of the tail

shrink_factor : float, optional, default
Fraction of the arrow width at the middle point

transmute\((path, mutation\_size, linewidth)\)

class matplotlib.patches.BoxStyle
Bases: matplotlib.patches._Style

BoxStyle is a container class which defines several boxstyle classes, which are used for FancyBoxPatch.

A style object can be created as:

BoxStyle.Round(pad=0.2)

or:

BoxStyle("Round", pad=0.2)

or:

BoxStyle("Round, pad=0.2")

Following boxstyle classes are defined.

<table>
<thead>
<tr>
<th>Class</th>
<th>Name</th>
<th>Attrs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Circle</td>
<td>circle</td>
<td>pad=0.3</td>
</tr>
<tr>
<td>DArrow</td>
<td>darrow</td>
<td>pad=0.3</td>
</tr>
<tr>
<td>LArrow</td>
<td>larrow</td>
<td>pad=0.3</td>
</tr>
<tr>
<td>RArrow</td>
<td>rarrow</td>
<td>pad=0.3</td>
</tr>
<tr>
<td>Round</td>
<td>round</td>
<td>pad=0.3,rounding_size=None</td>
</tr>
<tr>
<td>Round4</td>
<td>round4</td>
<td>pad=0.3,rounding_size=None</td>
</tr>
<tr>
<td>Roundtooth</td>
<td>roundtooth</td>
<td>pad=0.3,tooth_size=None</td>
</tr>
<tr>
<td>Sawtooth</td>
<td>sawtooth</td>
<td>pad=0.3,tooth_size=None</td>
</tr>
<tr>
<td>Square</td>
<td>square</td>
<td>pad=0.3</td>
</tr>
</tbody>
</table>

An instance of any boxstyle class is an callable object, whose call signature is:
and returns a Path instance. \( x_0, y_0, \text{width} \) and \( \text{height} \) specify the location and size of the box to be drawn. \( \text{mutation_scale} \) determines the overall size of the mutation (by which I mean the transformation of the rectangle to the fancy box). \( \text{mutation_aspect} \) determines the aspect-ratio of the mutation.

\[
\_\_\text{call}\_\_(self, x0, y0, width, height, \text{mutation_size}, \text{aspect_ratio}=1.)
\]

return the instance of the subclass with the given style name.
class Circle
    Bases: matplotlib.patches._Base
    A simple circle box.

    Parameters
    pad : float
        The amount of padding around the original box.

transmute(x0, y0, width, height, mutation_size)

class DArrow
    Bases: matplotlib.patches._Base
    (Double) Arrow Box

transmute(x0, y0, width, height, mutation_size)

class LArrow
    Bases: matplotlib.patches._Base
    (left) Arrow Box

transmute(x0, y0, width, height, mutation_size)

class RArrow
    Bases: matplotlib.patches.LArrow
    (right) Arrow Box

transmute(x0, y0, width, height, mutation_size)

class Round
    Bases: matplotlib.patches._Base
    A box with round corners.

    Parameters
    pad : float
        amount of padding

    rounding_size : float
        rounding radius of corners. pad if None

    transmute(x0, y0, width, height, mutation_size)

class Round4
    Bases: matplotlib.patches._Base
    Another box with round edges.

    Parameters
    pad : float
        amount of padding

    rounding_size : float
        rounding size of edges. pad if None

    transmute(x0, y0, width, height, mutation_size)
class Roundtooth(pad=0.3, tooth_size=None)
    Bases: matplotlib.patches.Sawtooth

    A rounded tooth box.
    pad amount of padding
    tooth_size size of the sawtooth. pad* if None
    transmute(x0, y0, width, height, mutation_size)

class Sawtooth(pad=0.3, tooth_size=None)
    Bases: matplotlib.patches._Base

    A sawtooth box.
    pad amount of padding
    tooth_size size of the sawtooth. pad* if None
    transmute(x0, y0, width, height, mutation_size)

class Square(pad=0.3)
    Bases: matplotlib.patches._Base

    A simple square box.
    pad amount of padding
    transmute(x0, y0, width, height, mutation_size)

class matplotlib.patches.Circle(xy, radius=5, **kwargs)
    Bases: matplotlib.patches.Ellipse

    A circle patch.

    Create true circle at center $xy = (x, y)$ with given $radius$. Unlike CirclePolygon which is a polygonal approximation, this uses Bézier splines and is much closer to a scale-free circle.

    Valid kwargs are:

    | Property       | Description                                      |
    |----------------|--------------------------------------------------|
    | agg_filter     | unknown                                          |
    | alpha          | float or None                                    |
    | animated       | [True | False]                                     |
    | antialiased or aa | [True | False] or None for default                  |
    | axes           | an Axes instance                                 |
    | capstyle       | ['butt’ | ‘round’ | ‘projecting’]                                 |
    | clip_box       | a matplotlib.transforms.Bbox instance            |
    | clip_on        | [True | False]                                     |
    | clip_path      | [(Path, Transform) | Patch | None ]                                    |
    | color          | matplotlib color spec                           |

Continued on next page
null
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or ec</td>
<td>mpl color spec, None, 'none', or 'auto'</td>
</tr>
<tr>
<td>facecolor or fc</td>
<td>mpl color spec, or None for default, or 'none' for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>['/'</td>
</tr>
<tr>
<td>jointstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['solid'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float or None for default</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

class matplotlib.patches.ConnectionPatch(xyA, xyB, coordsA, coordsB=None, axesA=None, axesB=None, arrowstyle='-', arrow_transmuter=None, connectionstyle='arc3', connector=None, patchA=None, patchB=None, shrinkA=0.0, shrinkB=0.0, mutation_scale=10.0, mutation_aspect=None, clip_on=False, dpi_cor=1.0, **kwargs)

Bases: matplotlib.patches.FancyArrowPatch

A ConnectionPatch class is to make connecting lines between two points (possibly in different axes).

Connect point xyA in coordsA with point xyB in coordsB

Valid keys are
### Key
<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>arrowstyle</td>
<td>the arrow style</td>
</tr>
<tr>
<td>connectionstyle</td>
<td>the connection style</td>
</tr>
<tr>
<td>relpos</td>
<td>default is (0.5, 0.5)</td>
</tr>
<tr>
<td>patchA</td>
<td>default is bounding box of the text</td>
</tr>
<tr>
<td>patchB</td>
<td>default is None</td>
</tr>
<tr>
<td>shrinkA</td>
<td>default is 2 points</td>
</tr>
<tr>
<td>shrinkB</td>
<td>default is 2 points</td>
</tr>
<tr>
<td>mutation_scale</td>
<td>default is text size (in points)</td>
</tr>
<tr>
<td>mutation_aspect</td>
<td>default is 1.</td>
</tr>
<tr>
<td>?</td>
<td>any key for <code>matplotlib.patches.PathPatch</code></td>
</tr>
</tbody>
</table>

`coordsA` and `coordsB` are strings that indicate the coordinates of `xyA` and `xyB`.

### Property
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘figure points’</td>
<td>points from the lower left corner of the figure</td>
</tr>
<tr>
<td>‘figure pixels’</td>
<td>pixels from the lower left corner of the figure</td>
</tr>
<tr>
<td>‘figure fraction’</td>
<td>0,0 is lower left of figure and 1,1 is upper, right</td>
</tr>
<tr>
<td>‘axes points’</td>
<td>points from lower left corner of axes</td>
</tr>
<tr>
<td>‘axes pixels’</td>
<td>pixels from lower left corner of axes</td>
</tr>
<tr>
<td>‘axes fraction’</td>
<td>0,1 is lower left of axes and 1,1 is upper right</td>
</tr>
<tr>
<td>‘data’</td>
<td>use the coordinate system of the object being annotated (default)</td>
</tr>
<tr>
<td>‘offset points’</td>
<td>Specify an offset (in points) from the <code>xy</code> value</td>
</tr>
<tr>
<td>‘polar’</td>
<td>you can specify <code>theta</code>, <code>r</code> for the annotation, even in cartesian plots. Note that if you are using a polar axes, you do not need to specify polar for the coordinate system since that is the native “data” coordinate system.</td>
</tr>
</tbody>
</table>

`draw(renderer)`
Draw.

`get_annotation_clip()`
Return `annotation_clip` attribute. See `set_annotation_clip()` for the meaning of return values.

`get_path_in_displaycoord()`
Return the mutated path of the arrow in the display coord

`set_annotation_clip(b)`
set `annotation_clip` attribute.

- True: the annotation will only be drawn when `self.xy` is inside the axes.
• False: the annotation will always be drawn regardless of its position.
• None: the self.xy will be checked only if xycoords is “data”

class matplotlib.patches.ConnectionStyle
Bases: matplotlib.patches._Style

ConnectionStyle is a container class which defines several connectionstyle classes, which is used to create a path between two points. These are mainly used with FancyArrowPatch.

A connectionstyle object can be either created as:

ConnectionStyle.Arc3(rad=0.2)

or:

ConnectionStyle("Arc3", rad=0.2)

or:

ConnectionStyle("Arc3, rad=0.2")

The following classes are defined:

<table>
<thead>
<tr>
<th>Class</th>
<th>Name</th>
<th>Attrs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Angle</td>
<td>angle</td>
<td>angleA=90,angleB=0,rad=0.0</td>
</tr>
<tr>
<td>Angle3</td>
<td>angle3</td>
<td>angleA=90,angleB=0</td>
</tr>
<tr>
<td>Arc</td>
<td>arc</td>
<td>angleA=0,angleB=0,armA=None,armB=None,rad=0.0</td>
</tr>
<tr>
<td>Arc3</td>
<td>arc3</td>
<td>rad=0.0</td>
</tr>
<tr>
<td>Bar</td>
<td>bar</td>
<td>armA=0.0,armB=0.0,fraction=0.3,angle=None</td>
</tr>
</tbody>
</table>

An instance of any connection style class is a callable object, whose call signature is:

```python
__call__(self, posA, posB,
    patchA=None, patchB=None,
    shrinkA=2., shrinkB=2.)
```

and it returns a Path instance. posA and posB are tuples of x,y coordinates of the two points to be connected. patchA (or patchB) is given, the returned path is clipped so that it start (or end) from the boundary of the patch. The path is further shrunk by shrinkA (or shrinkB) which is given in points.

return the instance of the subclass with the given style name.

class Angle(angleA=90, angleB=0, rad=0.0)
Bases: matplotlib.patches._Base

Creates a piecewise continuous quadratic bezier path between two points. The path has a one passing-through point placed at the intersecting point of two lines which crosses the start (or end) point and has a angle of angleA (or angleB). The connecting edges are rounded with rad.

angleA starting angle of the path
angleB ending angle of the path
rad rounding radius of the edge
connect(posA, posB)

class Angle3(angleA=90, angleB=0)
    Bases: matplotlib.patches._Base
    Creates a simple quadratic bezier curve between two points. The middle control points is placed at the intersecting point of two lines which crosses the start (or end) point and has a angle of angleA (or angleB).
    
    angleA: starting angle of the path
    angleB: ending angle of the path
    connect(posA, posB)

class Arc(angleA=0, angleB=0, armA=None, armB=None, rad=0.0)
    Bases: matplotlib.patches._Base
    Creates a picewise continuous quadratic bezier path between two points. The path can have two passing-through points, a point placed at the distance of armA and angle of angleA from point A, another point with respect to point B. The edges are rounded with rad.
    
    angleA : starting angle of the path
    angleB : ending angle of the path
    armA : length of the starting arm
    armB : length of the ending arm
    rad : rounding radius of the edges
    connect(posA, posB)

class Arc3(rad=0.0)
    Bases: matplotlib.patches._Base
    Creates a simple quadratic bezier curve between two points. The curve is created so that the middle control points (C1) is located at the same distance from the start (C0) and end points(C2) and the distance of the C1 to the line connecting C0-C2 is rad times the distance of C0-C2.
    
    rad : curvature of the curve.
    connect(posA, posB)

class Bar(armA=0.0, armB=0.0, fraction=0.3, angle=None)
    Bases: matplotlib.patches._Base
    A line with angle between A and B with armA and armB. One of the arms is extended so that they are connected in a right angle. The length of armA is determined by (armA + fraction x AB distance). Same for armB.
    
    Parameters armA : float
minimum length of armA

**armB** : float

minimum length of armB

**fraction** : float

a fraction of the distance between two points that will be added to armA and armB.

**angle** : float or None

angle of the connecting line (if None, parallel to A and B)

```
connect(posA, posB)
```

class matplotlib.patches.Ellipse(xy, width, height, angle=0.0, **kwargs)

Bases: matplotlib.patches.Patch

A scale-free ellipse.

**xy** center of ellipse

**width** total length (diameter) of horizontal axis

**height** total length (diameter) of vertical axis

**angle** rotation in degrees (anti-clockwise)

Valid kwargs are:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or ec</td>
<td>mpl color spec, None, 'none', or 'auto'</td>
</tr>
<tr>
<td>facecolor or fc</td>
<td>mpl color spec, or None for default, or 'none' for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>['/'</td>
</tr>
<tr>
<td>joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>linestyle or ls</td>
<td>['solid', 'dashed', 'dashdot', 'dotted', (offset, on-off-dash-seq), '-' or '-' or '-' or '-' or ':' or 'None']</td>
</tr>
<tr>
<td>Linewidth or lw</td>
<td>float or None for default</td>
</tr>
<tr>
<td>Path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>Picker</td>
<td>[None, float, boolean, callable]</td>
</tr>
<tr>
<td>Rasterized</td>
<td>[True, False, None]</td>
</tr>
<tr>
<td>Sketch Params</td>
<td>unknown</td>
</tr>
<tr>
<td>Snap</td>
<td>unknown</td>
</tr>
<tr>
<td>Transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>Url</td>
<td>a url string</td>
</tr>
<tr>
<td>Visible</td>
<td>[True, False]</td>
</tr>
<tr>
<td>Zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

```python
def get_patch_transform()

def get_path()
    Return the vertices of the rectangle

class FancyArrow(matplotlib.patches.Polygon)
    Like Arrow, but lets you set head width and head height independently.

    Constructor arguments:

    **width**: float (default: 0.001) width of full arrow tail

    **length_includes_head**: [True | False] (default: False) True if head is to be counted in calculating the length.

    **head_width**: float or None (default: 3*width) total width of the full arrow head

    **head_length**: float or None (default: 1.5 * head_width) length of arrow head

    **shape**: ['full', 'left', 'right'] (default: 'full') draw the left-half, right-half, or full arrow

    **overhang**: float (default: 0) fraction that the arrow is swept back (0 overhang means triangular shape). Can be negative or greater than one.

    **head_starts_at_zero**: [True | False] (default: False) if True, the head starts being drawn at coordinate 0 instead of ending at coordinate 0.

Other valid kwargs (inherited from Patch) are:
Table 41.6 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>axes</td>
<td>[True</td>
</tr>
<tr>
<td>capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor</td>
<td>mpl color spec, None, ‘none’, or ‘auto’</td>
</tr>
<tr>
<td>facecolor</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>['f'</td>
</tr>
<tr>
<td>joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle</td>
<td>['solid'</td>
</tr>
<tr>
<td>linewidth</td>
<td>float or None for default</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

class matplotlib.patches.FancyArrowPatch(posA=None, posB=None, path=None, arrowstyle='simple', arrow_transmuter=None, connectionstyle='arc3', connector=None, patchA=None, patchB=None, shrinkA=2.0, shrinkB=2.0, mutation_scale=1.0, mutation_aspect=None, dpi_cor=1.0, **kwargs)

Bases: matplotlib.patches.Patch

A fancy arrow patch. It draws an arrow using the :class:ArrowStyle.

If posA and posB is given, a path connecting two point are created according to the connectionstyle. The path will be clipped with patchA and patchB and further shrunk by shrinkA and shrinkB. An arrow is drawn along this resulting path using the arrowstyle parameter. If path provided, an arrow is drawn along this path and patchA, patchB, shrinkA, and shrinkB are ignored.
The \textit{connectionstyle} describes how \textit{posA} and \textit{posB} are connected. It can be an instance of the ConnectionStyle class (matplotlib.patches.ConnectionStyle) or a string of the connectionstyle name, with optional comma-separated attributes. The following connection styles are available.

<table>
<thead>
<tr>
<th>Class</th>
<th>Name</th>
<th>Attrs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Angle</td>
<td>angle</td>
<td>angleA=90,angleB=0,rad=0.0</td>
</tr>
<tr>
<td>Angle3</td>
<td>angle3</td>
<td>angleA=90,angleB=0</td>
</tr>
<tr>
<td>Arc</td>
<td>arc</td>
<td>angleA=0,angleB=0,armA=None,armB=None,rad=0.0</td>
</tr>
<tr>
<td>Arc3</td>
<td>arc3</td>
<td>rad=0.0</td>
</tr>
<tr>
<td>Bar</td>
<td>bar</td>
<td>armA=0.0,armB=0.0,fraction=0.3,angle=None</td>
</tr>
</tbody>
</table>

The \textit{arrowstyle} describes how the fancy arrow will be drawn. It can be string of the available arrowstyle names, with optional comma-separated attributes, or one of the ArrowStyle instance. The optional attributes are meant to be scaled with the \textit{mutation_scale}. The following arrow styles are available.

<table>
<thead>
<tr>
<th>Class</th>
<th>Name</th>
<th>Attrs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Curve</td>
<td>-</td>
<td>None</td>
</tr>
<tr>
<td>CurveB</td>
<td>-&gt;</td>
<td>head_length=0.4,head_width=0.2</td>
</tr>
<tr>
<td>BracketB</td>
<td>-[</td>
<td>widthB=1.0,lengthB=0.2,angleB=None</td>
</tr>
<tr>
<td>Curve-FilledB</td>
<td>-</td>
<td>&gt;</td>
</tr>
<tr>
<td>CurveA</td>
<td>&lt;-</td>
<td>head_length=0.4,head_width=0.2</td>
</tr>
<tr>
<td>CurveAB</td>
<td>&lt;-&gt;</td>
<td>head_length=0.4,head_width=0.2</td>
</tr>
<tr>
<td>Curve-FilledA</td>
<td>&lt;</td>
<td></td>
</tr>
<tr>
<td>Curve-FilledAB</td>
<td>&lt;</td>
<td>-</td>
</tr>
<tr>
<td>BracketA</td>
<td>]-</td>
<td>widthA=1.0,lengthA=0.2,angleA=None,angleB=None</td>
</tr>
<tr>
<td>BracketAB</td>
<td>]-[</td>
<td>widthA=1.0,lengthA=0.2,angleA=None,widthB=1.0,lengthB=0.2,angleB=None</td>
</tr>
<tr>
<td>Fancy</td>
<td>fancy</td>
<td>head_length=0.4,head_width=0.4,tail_width=0.4</td>
</tr>
<tr>
<td>Simple</td>
<td>simple</td>
<td>head_length=0.5,head_width=0.5,tail_width=0.2</td>
</tr>
<tr>
<td>Wedge</td>
<td>wedge</td>
<td>tail_width=0.3,shrink_factor=0.5</td>
</tr>
<tr>
<td>BarAB</td>
<td></td>
<td>-</td>
</tr>
</tbody>
</table>

\textit{mutation_scale} [a value with which attributes of arrowstyle] (e.g., head\_length) will be scaled. default=1.

\textit{mutation_aspect} [The height of the rectangle will be] squeezed by this value before the mutation and the mutated box will be stretched by the inverse of it. default=None.

Valid kwargs are:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
</tbody>
</table>

Continued on next page
Table 41.7 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>axes</td>
<td>an <code>Axes</code> instance</td>
</tr>
<tr>
<td>capstyle</td>
<td>['but’</td>
</tr>
<tr>
<td>clip_box</td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>((Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor</td>
<td>mpl color spec, None, ‘none’, or ‘auto’</td>
</tr>
<tr>
<td>facecolor</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[‘</td>
</tr>
<tr>
<td>joinstyle</td>
<td>['miter’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle</td>
<td>linestyle or ls [‘solid’</td>
</tr>
<tr>
<td>linewidth</td>
<td>float or None for default</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td><code>Transform</code> instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

draw(renderer)

get_arrowstyle()  
    Return the arrowstyle object

get_connectionstyle()  
    Return the ConnectionStyle instance

get_dpi_cor()  
    dpi_cor is currently used for linewidth-related things and shrink factor. Mutation scale is affected by this.

get_mutation_aspect()  
    Return the aspect ratio of the bbox mutation.

get_mutation_scale()  
    Return the mutation scale.
**get_path()**

return the path of the arrow in the data coordinate. Use get_path_in_displaycoord() method to retrieve the arrow path in the display coord.

**get_path_in_displaycoord()**

Return the mutated path of the arrow in the display coord

**set_arrowstyle(arrowstyle=None, **kw)**

Set the arrow style.

*arrowstyle can be a string with arrowstyle name with optional* comma-separated attributes. Alternatively, the attrs can be provided as keywords.

set_arrowstyle(“Fancy,head_length=0.2”) set_arrowstyle(“fancy”, head_length=0.2)

Old attrs simply are forgotten.

Without argument (or with arrowstyle=None), return available box styles as a list of strings.

**set_connectionstyle(connectionstyle, **kw)**

Set the connection style.

*connectionstyle can be a string with connectionstyle name with optional* comma-separated attributes. Alternatively, the attrs can be provided as keywords.

set_connectionstyle(“arc,angleA=0,armA=30,rad=10”) set_connectionstyle(“arc”, angleA=0,armA=30,rad=10)

Old attrs simply are forgotten.

Without argument (or with connectionstyle=None), return available styles as a list of strings.

**set_dpi_cor(dpi_cor)**

dpi_cor is currently used for linewidth-related things and shrink factor. Mutation scale is affected by this.

**set_mutation_aspect(aspect)**

Set the aspect ratio of the bbox mutation.

ACCEPTS: float

**set_mutation_scale(scale)**

Set the mutation scale.

ACCEPTS: float

**set_patchA(patchA)**

set the begin patch.

**set_patchB(patchB)**

set the begin patch

**set_positions(posA, posB)**

set the begin and end positions of the connecting path. Use current value if None.
class matplotlib.patches.FancyBboxPatch(xy, width, height, boxstyle='round',
bbox_transmuter=None, mutation_scale=1.0,
mutation_aspect=None, **kwargs)

Bases: matplotlib.patches.Patch

Draw a fancy box around a rectangle with lower left at \(xy^*=(x, y)\) with specified width and height.

FancyBboxPatch class is similar to Rectangle class, but it draws a fancy box around the rectangle. The transformation of the rectangle box to the fancy box is delegated to the BoxTransmuterBase and its derived classes.

\(xy\) = lower left corner

\(width, height\)

boxstyle determines what kind of fancy box will be drawn. It can be a string of the style name with a comma separated attribute, or an instance of BoxStyle. Following box styles are available.

<table>
<thead>
<tr>
<th>Class</th>
<th>Name</th>
<th>Attrs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Circle</td>
<td>circle</td>
<td>pad=0.3</td>
</tr>
<tr>
<td>DArrow</td>
<td>darrow</td>
<td>pad=0.3</td>
</tr>
<tr>
<td>LArrow</td>
<td>larrow</td>
<td>pad=0.3</td>
</tr>
<tr>
<td>RArrow</td>
<td>rarrow</td>
<td>pad=0.3</td>
</tr>
<tr>
<td>Round</td>
<td>round</td>
<td>pad=0.3, rounding_size=None</td>
</tr>
<tr>
<td>Round4</td>
<td>round4</td>
<td>pad=0.3, rounding_size=None</td>
</tr>
<tr>
<td>Roundtooth</td>
<td>roundtooth</td>
<td>pad=0.3, tooth_size=None</td>
</tr>
<tr>
<td>Sawtooth</td>
<td>sawtooth</td>
<td>pad=0.3, tooth_size=None</td>
</tr>
<tr>
<td>Square</td>
<td>square</td>
<td>pad=0.3</td>
</tr>
</tbody>
</table>

mutation_scale : a value with which attributes of boxstyle (e.g., pad) will be scaled. default=1.

mutation_aspect : The height of the rectangle will be squeezed by this value before the mutation and the mutated box will be stretched by the inverse of it. default=None.

Valid kwargs are:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or ec</td>
<td>mpl color spec, None, 'none', or 'auto'</td>
</tr>
<tr>
<td>facecolor or fc</td>
<td>mpl color spec, or None for default, or 'none' for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
</tbody>
</table>

Continued on next page
## Table 41.8 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[''</td>
</tr>
<tr>
<td>joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with %s conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['solid'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float or None for default</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

### get\_bbox()

Returns the bounding box of the rectangle.

### get\_boxstyle()

Returns the boxstyle object.

### get\_height()

Returns the height of the rectangle.

### get\_mutation\_aspect()

Returns the aspect ratio of the bbox mutation.

### get\_mutation\_scale()

Returns the mutation scale.

### get\_path()

Returns the mutated path of the rectangle.

### get\_width()

Returns the width of the rectangle.

### get\_x()

Returns the left coord of the rectangle.

### get\_y()

Returns the bottom coord of the rectangle.

### set\_bounds(*args)

Set the bounds of the rectangle: l,b,w,h

Accepts: (left, bottom, width, height)
set_boxstyle(boxstyle=None, **kw)

Set the box style.

boxstyle can be a string with boxstyle name with optional comma-separated attributes. Alternatively, the attrs can be provided as keywords:

```python
def set_boxstyle("round,pad=0.2")
def set_boxstyle("round", pad=0.2)
```

Old attrs simply are forgotten.

Without argument (or with `boxstyle=None`), it returns available box styles.

The following boxstyles are available:

<table>
<thead>
<tr>
<th>Class</th>
<th>Name</th>
<th>Attrs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Circle</td>
<td>circle</td>
<td>pad=0.3</td>
</tr>
<tr>
<td>DArrow</td>
<td>darrow</td>
<td>pad=0.3</td>
</tr>
<tr>
<td>LArrow</td>
<td>larrow</td>
<td>pad=0.3</td>
</tr>
<tr>
<td>RArrow</td>
<td>rarrow</td>
<td>pad=0.3</td>
</tr>
<tr>
<td>Round</td>
<td>round</td>
<td>pad=0.3, rounding_size=None</td>
</tr>
<tr>
<td>Round4</td>
<td>round4</td>
<td>pad=0.3, rounding_size=None</td>
</tr>
<tr>
<td>Roundtooth</td>
<td>roundtooth</td>
<td>pad=0.3, tooth_size=None</td>
</tr>
<tr>
<td>Sawtooth</td>
<td>sawtooth</td>
<td>pad=0.3, tooth_size=None</td>
</tr>
<tr>
<td>Square</td>
<td>square</td>
<td>pad=0.3</td>
</tr>
</tbody>
</table>

ACCEPTS: [‘circle’ | ‘darrow’ | ‘larrow’ | ‘rarrow’ | ‘round’ | ‘round4’ | ‘roundtooth’ | ‘sawtooth’ | ‘square’ ]

set_height(h)

Set the width rectangle

ACCEPTS: float

set_mutation_aspect(aspect)

Set the aspect ratio of the bbox mutation.

ACCEPTS: float

set_mutation_scale(scale)

Set the mutation scale.

ACCEPTS: float

set_width(w)

Set the width rectangle

ACCEPTS: float

set_x(x)

Set the left coord of the rectangle

ACCEPTS: float

set_y(y)

Set the bottom coord of the rectangle
ACCEPTS: float

class matplotlib.patches.Patch(edgecolor=\texttt{None}, facecolor=\texttt{None}, color=\texttt{None}, linewidth=\texttt{None}, linestyle=\texttt{None}, antialiased=\texttt{None}, hatch=\texttt{None}, fill=\texttt{True}, capstyle=\texttt{None}, joinstyle=\texttt{None}, **\texttt{kwargs})

Bases: \texttt{matplotlib.artist.Artist}

A patch is a 2D artist with a face color and an edge color.

If any of \texttt{edgecolor}, \texttt{facecolor}, \texttt{linewidth}, or \texttt{antialiased} are \texttt{None}, they default to their rc params setting.

The following kwarg properties are supported

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{agg_filter}</td>
<td>unknown</td>
</tr>
<tr>
<td>\texttt{alpha}</td>
<td>float or \texttt{None}</td>
</tr>
<tr>
<td>\texttt{animated}</td>
<td>\texttt{[True</td>
</tr>
<tr>
<td>\texttt{antialiased or aa}</td>
<td>\texttt{[True</td>
</tr>
<tr>
<td>\texttt{axes}</td>
<td>an \texttt{Axes} instance</td>
</tr>
<tr>
<td>\texttt{capstyle}</td>
<td>\texttt{['butt'</td>
</tr>
<tr>
<td>\texttt{clip_box}</td>
<td>a \texttt{matplotlib.transforms.Bbox} instance</td>
</tr>
<tr>
<td>\texttt{clip_on}</td>
<td>\texttt{[True</td>
</tr>
<tr>
<td>\texttt{clip_path}</td>
<td>\texttt{([Path, Transform]</td>
</tr>
<tr>
<td>\texttt{color}</td>
<td>\texttt{matplotlib color spec}</td>
</tr>
<tr>
<td>\texttt{contains}</td>
<td>a callable function</td>
</tr>
<tr>
<td>\texttt{edgecolor or ec}</td>
<td>\texttt{mpl color spec, \texttt{None}, 'none', or 'auto'}</td>
</tr>
<tr>
<td>\texttt{facecolor or fc}</td>
<td>\texttt{mpl color spec, or \texttt{None} for default, or 'none' for no color}</td>
</tr>
<tr>
<td>\texttt{figure}</td>
<td>a \texttt{matplotlib.figure.Figure} instance</td>
</tr>
<tr>
<td>\texttt{fill}</td>
<td>\texttt{[True</td>
</tr>
<tr>
<td>\texttt{gid}</td>
<td>an id string</td>
</tr>
<tr>
<td>\texttt{hatch}</td>
<td>\texttt{['/'</td>
</tr>
<tr>
<td>\texttt{joinstyle}</td>
<td>\texttt{['miter'</td>
</tr>
<tr>
<td>\texttt{label}</td>
<td>string or anything printable with \texttt{%s} conversion.</td>
</tr>
<tr>
<td>\texttt{linestyle or ls}</td>
<td>\texttt{['solid'</td>
</tr>
<tr>
<td>\texttt{linewidth or lw}</td>
<td>float or \texttt{None} for default</td>
</tr>
<tr>
<td>\texttt{path_effects}</td>
<td>unknown</td>
</tr>
<tr>
<td>\texttt{picker}</td>
<td>\texttt{[None</td>
</tr>
<tr>
<td>\texttt{rasterized}</td>
<td>\texttt{[True</td>
</tr>
<tr>
<td>\texttt{sketch_params}</td>
<td>unknown</td>
</tr>
<tr>
<td>\texttt{snap}</td>
<td>unknown</td>
</tr>
<tr>
<td>\texttt{transform}</td>
<td>\texttt{Transform} instance</td>
</tr>
<tr>
<td>\texttt{url}</td>
<td>a url string</td>
</tr>
<tr>
<td>\texttt{visible}</td>
<td>\texttt{[True</td>
</tr>
<tr>
<td>\texttt{zorder}</td>
<td>any number</td>
</tr>
</tbody>
</table>

\texttt{contains(mouseevent, radius=\texttt{None})}
Test whether the mouse event occurred in the patch.

Returns T/F. {}

c contains_point(point, radius=None)
Returns True if the given point is inside the path (transformed with its transform attribute).

d draw(artist, renderer, *args, **kwargs)
Draw the Patch to the given renderer.

fill
return whether fill is set

g et_aa()
Returns True if the Patch is to be drawn with antialiasing.

g et_anti aliased()
Returns True if the Patch is to be drawn with antialiasing.

g et_cap style()
Return the current capstyle

g et_data_transform()
Return the Transform instance which maps data coordinates to physical coordinates.

g et_ec()
Return the edge color of the Patch.

g et_ edgecolor()
Return the edge color of the Patch.

g et extents()
Return a Bbox object defining the axis-aligned extents of the Patch.

g et_facecolor()
Return the face color of the Patch.

g et_fc()
Return the face color of the Patch.

g et_fill()        return whether fill is set

g et_hatch()      Return the current hatching pattern

g et_join style() Return the current joinstyle

g et_linestyle() Return the linestyle. Will be one of ['solid' | 'dashed' | 'dashdot' | 'dotted']

g et_linewidth() Return the line width in points.

g et_ls()        Return the linestyle. Will be one of ['solid' | 'dashed' | 'dashdot' | 'dotted']
**get_lw()**
Return the line width in points.

**get_patch_transform()**
Return the **Transform** instance which takes patch coordinates to data coordinates.

For example, one may define a patch of a circle which represents a radius of 5 by providing coordinates for a unit circle, and a transform which scales the coordinates (the patch coordinate) by 5.

**get_path()**
Return the path of this patch

**get_transform()**
Return the **Transform** applied to the **Patch**.

**get_verts()**
Return a copy of the vertices used in this patch

If the patch contains Bezier curves, the curves will be interpolated by line segments. To access the curves as curves, use **get_path()**.

**get_window_extent**(**renderer**=*None*)

**set_aa**(aa)
alias for **set_antialiased**

**set_alpha**(alpha)
Set the alpha transparency of the patch.

ACCEPTS: float or None

**set_antialiased**(aa)
Set whether to use antialiased rendering

ACCEPTS: [True | False] or None for default

**set_capstyle**(s)
Set the patch capstyle

ACCEPTS: [’butt’, ’round’, ’projecting’]

**set_color**(c)
Set both the edgecolor and the facecolor.

ACCEPTS: matplotlib color spec

See also:

**set_facecolor**, **set_edgecolor** For setting the edge or face color individually.

**set_ec**(color)
alias for **set_edgecolor**
**set_edgecolor**(color)
Set the patch edge color

ACCEPSES: mpl color spec, None, ‘none’, or ‘auto’

**set_facecolor**(color)
Set the patch face color

ACCEPSES: mpl color spec, or None for default, or ‘none’ for no color

**set_fc**(color)
alias for set_facecolor

**set_fill**(b)
Set whether to fill the patch

ACCEPSES: [True | False]

**set_hatch**(hatch)
Set the hatching pattern

*hatch* can be one of:

```
/ - diagonal hatching
\ - back diagonal
| - vertical
- - horizontal
+ - crossed
x - crossed diagonal
o - small circle
O - large circle
. - dots
* - stars
```

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

ACCEPSES: ['/ | \ | | | - | + | x | o | O | . | *']

**set_joinstyle**(s)
Set the patch joinstyle

ACCEPSES: ['miter' | 'round' | 'bevel']

**set_linestyle**(ls)
Set the patch linestyle

<table>
<thead>
<tr>
<th>linestyle</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'-' or 'solid'</td>
<td>solid line</td>
</tr>
<tr>
<td>'--' or 'dashed'</td>
<td>dashed line</td>
</tr>
<tr>
<td>'-.' or 'dashdot'</td>
<td>dash-dotted line</td>
</tr>
<tr>
<td>':' or 'dotted'</td>
<td>dotted line</td>
</tr>
</tbody>
</table>

Alternatively a dash tuple of the following form can be provided:
(offset, onoffseq),

where onoffseq is an even length tuple of on and off ink in points.

ACCEPST: ['solid', 'dashed', 'dashdot', 'dotted'] | (offset, on-off-dash-seq) | '-' | '--' | '-.' | ':' | 'None' | ' ' | ''

Parameters ls : { ' ', '−', '−−', '−.' } and more see description

The line style.

set_linewidth(w)
Set the patch linewidth in points

ACCEPTS: float or None for default

set_ls(ls)
alias for set_linestyle

set_lw(lw)
alias for set_linewidth

update_from(other)
Updates this Patch from the properties of other.

validCap = ('butt', 'round', 'projecting')

validJoin = ('miter', 'round', 'bevel')

zorder = 1

class matplotlib.patches.PathPatch(path, **kwargs)
Bases: matplotlib.patches.Patch

A general polycurve path patch.

*path* is a matplotlib.path.Path object.

Valid kwargs are:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
</tbody>
</table>

Continued on next page
Properties and descriptions from a table:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clip_path</td>
<td>[ (Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or ec</td>
<td>mpl color spec, None, ‘none’, or ‘auto’</td>
</tr>
<tr>
<td>facecolor or fc</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>['', '/', '', '-', '+', 'x', 'o', 'O', '.', '*']</td>
</tr>
<tr>
<td>joinstyle</td>
<td>['miter', 'round', 'bevel']</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with '%s' conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['solid', 'dashed', 'dashdot', 'dotted']</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float or None for default</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

See also:

Patch For additional kwargs

get_path()

```python
class matplotlib.patches.Polygon(xy, closed=True, **kwargs)
```

A general polygon patch.

xy is a numpy array with shape Nx2.

If closed is True, the polygon will be closed so the starting and ending points are the same.

Valid kwarg:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
</tbody>
</table>
### Table 41.11 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or ec</td>
<td>mpl color spec, None, ‘none’, or ‘auto’</td>
</tr>
<tr>
<td>facecolor or fc</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>['/\</td>
</tr>
<tr>
<td>joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['solid'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float or None for default</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

**See also:**

**Patch** For additional kwargs

**get_closed()**

Returns if the polygon is closed

**Returns closed** : bool

  If the path is closed

**get_path()**

Get the path of the polygon

**Returns path** : Path
The `Path` object for the polygon

```python
def get_xy()
    Get the vertices of the path

    Returns vertices : numpy array

    The coordinates of the vertices as a Nx2 ndarray.
```

```python
def set_closed(closed)
    Set if the polygon is closed

    Parameters closed : bool

    True if the polygon is closed
```

```python
def set_xy(xy)
    Set the vertices of the polygon

    Parameters xy : numpy array or iterable of pairs

    The coordinates of the vertices as a Nx2 ndarray or iterable of pairs.
```

```python
Matplotlib, Release 2.0.2

1358 Chapter 41. patches

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or ec</td>
<td>mpl color spec, None, 'none', or 'auto'</td>
</tr>
<tr>
<td>facecolor or fc</td>
<td>mpl color spec, or None for default, or 'none' for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
</tbody>
</table>
```

Continued on next page
Table 41.12 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>['/'</td>
</tr>
<tr>
<td>joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['solid'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float or None for default</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

get_bbox()

get_height()
   Return the height of the rectangle

get_patch_transform()

get_path()
   Return the vertices of the rectangle

get_width()
   Return the width of the rectangle

get_x()
   Return the left coord of the rectangle

get_xy()
   Return the left and bottom coords of the rectangle

get_y()
   Return the bottom coord of the rectangle

set_bounds(*args)
   Set the bounds of the rectangle: l,b,w,h
   ACCEPTS: (left, bottom, width, height)

set_height(h)
   Set the width rectangle
   ACCEPTS: float
**set_width**(*w*)
Set the width rectangle

**ACCEPTS:** float

**set_x**(*x*)
Set the left coord of the rectangle

**ACCEPTS:** float

**set_xy**(*xy*)
Set the left and bottom coords of the rectangle

**ACCEPTS:** 2-item sequence

**set_y**(*y*)
Set the bottom coord of the rectangle

**ACCEPTS:** float

**xy**
Return the left and bottom coords of the rectangle

---

**class** **matplotlib.patches.RegularPolygon**(*xy*, *numVertices*, *radius=5*, *orientation=0*, **kwargs)

Bases: **matplotlib.patches.Patch**

A regular polygon patch.

Constructor arguments:

* **xy** A length 2 tuple (x, y) of the center.

* **numVertices** the number of vertices.

* **radius** The distance from the center to each of the vertices.

* **orientation** rotates the polygon (in radians).

Valid kwargs are:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an <em>Axes</em> instance</td>
</tr>
<tr>
<td>capstyle</td>
<td>[‘butt’</td>
</tr>
<tr>
<td>clip_box</td>
<td>a <em>matplotlib.transforms.Bbox</em> instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgcolor or ec</td>
<td>mpl color spec, None, ‘none’, or ‘auto’</td>
</tr>
</tbody>
</table>
Table 41.13 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>facecolor</code> or <code>fc</code></td>
<td>mpl color spec, or None for default, or 'none' for no color</td>
</tr>
<tr>
<td><code>figure</code></td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td><code>fill</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>gid</code></td>
<td>an id string</td>
</tr>
<tr>
<td><code>hatch</code></td>
<td>['/', '', '</td>
</tr>
<tr>
<td><code>joinstyle</code></td>
<td>['miter', 'round', 'bevel']</td>
</tr>
<tr>
<td><code>label</code></td>
<td>string or anything printable with '%s' conversion.</td>
</tr>
<tr>
<td><code>linestyle</code> or <code>ls</code></td>
<td>['solid', 'dashed', 'dashdot', 'dotted', (offset, on-off-dash-seq)]</td>
</tr>
<tr>
<td><code>linewidth</code> or <code>lw</code></td>
<td>float or None for default</td>
</tr>
<tr>
<td><code>path_effects</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>picker</code></td>
<td>[None</td>
</tr>
<tr>
<td><code>rasterized</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>sketch_params</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>snap</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>transform</code></td>
<td><code>Transform</code> instance</td>
</tr>
<tr>
<td><code>url</code></td>
<td>a url string</td>
</tr>
<tr>
<td><code>visible</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>zorder</code></td>
<td>any number</td>
</tr>
</tbody>
</table>

```python
get_patch_transform()
```

```python
get_path()
```

```python
numvertices
```

```python
orientation
```

```python
radius
```

```python
xy
```

```python
class matplotlib.patches.Shadow
```

Create a shadow of the given `patch` offset by `ox`, `oy`. `props=\(None\)`, `**kwargs`)

Bases: `matplotlib.patches.Patch`

Create a shadow of the given `patch` offset by `ox`, `oy`. `props`, if not `None`, is a patch property update dictionary. If `None`, the shadow will have the same color as the face, but darkened.

`**kwargs` are
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>agg_filter</strong></td>
<td>unknown</td>
</tr>
<tr>
<td><strong>alpha</strong></td>
<td>float or None</td>
</tr>
<tr>
<td><strong>animated</strong></td>
<td>[True</td>
</tr>
<tr>
<td><strong>antialiased</strong></td>
<td>[True</td>
</tr>
<tr>
<td><strong>axes</strong></td>
<td>an Axes instance</td>
</tr>
<tr>
<td><strong>capstyle</strong></td>
<td>['butt'</td>
</tr>
<tr>
<td><strong>clip_box</strong></td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td><strong>clip_on</strong></td>
<td>[True</td>
</tr>
<tr>
<td><strong>clip_path</strong></td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td><strong>color</strong></td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td><strong>contains</strong></td>
<td>a callable function</td>
</tr>
<tr>
<td><strong>edgecolor</strong></td>
<td>or ec mpl color spec, None, ‘none’, or ‘auto’</td>
</tr>
<tr>
<td><strong>facecolor</strong></td>
<td>or fc mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td><strong>figure</strong></td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td><strong>fill</strong></td>
<td>[True</td>
</tr>
<tr>
<td><strong>gid</strong></td>
<td>an id string</td>
</tr>
<tr>
<td><strong>hatch</strong></td>
<td>['/'</td>
</tr>
<tr>
<td><strong>joinstyle</strong></td>
<td>['miter'</td>
</tr>
<tr>
<td><strong>label</strong></td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td><strong>linestyle</strong></td>
<td>or ls ['solid'</td>
</tr>
<tr>
<td><strong>linewidth</strong></td>
<td>or lw float or None for default</td>
</tr>
<tr>
<td><strong>path_effects</strong></td>
<td>unknown</td>
</tr>
<tr>
<td><strong>picker</strong></td>
<td>[None</td>
</tr>
<tr>
<td><strong>rasterized</strong></td>
<td>[True</td>
</tr>
<tr>
<td><strong>sketch_params</strong></td>
<td>unknown</td>
</tr>
<tr>
<td><strong>snap</strong></td>
<td>unknown</td>
</tr>
<tr>
<td><strong>transform</strong></td>
<td>Transform instance</td>
</tr>
<tr>
<td><strong>url</strong></td>
<td>a url string</td>
</tr>
<tr>
<td><strong>visible</strong></td>
<td>[True</td>
</tr>
<tr>
<td><strong>zorder</strong></td>
<td>any number</td>
</tr>
</tbody>
</table>

```
    draw(renderer)

    get_patch_transform()

    get_path()
```

**class matplotlib.patches.Wedge**(center, r, theta1, theta2, width=None, **kwargs)

    Bases: matplotlib.patches.Patch

    Wedge shaped patch.

    Draw a wedge centered at x, y center with radius r that sweeps theta1 to theta2 (in degrees). If width
is given, then a partial wedge is drawn from inner radius \( r - width \) to outer radius \( r \).

Valid kwargs are:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or ec</td>
<td>mpl color spec, None, ‘none’, or ‘auto’</td>
</tr>
<tr>
<td>facecolor or fc</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>['/'</td>
</tr>
<tr>
<td>joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['solid'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float or None for default</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None[float</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

get_path()

set_center(center)

set_radius(radius)

set_thetal(theta)
set_theta2(theta2)

set_width(width)

class matplotlib.patches.YAArrow(figure, xytip, xybase, width=4, frac=0.1, headwidth=12, **kwargs)

Bases: matplotlib.patches.Patch

Yet another arrow class.

This is an arrow that is defined in display space and has a tip at x1, y1 and a base at x2, y2.

Constructor arguments:

xytip (x, y) location of arrow tip

xybase (x, y) location the arrow base mid point

figure The Figure instance (fig.dpi)

width The width of the arrow in points

frac The fraction of the arrow length occupied by the head

headwidth The width of the base of the arrow head in points

Valid kwargs are:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased</td>
<td>or aa [True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor</td>
<td>or ec mpl color spec, None, ‘none’, or ‘auto’</td>
</tr>
<tr>
<td>facecolor</td>
<td>or fc mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>['-'</td>
</tr>
<tr>
<td>joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle</td>
<td>or ls ['solid'</td>
</tr>
<tr>
<td>linewidth</td>
<td>or lw float or None for default</td>
</tr>
</tbody>
</table>

Continued on next page
Table 41.16 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

get_patch_transform()

get_path()

getpoints(x1, y1, x2, y2, k)

For line segment defined by (x₁, y₁) and (x₂, y₂) return the points on the line that is perpendicular to the line and intersects (x₂, y₂) and the distance from (x₂, y₂) of the returned points is k.

matplotlib.patches.bbox_artist(artist, renderer, props=None, fill=True)

This is a debug function to draw a rectangle around the bounding box returned by get_window_extent() of an artist, to test whether the artist is returning the correct bbox.

props is a dict of rectangle props with the additional property ‘pad’ that sets the padding around the bbox in points.

matplotlib.patches.draw_bbox(bbox, renderer, color='k', trans=None)

This is a debug function to draw a rectangle around the bounding box returned by get_window_extent() of an artist, to test whether the artist is returning the correct bbox.
42.1 matplotlib.path

A module for dealing with the polylines used throughout matplotlib.

The primary class for polyline handling in matplotlib is Path. Almost all vector drawing makes use of Paths somewhere in the drawing pipeline.

Whilst a Path instance itself cannot be drawn, there exists Artist subclasses which can be used for convenient Path visualisation - the two most frequently used of these are PathPatch and PathCollection.

class matplotlib.path.Path(vertices, codes=None, _interpolation_steps=1, closed=False, reads-only=False)

Bases: object

Path represents a series of possibly disconnected, possibly closed, line and curve segments.

The underlying storage is made up of two parallel numpy arrays:

- vertices: an Nx2 float array of vertices
- codes: an N-length uint8 array of vertex types

These two arrays always have the same length in the first dimension. For example, to represent a cubic curve, you must provide three vertices as well as three codes CURVE3.

The code types are:

- **STOP** [1 vertex (ignored)] A marker for the end of the entire path (currently not required and ignored)
- **MOVETO** [1 vertex] Pick up the pen and move to the given vertex.
- **LINETO** [1 vertex] Draw a line from the current position to the given vertex.
- **CURVE3** [1 control point, 1 endpoint] Draw a quadratic Bezier curve from the current position, with the given control point, to the given end point.
- **CURVE4** [2 control points, 1 endpoint] Draw a cubic Bezier curve from the current position, with the given control points, to the given end point.
- **CLOSEPOLY** [1 vertex (ignored)] Draw a line segment to the start point of the current polyline.
Users of Path objects should not access the vertices and codes arrays directly. Instead, they should use `iter_segments()` or `cleaned()` to get the vertex/code pairs. This is important, since many Path objects, as an optimization, do not store a codes at all, but have a default one provided for them by `iter_segments()`.

**Note:** The vertices and codes arrays should be treated as immutable – there are a number of optimizations and assumptions made up front in the constructor that will not change when the data changes.

Create a new path with the given vertices and codes.

**Parameters**

- **vertices**: array_like
  - The (n, 2) float array, masked array or sequence of pairs representing the vertices of the path.
  - If `vertices` contains masked values, they will be converted to NaNs which are then handled correctly by the Agg PathIterator and other consumers of path data, such as `iter_segments()`.

- **codes**: {None, array_like}, optional
  - n-length array integers representing the codes of the path. If not None, codes must be the same length as vertices. If None, `vertices` will be treated as a series of line segments.

- **_interpolation_steps**: int, optional
  - Used as a hint to certain projections, such as Polar, that this path should be linearly interpolated immediately before drawing. This attribute is primarily an implementation detail and is not intended for public use.

- **closed**: bool, optional
  - If `codes` is None and closed is True, vertices will be treated as line segments of a closed polygon.

- **readonly**: bool, optional
  - Makes the path behave in an immutable way and sets the vertices and codes as read-only arrays.

```
CLOSEPOLY = 79
CURVE3 = 3
CURVE4 = 4
LINETO = 2
MOVETO = 1
```
**NUM_VERTICES_FOR_CODE** = {0: 1, 1: 1, 2: 1, 3: 2, 4: 3, 79: 1}
A dictionary mapping Path codes to the number of vertices that the code expects.

STOP = 0

**classmethod arc**(*theta1*, *theta2*, *n=None, is_wedge=False*)
Return an arc on the unit circle from angle *theta1* to angle *theta2* (in degrees).
If *n* is provided, it is the number of spline segments to make. If *n* is not provided, the number of
spline segments is determined based on the delta between *theta1* and *theta2*.

Masionobe, L. 2003. Drawing an elliptical arc using polylines, quadratic or cubic
Bezier curves.

**classmethod circle**(*center=(0.0, 0.0), radius=1.0, readonly=False*)
Return a Path representing a circle of a given radius and center.

**Parameters**

- **center**: pair of floats
  The center of the circle. Default (0, 0).

- **radius**: float
  The radius of the circle. Default is 1.

- **readonly**: bool
  Whether the created path should have the “readonly” argument set when cre-
  a ting the Path instance.

**Notes**

The circle is approximated using cubic Bezier curves. This uses 8 splines around the circle using
the approach presented here:

Lancaster, Don. Approximating a Circle or an Ellipse Using Four Bezier Cubic
Splines.

**cleaned**(*transform=None, remove_nans=False, clip=None, quantize=False, simplify=False,
curves=False, stroke_width=1.0, snap=False, sketch=None*)
Cleans up the path according to the parameters returning a new Path instance.

**See also:**

See **iter_segments()** for details of the keyword arguments.

**Returns** Path instance with cleaned up vertices and codes.

**clip_to_bbox**(*bbox, inside=True*)
Clip the path to the given bounding box.

The path must be made up of one or more closed polygons. This algorithm will not behave
correctly for unclosed paths.
If \textit{inside} is True, clip to the inside of the box, otherwise to the outside of the box.

\textbf{code_type}  
alias of \texttt{uint8}

\textbf{codes}  
The list of codes in the \texttt{Path} as a 1-D numpy array. Each code is one of \texttt{STOP}, \texttt{MOVETO}, \texttt{LINETO}, \texttt{CURVE3}, \texttt{CURVE4} or \texttt{CLOSEPOLY}. For codes that correspond to more than one vertex (\texttt{CURVE3} and \texttt{CURVE4}), that code will be repeated so that the length of \texttt{self.vertices} and \texttt{self.codes} is always the same.

\textbf{contains_path}(\texttt{path}, \texttt{transform=\texttt{None}})  
Returns whether this (closed) path completely contains the given path.  
If \texttt{transform} is not \texttt{None}, the path will be transformed before performing the test.

\textbf{contains_point}(\texttt{point}, \texttt{transform=\texttt{None}}, \texttt{radius=0.0})  
Returns whether the (closed) path contains the given point.  
If \texttt{transform} is not \texttt{None}, the path will be transformed before performing the test.  
\texttt{radius} allows the path to be made slightly larger or smaller.

\textbf{contains_points}(\texttt{points}, \texttt{transform=\texttt{None}}, \texttt{radius=0.0})  
Returns a bool array which is True if the (closed) path contains the corresponding point.  
If \texttt{transform} is not \texttt{None}, the path will be transformed before performing the test.  
\texttt{radius} allows the path to be made slightly larger or smaller.

\textbf{copy}()  
Returns a shallow copy of the \texttt{Path}, which will share the vertices and codes with the source \texttt{Path}.

\textbf{deepcopy}(\texttt{\texttt{memo}=\texttt{None}})  
Returns a deepcopy of the \texttt{Path}. The \texttt{Path} will not be readonly, even if the source \texttt{Path} is.

\textbf{get_extents}(\texttt{\texttt{transform}=\texttt{None}})  
Returns the extents \((x_{\text{min}}, y_{\text{min}}, x_{\text{max}}, y_{\text{max}})\) of the path.  
Unlike computing the extents on the \texttt{vertices} alone, this algorithm will take into account the curves and deal with control points appropriately.

\textbf{has_nonfinite}  
True if the vertices array has nonfinite values.

\textbf{classmethod hatch}(\texttt{hatchpattern}, \texttt{density=6})  
Given a hatch specifier, \texttt{hatchpattern}, generates a \texttt{Path} that can be used in a repeated hatching pattern. \texttt{density} is the number of lines per unit square.

\textbf{interpolated}(\texttt{steps})  
Returns a new path resampled to length \texttt{N} \times \texttt{steps}. Does not currently handle interpolating curves.

\textbf{intersects_bbox}(\texttt{bbox}, \texttt{filled=\texttt{True}})  
Returns True if this path intersects a given \texttt{Bbox}.
when True, treats the path as if it was filled. That is, if one path completely encloses the other, `intersects_path()` will return True.

**intersects_path(other, filled=True)**

Returns `True` if this path intersects another given path.

when True, treats the paths as if they were filled. That is, if one path completely encloses the other, `intersects_path()` will return True.

**iter_segments(transform=None, remove_nans=True, clip=None, snap=False, stroke_width=1.0, simplify=None, curves=True, sketch=None)**

Iterates over all of the curve segments in the path. Each iteration returns a 2-tuple `(vertices, code)`, where `vertices` is a sequence of 1 - 3 coordinate pairs, and `code` is one of the `Path` codes.

Additionally, this method can provide a number of standard cleanups and conversions to the path.

**Parameters**

**transform** : None or `Transform` instance

If not None, the given affine transformation will be applied to the path.

**remove_nans** : {False, True}, optional

If True, will remove all NaNs from the path and insert MOVETO commands to skip over them.

**clip** : None or sequence, optional

If not None, must be a four-tuple (x1, y1, x2, y2) defining a rectangle in which to clip the path.

**snap** : None or bool, optional

If None, auto-snap to pixels, to reduce fuzziness of rectilinear lines. If True, force snapping, and if False, don’t snap.

**stroke_width** : float, optional

The width of the stroke being drawn. Needed as a hint for the snapping algorithm.

**simplify** : None or bool, optional

If True, perform simplification, to remove vertices that do not affect the appearance of the path. If False, perform no simplification. If None, use the should_simplify member variable.

**curves** : {True, False}, optional

If True, curve segments will be returned as curve segments. If False, all curves will be converted to line segments.

**sketch** : None or sequence, optional

If not None, must be a 3-tuple of the form (scale, length, randomness), representing the sketch parameters.
**classmethod make_compound_path(**args**)  
Make a compound path from a list of Path objects.

**classmethod make_compound_path_from_polys(**XY**)  
Make a compound path object to draw a number of polygons with equal numbers of sides XY is a (numpolys x numsides x 2) numpy array of vertices. Return object is a Path

**readonly**  
True if the Path is read-only.

**should_simplify**  
True if the vertices array should be simplified.

**simplify_threshold**  
The fraction of a pixel difference below which vertices will be simplified out.

**to_polygons**(transform=None, width=0, height=0, closed_only=True)  
Convert this path to a list of polygons or polylines. Each polygon/polyline is an Nx2 array of vertices. In other words, each polygon has no MOVETO instructions or curves. This is useful for displaying in backends that do not support compound paths or Bezier curves, such as GDK.

If width and height are both non-zero then the lines will be simplified so that vertices outside of (0, 0), (width, height) will be clipped.

If closed_only is True (default), only closed polygons, with the last point being the same as the first point, will be returned. Any unclosed polylines in the path will be explicitly closed. If
closed_only is False, any unclosed polygons in the path will be returned as unclosed polygons, and the closed polygons will be returned explicitly closed by setting the last point to the same as the first point.

transformed(transform)
Return a transformed copy of the path.

See also:

matplotlib.transforms.TransformedPath A specialized path class that will cache the transformed result and automatically update when the transform changes.

classmethod unit_circle()
Return the readonly Path of the unit circle.

For most cases, Path.circle() will be what you want.

classmethod unit_circle_righthalf()
Return a Path of the right half of a unit circle. The circle is approximated using cubic Bezier curves. This uses 4 splines around the circle using the approach presented here:

Lancaster, Don. Approximating a Circle or an Ellipse Using Four Bezier Cubic Splines.

classmethod unit_rectangle()
Return a Path instance of the unit rectangle from (0, 0) to (1, 1).

classmethod unit_regular_asterisk(numVertices)
Return a Path for a unit regular asterisk with the given numVertices and radius of 1.0, centered at (0, 0).

classmethod unit_regular_polygon(numVertices)
Return a Path instance for a unit regular polygon with the given numVertices and radius of 1.0, centered at (0, 0).

classmethod unit_regular_star(numVertices, innerCircle=0.5)
Return a Path for a unit regular star with the given numVertices and radius of 1.0, centered at (0, 0).

vertices
The list of vertices in the Path as an Nx2 numpy array.

classmethod wedge(theta1, theta2, n=None)
Return a wedge of the unit circle from angle theta1 to angle theta2 (in degrees).

If n is provided, it is the number of spline segments to make. If n is not provided, the number of spline segments is determined based on the delta between theta1 and theta2.

matplotlib.path.get_path_collection_extents(master_transform, paths, transforms, offsets, offset_transform)
Given a sequence of Path objects, Transform objects and offsets, as found in a PathCollection, returns the bounding box that encapsulates all of them.

master_transform is a global transformation to apply to all paths
paths is a sequence of \textit{Path} instances.

transforms is a sequence of \textit{Affine2D} instances.

offsets is a sequence of (x, y) offsets (or an Nx2 array)

offset_transform is a \textit{Affine2D} to apply to the offsets before applying the offset to the path.

The way that paths, transforms and offsets are combined follows the same method as for collections. Each is iterated over independently, so if you have 3 paths, 2 transforms and 1 offset, their combinations are as follows:

(A, A, A), (B, B, A), (C, A, A)

\texttt{matplotlib.path.get_paths_extents(paths, transforms=[])}

Given a sequence of \textit{Path} objects and optional \textit{Transform} objects, returns the bounding box that encapsulates all of them.

paths is a sequence of \textit{Path} instances.

transforms is an optional sequence of \textit{Affine2D} instances to apply to each path.
43.1 matplotlib.patheffects

Defines classes for path effects. The path effects are supported in Text, Line2D and Patch.

```python
class matplotlib.patheffects(AbstractPathEffect(offset=(0.0, 0.0))
    Bases: object

    A base class for path effects.
    Subclasses should override the draw_path method to add effect functionality.

    Parameters offset : pair of floats
        The offset to apply to the path, measured in points.

draw_path(renderer, gc, tpath, affine, rgbFace=None)
    Derived should override this method. The arguments are the same as matplotlib.backend_bases.RendererBase.draw_path() except the first argument is a renderer.
```

```python
class matplotlib.patheffects.Normal(offset=(0.0, 0.0))
    Bases: matplotlib.patheffects.AbstractPathEffect

    The “identity” PathEffect.
    The Normal PathEffect’s sole purpose is to draw the original artist with no special path effect.

    Parameters offset : pair of floats
        The offset to apply to the path, measured in points.
```

```python
class matplotlib.patheffects.PathEffectRenderer(path_effects, renderer)
    Bases: matplotlib.backend_bases.RendererBase

    Implements a Renderer which contains another renderer.
    This proxy then intercepts draw calls, calling the appropriate AbstractPathEffect draw method.
```

**Note:** Not all methods have been overridden on this RendererBase subclass. It may be necessary to add further methods to extend the PathEffects capabilities further.
Parameters **path_effects** : iterable of *AbstractPathEffect*

The path effects which this renderer represents.

**renderer** : *matplotlib.backend_bases.RendererBase* instance

**copy_with_path_effect**(path_effects)

**draw_markers**(gc, marker_path, marker_trans, path, *args, **kwargs)

**draw_path**(gc, tpath, affine, rgbFace=None)

**draw_path_collection**(gc, master_transform, paths, *args, **kwargs)

**new_gc**()

**points_to_pixels**(points)

class *matplotlib.patheffects.PathPatchEffect*(offset=(0, 0), **kwargs)

Bases: *matplotlib.patheffects.AbstractPathEffect*

Draws a *PathPatch* instance whose Path comes from the original PathEffect artist.

**Parameters**

**offset** : pair of floats

The offset to apply to the path, in points.

**kwargs** :

All keyword arguments are passed through to the *PathPatch* constructor.
The properties which cannot be overridden are “path”, “clip_box” “transform” and “clip_path”.

**draw_path**(renderer, gc, tpath, affine, rgbFace)

class *matplotlib.patheffects.SimpleLineShadow*(offset=(2, -2), shadow_color='k', alpha=0.3, rho=0.3, **kwargs)

Bases: *matplotlib.patheffects.AbstractPathEffect*

A simple shadow via a line.

**Parameters**

**offset** : pair of floats

The offset to apply to the path, in points.

**shadow_color** : color

The shadow color. Default is black. A value of *None* takes the original artist’s color with a scale factor of *rho*.

**alpha** : float
The alpha transparency of the created shadow patch. Default is 0.3.

rho : float

A scale factor to apply to the rgbFace color if shadow_rgbFace is None. Default is 0.3.

**kwargs

Extra keywords are stored and passed through to AbstractPathEffect. _update_gc().

draw_path(renderer, gc, tpath, affine, rgbFace)

Overrides the standard draw_path to add the shadow offset and necessary color changes for the shadow.

class matplotlib.pathEffects.SimplePatchShadow(offset=(2, -2), shadow_rgbFace=None, alpha=None, rho=0.3, **kwargs)

Bases: matplotlib.pathEffects.AbstractPathEffect

A simple shadow via a filled patch.

Parameters offset : pair of floats

The offset of the shadow in points.

shadow_rgbFace : color

The shadow color.

alpha : float

The alpha transparency of the created shadow patch. Default is 0.3. http://matplotlib.1069221.n5.nabble.com/path-effects-question-td27630.html

rho : float

A scale factor to apply to the rgbFace color if shadow_rgbFace is not specified. Default is 0.3.

**kwargs

Extra keywords are stored and passed through to AbstractPathEffect. _update_gc().

draw_path(renderer, gc, tpath, affine, rgbFace)

Overrides the standard draw_path to add the shadow offset and necessary color changes for the shadow.

class matplotlib.pathEffects.Stroke(offset=(0, 0), **kwargs)

Bases: matplotlib.pathEffects.AbstractPathEffect

A line based PathEffect which re-draws a stroke.

The path will be stroked with its gc updated with the given keyword arguments, i.e., the keyword arguments should be valid gc parameter values.

draw_path(renderer, gc, tpath, affine, rgbFace)

draw the path with updated gc.
class matplotlib.patheffects.withSimplePatchShadow

Bases: matplotlib.patheffects.SimplePatchShadow

Adds a simple SimplePatchShadow and then draws the original Artist to avoid needing to call Normal.

**Parameters**
- **offset**: pair of floats
  - The offset of the shadow in points.
- **shadow_rgbFace**: color
  - The shadow color.
- **alpha**: float
  - The alpha transparency of the created shadow patch. Default is 0.3. [Link](http://matplotlib.1069221.n5.nabble.com/path-effects-question-td27630.html)
- **rho**: float
  - A scale factor to apply to the rgbFace color if shadow_rgbFace is not specified. Default is 0.3.
- **kwargs**
  - Extra keywords are stored and passed through to AbstractPathEffect._update_gc().

**draw_path**(renderer, gc, tpath, affine, rgbFace)

class matplotlib.patheffects.withStroke

Bases: matplotlib.patheffects.Stroke

Adds a simple Stroke and then draws the original Artist to avoid needing to call Normal.

The path will be stroked with its gc updated with the given keyword arguments, i.e., the keyword arguments should be valid gc parameter values.

**draw_path**(renderer, gc, tpath, affine, rgbFace)
44.1 matplotlib.projections

```python
class matplotlib.projections.ProjectionRegistry
    Bases: object
    Manages the set of projections available to the system.

    get_projection_class(name)
        Get a projection class from its name.

    get_projection_names()
        Get a list of the names of all projections currently registered.

    register(*projections)
        Register a new set of projection(s).

matplotlib.projections.get_projection_class(projection=None)
    Get a projection class from its name.
    
    If `projection` is None, a standard rectilinear projection is returned.

matplotlib.projections.get_projection_names()
    Get a list of acceptable projection names.

matplotlib.projections.process_projection_requirements(figure, *args, **kwargs)
    Handle the args/kwargs to for add_axes/add_subplot(gca, returning:

    (axes_proj_class, proj_class_kwargs, proj_stack_key)

    Which can be used for new axes initialization/identification.

Note: `kwargs` is modified in place.
```
44.2 matplotlib.projections.polar

class matplotlib.projections.polar.InvertedPolarTransform(axis=None, use_rmin=True)

Bases: matplotlib.transforms.Transform

The inverse of the polar transform, mapping Cartesian coordinate space \( x \) and \( y \) back to \( \theta \) and \( r \).

inverted()

Return the corresponding inverse transformation.

The return value of this method should be treated as temporary. An update to \( self \) does not cause a corresponding update to its inverted copy.

\[ x = self.inverted().transform(self.transform(x)) \]

class matplotlib.projections.polar.PolarAffine(scale_transform, limits)

Bases: matplotlib.transforms.Affine2DBase

The affine part of the polar projection. Scales the output so that maximum radius rests on the edge of the axes circle.

\( \text{limits} \) is the view limit of the data. The only part of its bounds that is used is ymax (for the radius maximum). The theta range is always fixed to (0, 2\( \pi \)).

class matplotlib.projections.polar.PolarAxes(*args, **kwargs)

Bases: matplotlib.axes._axes.Axes

A polar graph projection, where the input dimensions are \( \theta, r \).

Theta starts pointing east and goes anti-clockwise.

class InvertedPolarTransform(axis=None, use_rmin=True)

Bases: matplotlib.transforms.Transform

The inverse of the polar transform, mapping Cartesian coordinate space \( x \) and \( y \) back to \( \theta \) and \( r \).

inverted()

Return the corresponding inverse transformation.
The return value of this method should be treated as temporary. An update to self does not cause a corresponding update to its inverted copy.

\[ x \equiv \text{self.inverted().transform(self.transform(x))} \]

**transform_non_affine** *(xy)*

Performs only the non-affine part of the transformation.

\[ \text{transform(values)} \text{ is always equivalent to transform_affine(transform_non_affine(values))}. \]

In non-affine transformations, this is generally equivalent to transform(values). In affine transformations, this is always a no-op.

Accepts a numpy array of shape \((N \times \text{input_dims})\) and returns a numpy array of shape \((N \times \text{output_dims})\).

Alternatively, accepts a numpy array of length \(\text{input_dims}\) and returns a numpy array of length \(\text{output_dims}\).

class **PolarAffine** *(scale_transform, limits)*

Bases: matplotlib.transforms.Affine2DBase

The affine part of the polar projection. Scales the output so that maximum radius rests on the edge of the axes circle.

*limits* is the view limit of the data. The only part of its bounds that is used is ymax (for the radius maximum). The theta range is always fixed to (0, 2pi).

**get_matrix** *

Get the Affine transformation array for the affine part of this transform.

class **PolarTransform** *(axis=None, use_rmin=True)*

Bases: matplotlib.transforms.Transform

The base polar transform. This handles projection \(\theta\) and \(r\) into Cartesian coordinate space \(x\) and \(y\), but does not perform the ultimate affine transformation into the correct position.

**inverted** *

Return the corresponding inverse transformation.

The return value of this method should be treated as temporary. An update to self does not cause a corresponding update to its inverted copy.

\[ x \equiv \text{self.inverted().transform(self.transform(x))} \]

**transform_non_affine** *(tr)*

Performs only the non-affine part of the transformation.

\[ \text{transform(values)} \text{ is always equivalent to transform_affine(transform_non_affine(values))}. \]

In non-affine transformations, this is generally equivalent to transform(values). In affine transformations, this is always a no-op.

Accepts a numpy array of shape \((N \times \text{input_dims})\) and returns a numpy array of shape \((N \times \text{output_dims})\).

Alternatively, accepts a numpy array of length \(\text{input_dims}\) and returns a numpy array of length \(\text{output_dims}\).
transform_path_non_affine(path)
Returns a path, transformed only by the non-affine part of this transform.

path: a Path instance.

transform_path(path) is equivalent to transform_path_affine(transform_path_non_affine(values))

class RadialLocator(base)
Bases: matplotlib.ticker.Locator
Used to locate radius ticks.
Ensures that all ticks are strictly positive. For all other tasks, it delegates to the base Locator (which may be different depending on the scale of the r-axis).

class ThetaFormatter
Bases: matplotlib.ticker.Formatter
Used to format the theta tick labels. Converts the native unit of radians into degrees and adds a degree symbol.

can_pan()
Return True if this axes supports the pan/zoom button functionality.
For polar axes, this is slightly misleading. Both panning and zooming are performed by the same button. Panning is performed in azimuth while zooming is done along the radial.

can_zoom()
Return True if this axes supports the zoom box button functionality.
Polar axes do not support zoom boxes.

format_coord(theta, r)
Return a format string formatting the coordinate using Unicode characters.

get_data_ratio()
Return the aspect ratio of the data itself. For a polar plot, this should always be 1.0

get_rlabel_position()
Returns float
The theta position of the radius labels in degrees.

get_theta_direction()
Get the direction in which theta increases.
-1: Theta increases in the clockwise direction
1: Theta increases in the counterclockwise direction

get_theta_offset()
Get the offset for the location of 0 in radians.

set_rgrids(radii, labels=None, angle=None, fmt=None, **kwargs)
Set the radial locations and labels of the r grids.
The labels will appear at radial distances radii at the given angle in degrees.
labels, if not None, is a \texttt{len(radii)} list of strings of the labels to use at each radius.

If \texttt{labels} is None, the built-in formatter will be used.

Return value is a list of tuples (line, label), where line is \texttt{Line2D} instances and the label is \texttt{Text} instances.

kwargs are optional text properties for the labels:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{agg_filter}</td>
<td>unknown</td>
</tr>
<tr>
<td>\texttt{alpha}</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>\texttt{animated}</td>
<td>[True</td>
</tr>
<tr>
<td>\texttt{axes}</td>
<td>an \texttt{Axes} instance</td>
</tr>
<tr>
<td>\texttt{backgroundcolor}</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>\texttt{bbox}</td>
<td>FancyBboxPatch prop dict</td>
</tr>
<tr>
<td>\texttt{clip_box}</td>
<td>a \texttt{matplotlib.transforms.Bbox} instance</td>
</tr>
<tr>
<td>\texttt{clip_on}</td>
<td>[True</td>
</tr>
<tr>
<td>\texttt{clip_path}</td>
<td>[(\texttt{Path}, \texttt{Transform})</td>
</tr>
<tr>
<td>\texttt{color}</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>\texttt{contains}</td>
<td>a callable function</td>
</tr>
<tr>
<td>\texttt{family} or \texttt{fontfamily} or \texttt{fontname} or \texttt{name}</td>
<td>[\texttt{FONTNAME}</td>
</tr>
<tr>
<td>\texttt{figure}</td>
<td>a \texttt{matplotlib.figure.Figure} instance</td>
</tr>
<tr>
<td>\texttt{fontproperties} or \texttt{font_properties}</td>
<td>a \texttt{matplotlib.font_manager.FontProperties} instance</td>
</tr>
<tr>
<td>\texttt{gid}</td>
<td>an id string</td>
</tr>
<tr>
<td>\texttt{horizontalalignment} or \texttt{ha}</td>
<td>[‘center’</td>
</tr>
<tr>
<td>\texttt{label}</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>\texttt{linespacing}</td>
<td>float (multiple of font size)</td>
</tr>
<tr>
<td>\texttt{multialignment}</td>
<td>[‘left’</td>
</tr>
<tr>
<td>\texttt{path_effects}</td>
<td>unknown</td>
</tr>
<tr>
<td>\texttt{picker}</td>
<td>[None</td>
</tr>
<tr>
<td>\texttt{position}</td>
<td>(x,y)</td>
</tr>
<tr>
<td>\texttt{rasterized}</td>
<td>[True</td>
</tr>
<tr>
<td>\texttt{rotation}</td>
<td>[ angle in degrees</td>
</tr>
<tr>
<td>\texttt{rotation_mode}</td>
<td>unknown</td>
</tr>
<tr>
<td>\texttt{size} or \texttt{fontsize}</td>
<td>[size in points</td>
</tr>
<tr>
<td>\texttt{sketch_params}</td>
<td>unknown</td>
</tr>
<tr>
<td>\texttt{snap}</td>
<td>unknown</td>
</tr>
<tr>
<td>\texttt{stretch} or \texttt{fontstretch}</td>
<td>[a numeric value in range 0-1000</td>
</tr>
<tr>
<td>\texttt{style} or \texttt{fontstyle}</td>
<td>[‘normal’</td>
</tr>
<tr>
<td>\texttt{text}</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>\texttt{transform}</td>
<td>\texttt{Transform} instance</td>
</tr>
<tr>
<td>\texttt{url}</td>
<td>a url string</td>
</tr>
<tr>
<td>\texttt{usetex}</td>
<td>unknown</td>
</tr>
<tr>
<td>\texttt{variant} or \texttt{fontvariant}</td>
<td>[‘normal’</td>
</tr>
<tr>
<td>\texttt{verticalalignment} or \texttt{ma or va}</td>
<td>[‘center’</td>
</tr>
<tr>
<td>\texttt{visible}</td>
<td>[True</td>
</tr>
</tbody>
</table>
Table 44.1 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wrap</td>
<td>unknown</td>
</tr>
<tr>
<td>x</td>
<td>float</td>
</tr>
<tr>
<td>y</td>
<td>float</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

ACCEPTS: sequence of floats

```
set_rlabel_position(value)
```
Updates the theta position of the radius labels.

Parameters `value` : number

The angular position of the radius labels in degrees.

```
set_theta_direction(direction)
```
Set the direction in which theta increases.

- clockwise, -1: Theta increases in the clockwise direction
- counterclockwise, anticlockwise, 1: Theta increases in the counterclockwise direction

```
set_theta_offset(offset)
```
Set the offset for the location of 0 in radians.

```
set_theta_zero_location(loc)
```
Sets the location of theta’s zero. (Calls set_theta_offset with the correct value in radians under the hood.)

May be one of “N”, “NW”, “W”, “SW”, “S”, “SE”, “E”, or “NE”.

```
set_thetagrids(angles, labels=None, frac=None, fmt=None, **kwargs)
```
Set the angles at which to place the theta grids (these gridlines are equal along the theta dimension). `angles` is in degrees.

`labels`, if not None, is a `len(angles)` list of strings of the labels to use at each angle.

If `labels` is None, the labels will be `fmt % angle`

`frac` is the fraction of the polar axes radius at which to place the label (1 is the edge). e.g., 1.05 is outside the axes and 0.95 is inside the axes.

Return value is a list of tuples (`line`, `label`), where `line` is `Line2D` instances and the `label` is `Text` instances.

kwargs are optional text properties for the labels:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>backgroundcolor</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>bbox</td>
<td>FancyBboxPatch prop dict</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>family or fontfamily or fontname or name</td>
<td>[FONTNAME</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fontproperties or font_properties</td>
<td>a matplotlib.font_manager.FontProperties instance</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>horizontalalignment or ha</td>
<td>[ ‘center’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linespacing</td>
<td>float (multiple of font size)</td>
</tr>
<tr>
<td>multialignment</td>
<td>[’left’</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>position</td>
<td>(x,y)</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>rotation</td>
<td>[ angle in degrees</td>
</tr>
<tr>
<td>rotation_mode</td>
<td>unknown</td>
</tr>
<tr>
<td>size or fontsize</td>
<td>[size in points</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>stretch or fontstretch</td>
<td>[a numeric value in range 0-1000</td>
</tr>
<tr>
<td>style or fontstyle</td>
<td>[ ‘normal’</td>
</tr>
<tr>
<td>text</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>usetex</td>
<td>unknown</td>
</tr>
<tr>
<td>variant or fontvariant</td>
<td>[ ‘normal’</td>
</tr>
<tr>
<td>verticalalignment or ma or va</td>
<td>[ ‘center’</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>weight or fontweight</td>
<td>[a numeric value in range 0-1000</td>
</tr>
<tr>
<td>wrap</td>
<td>unknown</td>
</tr>
<tr>
<td>x</td>
<td>float</td>
</tr>
<tr>
<td>y</td>
<td>float</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

ACCEPTS: sequence of floats
class matplotlib.projections.polar.PolarTransform(axis=None, use_rmin=True)

Bases: matplotlib.transforms.Transform

The base polar transform. This handles projection \theta and \rho into Cartesian coordinate space \(x\) and \(y\), but does not perform the ultimate affine transformation into the correct position.

inverted()

Return the corresponding inverse transformation.

The return value of this method should be treated as temporary. An update to self does not cause a corresponding update to its inverted copy.

\[ x \equiv \text{self.inverted().transform(self.transform(x))} \]

transform_non_affine(tr)

Performs only the non-affine part of the transformation.

\[ \text{transform(values)} \text{ is always equivalent to transform_affine(transform_non_affine(values))}. \]

In non-affine transformations, this is generally equivalent to transform(values). In affine transformations, this is always a no-op.

Accepts a numpy array of shape (\(N\times\text{input_dims}\)) and returns a numpy array of shape (\(N\times\text{output_dims}\)).

Alternatively, accepts a numpy array of length \text{input_dims} and returns a numpy array of length \text{output_dims}.

transform_path_non_affine(path)

Returns a path, transformed only by the non-affine part of this transform.

\[ \text{transform_path(path)} \text{ is equivalent to transform_path_affine(transform_path_non_affine(values))}. \]

class matplotlib.projections.polar.RadialLocator(base)

Bases: matplotlib.ticker.Locator

Used to locate radius ticks.

Ensures that all ticks are strictly positive. For all other tasks, it delegates to the base Locator (which may be different depending on the scale of the \(r\)-axis).

class matplotlib.projections.polar.ThaFormatter

Bases: matplotlib.ticker.Formatter

Used to format the \(\theta\) tick labels. Converts the native unit of radians into degrees and adds a degree symbol.
45.1 matplotlib.pyplot

Provides a MATLAB-like plotting framework.

pylab combines pyplot with numpy into a single namespace. This is convenient for interactive work, but for programming it is recommended that the namespaces be kept separate, e.g.:

```python
import numpy as np
import matplotlib.pyplot as plt

x = np.arange(0, 5, 0.1);
y = np.sin(x)
plt.plot(x, y)
```

`matplotlib.pyplot.acorr(x, hold=None, data=None, **kwargs)`

Plot the autocorrelation of x.

**Parameters**
- `x`: sequence of scalar
  - `hold`: boolean, optional, *deprecated*, default: True
  - `detrend`: callable, optional, default: `mlab.detrend_none`
    - x is detrended by the detrend callable. Default is no normalization.
  - `normed`: boolean, optional, default: True
    - if True, input vectors are normalised to unit length.
  - `usevlines`: boolean, optional, default: True
    - if True, Axes.vlines is used to plot the vertical lines from the origin to the acorr. Otherwise, Axes.plot is used.
  - `maxlags`: integer, optional, default: 10
    - number of lags to show. If None, will return all 2 * len(x) - 1 lags.

**Returns**
- `(lags, c, line, b)` where:
  - `lags` are a length 2*maxlags+1 lag vector.
  - `c` is the 2*maxlags+1 auto correlation vector
line is a Line2D instance returned by plot.

b is the x-axis.

Other Parameters

- **linestyle**: Line2D prop, optional, default: None
  
  Only used if usevlines is False.
  
  - **marker**: string, optional, default: ‘o’

Notes

The cross correlation is performed with numpy.correlate() with mode = 2.

Examples

xcorr is top graph, and acorr is bottom graph.

Note: In addition to the above described arguments, this function can take a data keyword argument. If such a data argument is given, the following arguments are replaced by data[<arg>]:

- All arguments with the following names: ‘x’.
matplotlib.pyplot.angle_spectrum(x, Fs=None, Fc=None, window=None, pad_to=None, sides=None, hold=None, data=None, **kwargs)

Plot the angle spectrum.

Call signature:

```python
angle_spectrum(x, Fs=2, Fc=0, window=mlab.window_hanning,
               pad_to=None, sides='default', **kwargs)
```

Compute the angle spectrum (wrapped phase spectrum) of $x$. Data is padded to a length of $pad\_to$ and the windowing function $window$ is applied to the signal.

**Parameters**

$x$ : 1-D array or sequence

Array or sequence containing the data.

$Fs$ : scalar

The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.

$window$ : callable or ndarray

A function or a vector of length $NFFT$. To create window vectors see $window\_hanning()$, $window\_none()$, $numpy.blackman()$, $numpy.hamming()$, $numpy.bartlett()$, $scipy.signal()$, $scipy.signal.get\_window()$, etc. The default is $window\_hanning()$. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

$sides$ : [‘default’ | ‘onesided’ | ‘twosided’ ]

Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. ‘onesided’ forces the return of a one-sided spectrum, while ‘twosided’ forces two-sided.

$pad\_to$ : integer

The number of points to which the data segment is padded when performing the FFT. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the $n$ parameter in the call to fft(). The default is None, which sets $pad\_to$ equal to the length of the input signal (i.e. no padding).

$Fc$ : integer

The center frequency of $x$ (defaults to 0), which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

**kwargs :

Keyword arguments control the $Line2D$ properties:
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>['butt’</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>['miter’</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>drawstyle</td>
<td>['default’</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>['full’</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>[‘solid’</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>marker</td>
<td>A valid marker style</td>
</tr>
<tr>
<td>markeredgecolor or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>[None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt’</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter’</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

**Returns spectrum**: 1-D array

The values for the angle spectrum in radians (real valued)
**freqs** : 1-D array

The frequencies corresponding to the elements in `spectrum`.

**line** : a `Line2D` instance

The line created by this function.

See also:

- `magnitude_spectrum()` `angle_spectrum()` plots the magnitudes of the corresponding frequencies.
- `phase_spectrum()` `phase_spectrum()` plots the unwrapped version of this function.
- `specgram()` `specgram()` can plot the angle spectrum of segments within the signal in a colormap.

In addition to the above described arguments, this function can take a `data` keyword argument. If such a `data` argument is given, the following arguments are replaced by `data[<arg>]: * All arguments with the following names: ‘x’.

**Examples**
Additional kwargs are passed to `Text`.

**Parameters**

- **s : str**
  
  The text of the annotation

- **xy : iterable**
  
  Length 2 sequence specifying the \((x,y)\) point to annotate

- **xytext : iterable, optional**
  
  Length 2 sequence specifying the \((x,y)\) to place the text at. If None, defaults to \(xy\).

- **xycoords : str, Artist, Transform, callable or tuple, optional**
  
  The coordinate system that \(xy\) is given in.

  For a str the allowed values are:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘figure points’</td>
<td>points from the lower left of the figure</td>
</tr>
<tr>
<td>‘figure pixels’</td>
<td>pixels from the lower left of the figure</td>
</tr>
<tr>
<td>‘figure fraction’</td>
<td>fraction of figure from lower left</td>
</tr>
<tr>
<td>‘axes points’</td>
<td>points from lower left corner of axes</td>
</tr>
<tr>
<td>‘axes pixels’</td>
<td>pixels from lower left corner of axes</td>
</tr>
<tr>
<td>‘axes fraction’</td>
<td>fraction of axes from lower left</td>
</tr>
<tr>
<td>‘data’</td>
<td>use the coordinate system of the object being annotated (default)</td>
</tr>
<tr>
<td>‘polar’</td>
<td>((\theta,r)) if not native ‘data’ coordinates</td>
</tr>
</tbody>
</table>

  If a `Artist` object is passed in the units are fraction if it’s bounding box.

  If a `Transform` object is passed in use that to transform \(xy\) to screen coordinates

  If a callable it must take a `RendererBase` object as input and return a `Transform` or `Bbox` object

  If a tuple must be length 2 tuple of str, `Artist`, `Transform` or callable objects. The first transform is used for the \(x\) coordinate and the second for \(y\).

  See `Advanced Annotation` for more details.

  Defaults to 'data'

- **textcoords : str, Artist, Transform, callable or tuple, optional**
  
  The coordinate system that \(xytext\) is given, which may be different than the coordinate system used for \(xy\).

  All \(xycoords\) values are valid as well as the following strings:
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'offset points'</td>
<td>offset (in points) from the xy value</td>
</tr>
<tr>
<td>'offset pixels'</td>
<td>offset (in pixels) from the xy value</td>
</tr>
</tbody>
</table>

defaults to the input of xycoords

**arrowprops** : dict, optional

If not None, properties used to draw a *FancyArrowPatch* arrow between xy and xytext.

If arrowprops does not contain the key 'arrowstyle' the allowed keys are:

<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>width</td>
<td>the width of the arrow in points</td>
</tr>
<tr>
<td>headwidth</td>
<td>the width of the base of the arrow head in points</td>
</tr>
<tr>
<td>headlength</td>
<td>the length of the arrow head in points</td>
</tr>
<tr>
<td>shrink</td>
<td>fraction of total length to 'shrink' from both ends</td>
</tr>
<tr>
<td>?</td>
<td>any key to <em>matplotlib.patches.FancyArrowPatch</em></td>
</tr>
</tbody>
</table>

If the arrowprops contains the key 'arrowstyle' the above keys are forbidden. The allowed values of 'arrowstyle' are:

<table>
<thead>
<tr>
<th>Name</th>
<th>Attr</th>
</tr>
</thead>
<tbody>
<tr>
<td>'-'</td>
<td>None</td>
</tr>
<tr>
<td>'-&gt;'</td>
<td>head_length=0.4,head_width=0.2</td>
</tr>
<tr>
<td>'-[-'</td>
<td>widthB=1.0,lengthB=0.2,angleB=None</td>
</tr>
<tr>
<td>'[</td>
<td>-'</td>
</tr>
<tr>
<td>'-</td>
<td>&gt;'</td>
</tr>
<tr>
<td>'&lt;-'</td>
<td>head_length=0.4,head_width=0.2</td>
</tr>
<tr>
<td>'&lt;-&gt;'</td>
<td>head_length=0.4,head_width=0.2</td>
</tr>
<tr>
<td>'&lt;</td>
<td>- '</td>
</tr>
<tr>
<td>'&lt;</td>
<td>-&gt; '</td>
</tr>
<tr>
<td>'&lt;</td>
<td>-&gt; '</td>
</tr>
<tr>
<td>'&lt;</td>
<td>-&gt; '</td>
</tr>
<tr>
<td>'&lt;</td>
<td>-&gt; '</td>
</tr>
<tr>
<td>'fancy'</td>
<td>head_length=0.4,head_width=0.4,tail_width=0.4</td>
</tr>
<tr>
<td>'simple'</td>
<td>head_length=0.5,head_width=0.5,tail_width=0.2</td>
</tr>
<tr>
<td>'wedge'</td>
<td>tail_width=0.3,shrink_factor=0.5</td>
</tr>
</tbody>
</table>

Valid keys for *FancyArrowPatch* are:

<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>arrowstyle</td>
<td>the arrow style</td>
</tr>
<tr>
<td>connectionstyle</td>
<td>the connection style</td>
</tr>
<tr>
<td>relpos</td>
<td>default is (0.5, 0.5)</td>
</tr>
<tr>
<td>patchA</td>
<td>default is bounding box of the text</td>
</tr>
<tr>
<td>patchB</td>
<td>default is None</td>
</tr>
<tr>
<td>shrinkA</td>
<td>default is 2 points</td>
</tr>
<tr>
<td>shrinkB</td>
<td>default is 2 points</td>
</tr>
<tr>
<td>mutation_scale</td>
<td>default is text size (in points)</td>
</tr>
<tr>
<td>mutation_aspect</td>
<td>default is 1.</td>
</tr>
<tr>
<td>?</td>
<td>any key for <em>matplotlib.patches.PathPatch</em></td>
</tr>
</tbody>
</table>
Defaults to None

**annotation_clip** : bool, optional

Controls the visibility of the annotation when it goes outside the axes area.

If `True`, the annotation will only be drawn when the `xy` is inside the axes. If `False`, the annotation will always be drawn regardless of its position.

The default is `None`, which behave as `True` only if `xycoords` is “data”.

**Returns** Annotation

`matplotlib.pyplot.arrow(x, y, dx, dy, hold=None, **kwargs)`

Add an arrow to the axes.

Draws arrow on specified axis from `(x, y)` to `(x + dx, y + dy)`. Uses FancyArrow patch to construct the arrow.

**Parameters**

- `x` : float
  
  X-coordinate of the arrow base

- `y` : float
  
  Y-coordinate of the arrow base

- `dx` : float
  
  Length of arrow along x-coordinate

- `dy` : float
  
  Length of arrow along y-coordinate

**Returns**

`a` : FancyArrow

patches.FancyArrow object

**Other Parameters**

Optional `kwargs` (inherited from FancyArrow patch) control the arrow construction and properties:

**Constructor arguments**

- `width` : float (default: 0.001) width of full arrow tail

- `length_includes_head` : [True | False] (default: False) True if head is to be counted in calculating the length.

- `head_width` : float or None (default: 3*width) total width of the full arrow head

- `head_length` : float or None (default: 1.5 * head_width) length of arrow head

- `shape` : ['full', 'left', 'right'] (default: ‘full’) draw the left-half, right-half, or full arrow
overhang: float (default: 0) fraction that the arrow is swept back (0 overhang means triangular shape). Can be negative or greater than one.

head_starts_at_zero: [True | False] (default: False) if True, the head starts being drawn at coordinate 0 instead of ending at coordinate 0.

Other valid kwargs (inherited from :class:`Patch`) are:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>capstyle</td>
<td>[‘butt’</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor</td>
<td>mpl color spec, None, ‘none’, or ‘auto’</td>
</tr>
<tr>
<td>facecolor</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[‘/’</td>
</tr>
<tr>
<td>joinstyle</td>
<td>[‘miter’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyles</td>
<td>[‘solid’</td>
</tr>
<tr>
<td>linewidth</td>
<td>float or None for default</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

Notes

The resulting arrow is affected by the axes aspect ratio and limits. This may produce an arrow whose head is not square with its stem. To create an arrow whose head is square with its stem,
use `annotate()` for example:

```python
ax.annotate('', xy=(0.5, 0.5), xytext=(0, 0),
            arrowprops=dict(arrowstyle='->'))
```

**Examples**

```
A3
 rAC
 rAG
 rAT
 rCA
 rCG
 rCT
```

```
T3
 rAC
 rGT
 rCT
```

```
G3
 rGA
 rGC
 rGT
```

```
C3
 rGG
 rCG
 rTC
```

```python
matplotlib.pyplot.autoscale(enable=True, axis='both', tight=None)
```

Autoscale the axis view to the data (toggle).

Convenience method for simple axis view autoscaling. It turns autoscaling on or off, and then, if autoscaling for either axis is on, it performs the autoscaling on the specified axis or axes.

**enable:** [True | False | None] True (default) turns autoscaling on, False turns it off. None leaves the autoscaling state unchanged.

**axis:** ['x' | 'y' | 'both'] which axis to operate on; default is 'both'

**tight:** [True | False | None] If True, set view limits to data limits; if False, let the locator and margins expand the view limits; if None, use tight scaling if the only artist is an image, otherwise treat `tight` as False. The `tight` setting is retained for future autoscaling until it is explicitly changed.

Returns None.

```python
matplotlib.pyplot.autumn()
```

set the default colormap to autumn and apply to current image if any. See help(colormaps) for more information
matplotlib.pyplot.axes(*args, **kwargs)

Add an axes to the figure.

The axes is added at position rect specified by:

- `axes()` by itself creates a default full subplot(111) window axis.
- `axes(rect, facecolor='w')` where `rect = [left, bottom, width, height]` in normalized (0, 1) units. `facecolor` is the background color for the axis, default white.
- `axes(h)` where `h` is an axes instance makes `h` the current axis. An `Axes` instance is returned.

<table>
<thead>
<tr>
<th>kwarg</th>
<th>Accepts</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>facecolor</td>
<td>color</td>
<td>the axes background color</td>
</tr>
<tr>
<td>frameon</td>
<td>[True</td>
<td>False]</td>
</tr>
<tr>
<td>sharex</td>
<td>otherax</td>
<td>current axes shares xaxis attribute with otherax</td>
</tr>
<tr>
<td>sharey</td>
<td>otherax</td>
<td>current axes shares yaxis attribute with otherax</td>
</tr>
<tr>
<td>polar</td>
<td>[True</td>
<td>False]</td>
</tr>
</tbody>
</table>
| aspect  | [str | num]   | ['equal', 'auto'] or a number. If a number the ratio of x-unit/y-unit in screen-space. Also see `set_aspect()`.

Examples:

- `examples/pylab_examples/axes_demo.py` places custom axes.
- `examples/pylab_examples/shared_axis_demo.py` uses `sharex` and `sharey`.

matplotlib.pyplot.axhline(y=0, xmin=0, xmax=1, hold=None, **kwargs)

Add a horizontal line across the axis.

**Parameters**

- `y`: scalar, optional, default: 0
  - y position in data coordinates of the horizontal line.
- `xmin` : scalar, optional, default: 0
  - Should be between 0 and 1, 0 being the far left of the plot, 1 the far right of the plot.
- `xmax` : scalar, optional, default: 1
  - Should be between 0 and 1, 0 being the far left of the plot, 1 the far right of the plot.

**Returns** `Line2D`

**See also:**

`axhspan` for example plot and source code

**Notes**

kwargs are passed to `Line2D` and can be used to control the line properties.
Examples

- Draw a thick red hline at ‘y’ = 0 that spans the xrange:

  ```python
  >>> axhline(linewidth=4, color='r')
  ```

- Draw a default hline at ‘y’ = 1 that spans the xrange:

  ```python
  >>> axhline(y=1)
  ```

- Draw a default hline at ‘y’ = .5 that spans the middle half of the xrange:

  ```python
  >>> axhline(y=.5, xmin=0.25, xmax=0.75)
  ```

Valid kwargs are Line2D properties, with the exception of ‘transform’:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>agg_filter</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>alpha</code></td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td><code>animated</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>antialiased</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>axes</code></td>
<td>an Axes instance</td>
</tr>
<tr>
<td><code>clip_box</code></td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td><code>clip_on</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>clip_path</code></td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td><code>color</code> or <code>c</code></td>
<td>any matplotlib color</td>
</tr>
<tr>
<td><code>contains</code></td>
<td>a callable function</td>
</tr>
<tr>
<td><code>dash_capstyle</code></td>
<td>['butt'</td>
</tr>
<tr>
<td><code>dash_joinstyle</code></td>
<td>['miter'</td>
</tr>
<tr>
<td><code>dashes</code></td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td><code>drawstyle</code></td>
<td>['default'</td>
</tr>
<tr>
<td><code>figure</code></td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td><code>fillstyle</code></td>
<td>['full'</td>
</tr>
<tr>
<td><code>gid</code></td>
<td>an id string</td>
</tr>
<tr>
<td><code>label</code></td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td><code>linestyle</code> or <code>ls</code></td>
<td>['solid'</td>
</tr>
<tr>
<td><code>linewidth</code> or <code>lw</code></td>
<td>float value in points</td>
</tr>
<tr>
<td><code>marker</code></td>
<td>A valid marker style</td>
</tr>
<tr>
<td><code>markeredgecolor</code> or <code>mec</code></td>
<td>any matplotlib color</td>
</tr>
<tr>
<td><code>markeredgewidth</code> or <code>mew</code></td>
<td>float value in points</td>
</tr>
<tr>
<td><code>markerfacecolor</code> or <code>mfc</code></td>
<td>any matplotlib color</td>
</tr>
<tr>
<td><code>markerfacecoloralt</code> or <code>mfcalt</code></td>
<td>any matplotlib color</td>
</tr>
<tr>
<td><code>markersize</code> or <code>ms</code></td>
<td>float</td>
</tr>
<tr>
<td><code>markevery</code></td>
<td>[None</td>
</tr>
<tr>
<td><code>path_effects</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>picker</code></td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
</tbody>
</table>
Table 45.3 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>pickradius</strong></td>
<td>float distance in points</td>
</tr>
<tr>
<td><strong>rasterized</strong></td>
<td>[True</td>
</tr>
<tr>
<td><strong>sketch_params</strong></td>
<td>unknown</td>
</tr>
<tr>
<td><strong>snap</strong></td>
<td>unknown</td>
</tr>
<tr>
<td><strong>solid_capstyle</strong></td>
<td>['butt'</td>
</tr>
<tr>
<td><strong>solid_joinstyle</strong></td>
<td>['miter'</td>
</tr>
<tr>
<td><strong>transform</strong></td>
<td>a <em>matplotlib</em>.transforms.Transform instance</td>
</tr>
<tr>
<td><strong>url</strong></td>
<td>a url string</td>
</tr>
<tr>
<td><strong>visible</strong></td>
<td>[True</td>
</tr>
<tr>
<td><strong>xdata</strong></td>
<td>1D array</td>
</tr>
<tr>
<td><strong>ydata</strong></td>
<td>1D array</td>
</tr>
<tr>
<td><strong>zorder</strong></td>
<td>any number</td>
</tr>
</tbody>
</table>

```python
matplotlib.pyplot.axhspan(ymin, ymax, xmin=0, xmax=1, hold=None, **kwargs)
```

Add a horizontal span (rectangle) across the axis.

Draw a horizontal span (rectangle) from `ymin` to `ymax`. With the default values of `xmin` = 0 and `xmax` = 1, this always spans the xrange, regardless of the xlim settings, even if you change them, e.g., with the `set_xlim()` command. That is, the horizontal extent is in axes coords: 0=left, 0.5=middle, 1.0=right but the y location is in data coordinates.

**Parameters**

- **ymin** : float
  Lower limit of the horizontal span in data units.

- **ymax** : float
  Upper limit of the horizontal span in data units.

- **xmin** : float, optional, default: 0
  Lower limit of the vertical span in axes (relative 0-1) units.

- **xmax** : float, optional, default: 1
  Upper limit of the vertical span in axes (relative 0-1) units.

**Returns**

*Polygon*

**Other Parameters**

- **kwargs** : *Polygon* properties.

```
Property            | Description            |
---------------------|------------------------|
agg_filter           | unknown                |
alpha                | float or None          |
animated             | [True | False]           |
antialiased or aa    | [True | False] or None for default |
axes                 | an Axes instance       |
```

Continued on next page
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>capstyle</td>
<td>['butt', 'round', 'projecting']</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True, False]</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform), Patch, None]</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or ec</td>
<td>mpl color spec, None, 'none', or 'auto'</td>
</tr>
<tr>
<td>facecolor or fc</td>
<td>mpl color spec, or None for default, or 'none' for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True, False]</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>['/', '', '</td>
</tr>
<tr>
<td>joinstyle</td>
<td>['miter', 'round', 'bevel']</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with '%s' conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['solid', 'dashed', 'dashdot', 'dotted']</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float or None for default</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None, float, boolean, callable]</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True, False, None]</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True, False]</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

See also:

axvspan Add a vertical span (rectangle) across the axes.
Examples

```python
matplotlib.pyplot.axis(*v, **kwargs)

Convenience method to get or set axis properties.

Calling with no arguments:

```python
>>> axis()
```

returns the current axes limits `[xmin, xmax, ymin, ymax]`:

```python
>>> axis(v)
```

sets the min and max of the x and y axes, with `v = [xmin, xmax, ymin, ymax]`:

```python
>>> axis('off')
```

turns off the axis lines and labels:

```python
>>> axis('equal')
```

changes limits of x or y axis so that equal increments of x and y have the same length; a circle is circular:
achieves the same result by changing the dimensions of the plot box instead of the axis data limits:

```
>>> axis('scaled')
```

changes x and y axis limits such that all data is shown. If all data is already shown, it will move it to the center of the figure without modifying (xmax - xmin) or (ymax - ymin). Note this is slightly different than in MATLAB:

```
>>> axis('tight')
```

is 'scaled' with the axis limits equal to the data limits:

```
>>> axis('image')
```

are deprecated. They restore default behavior; axis limits are automatically scaled to make the data fit comfortably within the plot box.

if `len(*v)==0`, you can pass in xmin, xmax, ymin, ymax as kwargs selectively to alter just those limits without changing the others.

```
>>> axis('square')
```

changes the limit ranges (xmax-xmin) and (ymax-ymin) of the x and y axes to be the same, and have the same scaling, resulting in a square plot.

The xmin, xmax, ymin, ymax tuple is returned

See also:

`xlim()`, `ylim()` For setting the x- and y-limits individually.

Matplotlib.pyplot.axvline(x=0, ymin=0, ymax=1, hold=None, **kwargs)
Add a vertical line across the axes.

**Parameters**

- `x`: scalar, optional, default: 0
  x position in data coordinates of the vertical line.
- `ymin`: scalar, optional, default: 0
  Should be between 0 and 1, 0 being the bottom of the plot, 1 the top of the plot.
- `ymax`: scalar, optional, default: 1
  Should be between 0 and 1, 0 being the bottom of the plot, 1 the top of the plot.
**Returns** *Line2D*

**See also:**

*axhspan* for example plot and source code

**Examples**

- draw a thick red vline at $x = 0$ that spans the yrange:

  ```
  >>> axvline(linewidth=4, color='r')
  ```

- draw a default vline at $x = 1$ that spans the yrange:

  ```
  >>> axvline(x=1)
  ```

- draw a default vline at $x = .5$ that spans the middle half of the yrange:

  ```
  >>> axvline(x=.5, ymin=0.25, ymax=0.75)
  ```

Valid kwargs are *Line2D* properties, with the exception of ‘transform’:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>agg_filter</strong></td>
<td>unknown</td>
</tr>
<tr>
<td><strong>alpha</strong></td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td><strong>animated</strong></td>
<td>[True</td>
</tr>
<tr>
<td><strong>antialiased</strong>  or aa</td>
<td>[True</td>
</tr>
<tr>
<td><strong>axes</strong></td>
<td>an <em>Axes</em> instance</td>
</tr>
<tr>
<td><strong>clip_box</strong></td>
<td>a <em>matplotlib.transforms.Bbox</em> instance</td>
</tr>
<tr>
<td><strong>clip_on</strong></td>
<td>[True</td>
</tr>
<tr>
<td><strong>clip_path</strong></td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td><strong>color</strong> or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td><strong>contains</strong></td>
<td>a callable function</td>
</tr>
<tr>
<td><strong>dash_capstyle</strong></td>
<td>['butt'</td>
</tr>
<tr>
<td><strong>dash_joinstyle</strong></td>
<td>['miter'</td>
</tr>
<tr>
<td><strong>dashes</strong></td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td><strong>drawstyle</strong></td>
<td>['default'</td>
</tr>
<tr>
<td><strong>figure</strong></td>
<td>a <em>matplotlib.figure.Figure</em> instance</td>
</tr>
<tr>
<td><strong>fillstyle</strong></td>
<td>['full'</td>
</tr>
<tr>
<td><strong>gid</strong></td>
<td>an id string</td>
</tr>
<tr>
<td><strong>label</strong></td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td><strong>linestyle</strong> or ls</td>
<td>['solid'</td>
</tr>
<tr>
<td><strong>linewidth</strong> or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td><strong>marker</strong></td>
<td>A valid marker style</td>
</tr>
<tr>
<td><strong>markeredgecolor</strong> or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td><strong>markeredgedwidth</strong> or mew</td>
<td>float value in points</td>
</tr>
</tbody>
</table>
Table 45.5 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>[None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

```python
matplotlib.pyplot.axvspan(xmin, xmax, ymin=0, ymax=1, hold=None, **kwargs)
```

Add a vertical span (rectangle) across the axes.

Draw a vertical span (rectangle) from xmin to xmax. With the default values of ymin = 0 and ymax = 1. This always spans the yrange, regardless of the ylim settings, even if you change them, e.g., with the set_ylim() command. That is, the vertical extent is in axes coords: 0=bottom, 0.5=middle, 1.0=top but the y location is in data coordinates.

**Parameters xmin** : scalar

   Number indicating the first X-axis coordinate of the vertical span rectangle in data units.

**xmax** : scalar

   Number indicating the second X-axis coordinate of the vertical span rectangle in data units.

**ymin** : scalar, optional

   Number indicating the first Y-axis coordinate of the vertical span rectangle in relative Y-axis units (0-1). Default to 0.

**ymax** : scalar, optional

   Number indicating the second Y-axis coordinate of the vertical span rectangle in relative Y-axis units (0-1). Default to 1.

**Returns rectangle** : matplotlib.patches.Polygon
Vertical span (rectangle) from (xmin, ymin) to (xmax, ymax).

**Other Parameters**

Optional parameters are properties of the class `matplotlib.patches.Polygon`.

**See also:**

`axhspan`

**Examples**

Draw a vertical, green, translucent rectangle from $x = 1.25$ to $x = 1.55$ that spans the yrange of the axes.

```python
>>> axvspan(1.25, 1.55, facecolor='g', alpha=0.5)
```

`matplotlib.pyplot.bar` *(left, height, width=0.8, bottom=None, hold=None, data=None, **kwargs)*

Make a bar plot.

Make a bar plot with rectangles bounded by:

*left, left + width, bottom, bottom + height* (left, right, bottom and top edges)

**Parameters**

`left` : sequence of scalars

the x coordinates of the left sides of the bars

`height` : sequence of scalars

the heights of the bars

`width` : scalar or array-like, optional

the width(s) of the bars default: 0.8

`bottom` : scalar or array-like, optional

the y coordinate(s) of the bars default: None

`color` : scalar or array-like, optional

the colors of the bar faces

`edgecolor` : scalar or array-like, optional

the colors of the bar edges

`linewidth` : scalar or array-like, optional

width of bar edge(s). If None, use default linewidth; If 0, don’t draw edges. default: None

`tick_label` : string or array-like, optional

the tick labels of the bars default: None
xerr : scalar or array-like, optional
    if not None, will be used to generate errorbar(s) on the bar chart default: None

yerr : scalar or array-like, optional
    if not None, will be used to generate errorbar(s) on the bar chart default: None

ecolor : scalar or array-like, optional
    specifies the color of errorbar(s) default: None

capsize : scalar, optional
    determines the length in points of the error bar caps default: None, which will take the value from the errorbar.capsize rcParam.

error_kw : dict, optional
    dictionary of kwargs to be passed to errorbar method. ecolor and capsize may be specified here rather than as independent kwargs.

align : {'center', 'edge'}, optional
    If ‘edge’, aligns bars by their left edges (for vertical bars) and by their bottom edges (for horizontal bars). If ‘center’, interpret the left argument as the coordinates of the centers of the bars. To align on the align bars on the right edge pass a negative width.

orientation : {'vertical', 'horizontal'}, optional
    The orientation of the bars.

log : boolean, optional
    If true, sets the axis to be log scale. default: False

Returns bars : matplotlib.container.BarContainer
    Container with all of the bars + errorbars

See also:

barh Plot a horizontal bar plot.

Notes

The optional arguments color, edgecolor, linewidth, xerr, and yerr can be either scalars or sequences of length equal to the number of bars. This enables you to use bar as the basis for stacked bar charts, or candlestick plots. Detail: xerr and yerr are passed directly to errorbar(), so they can also have shape 2xN for independent specification of lower and upper errors.

Other optional kwargs:
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or ec</td>
<td>mpl color spec, None, ‘none’, or ‘auto’</td>
</tr>
<tr>
<td>facecolor or fc</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[‘/’</td>
</tr>
<tr>
<td>joinstyle</td>
<td>[‘miter’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>[‘solid’</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float or None for default</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

**Examples**

**Example:** A stacked bar chart.
Note: In addition to the above described arguments, this function can take a data keyword argument. If such a data argument is given, the following arguments are replaced by data[arg]:


matplotlib.pyplot.barbs(*args, **kw)
Plot a 2-D field of barbs.

Call signatures:

```python
barb(U, V, **kw)
barb(U, V, C, **kw)
barb(X, Y, U, V, **kw)
barb(X, Y, U, V, C, **kw)
```

Arguments:

- $X, Y$: The x and y coordinates of the barb locations (default is head of barb; see pivot kwarg)
- $U, V$: Give the x and y components of the barb shaft
- $C$: An optional array used to map colors to the barbs
All arguments may be 1-D or 2-D arrays or sequences. If X and Y are absent, they will be generated as a uniform grid. If U and V are 2-D arrays but X and Y are 1-D, and if \texttt{len(X)} and \texttt{len(Y)} match the column and row dimensions of \texttt{U}, then \texttt{X} and \texttt{Y} will be expanded with \texttt{numpy.meshgrid()}. 

\texttt{U}, \texttt{V}, \texttt{C} may be masked arrays, but masked \texttt{X}, \texttt{Y} are not supported at present.

Keyword arguments:

- **length**: Length of the barb in points; the other parts of the barb are scaled against this. Default is 9
- **pivot**: [‘tip’ | ‘middle’] The part of the arrow that is at the grid point; the arrow rotates about this point, hence the name pivot. Default is ‘tip’
- **barbcolor**: [color | color sequence] Specifies the color all parts of the barb except any flags. This parameter is analogous to the \texttt{edgecolor} parameter for polygons, which can be used instead. However this parameter will override facecolor.
- **flagcolor**: [color | color sequence] Specifies the color of any flags on the barb. This parameter is analogous to the \texttt{facecolor} parameter for polygons, which can be used instead. However this parameter will override facecolor. If this is not set (and \texttt{C} has not either) then \texttt{flagcolor} will be set to match \texttt{barbcolor} so that the barb has a uniform color. If \texttt{C} has been set, \texttt{flagcolor} has no effect.
- **sizes**: A dictionary of coefficients specifying the ratio of a given feature to the length of the barb. Only those values one wishes to override need to be included. These features include:
  - ‘spacing’ - space between features (flags, full/half barbs)
  - ‘height’ - height (distance from shaft to top) of a flag or full barb
  - ‘width’ - width of a flag, twice the width of a full barb
  - ‘emptybarb’ - radius of the circle used for low magnitudes
- **fill_empty**: A flag on whether the empty bars (circles) that are drawn should be filled with the flag color. If they are not filled, they will be drawn such that no color is applied to the center. Default is False
- **rounding**: A flag to indicate whether the vector magnitude should be rounded when allocating barb components. If True, the magnitude is rounded to the nearest multiple of the half-barb increment. If False, the magnitude is simply truncated to the next lowest multiple. Default is True
- **barb_increments**: A dictionary of increments specifying values to associate with different parts of the barb. Only those values one wishes to override need to be included.
  - ‘half’ - half barbs (Default is 5)
  - ‘full’ - full barbs (Default is 10)
  - ‘flag’ - flags (default is 50)
- **flip_barb**: Either a single boolean flag or an array of booleans. Single boolean indicates whether the lines and flags should point opposite to normal for all barbs. An array (which should be the same size as the other data arrays) indicates whether to flip for
each individual barb. Normal behavior is for the barbs and lines to point right (comes from wind barbs having these features point towards low pressure in the Northern Hemisphere.) Default is False

Barbs are traditionally used in meteorology as a way to plot the speed and direction of wind observations, but can technically be used to plot any two dimensional vector quantity. As opposed to arrows, which give vector magnitude by the length of the arrow, the barbs give more quantitative information about the vector magnitude by putting slanted lines or a triangle for various increments in magnitude, as show schematically below:

```
:  /\  \\
:: /\ \\
:: / \ \\
:: /  \\
-----------------------------
```

The largest increment is given by a triangle (or “flag”). After those come full lines (barbs). The smallest increment is a half line. There is only, of course, ever at most 1 half line. If the magnitude is small and only needs a single half-line and no full lines or triangles, the half-line is offset from the end of the barb so that it can be easily distinguished from barbs with a single full line. The magnitude for the barb shown above would nominally be 65, using the standard increments of 50, 10, and 5.

lineweights and edgecolors can be used to customize the barb. Additional `PolyCollection` keyword arguments:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>agg_filter</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>alpha</code></td>
<td>float or None</td>
</tr>
<tr>
<td><code>animated</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>antialiased</code> or <code>antialiaseds</code></td>
<td>Boolean or sequence of booleans</td>
</tr>
<tr>
<td><code>array</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>axes</code></td>
<td>an <code>Axes</code> instance</td>
</tr>
<tr>
<td><code>clim</code></td>
<td>a length 2 sequence of floats</td>
</tr>
<tr>
<td><code>clip_box</code></td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td><code>clip_on</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>clip_path</code></td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td><code>cmap</code></td>
<td>a colormap or registered colormap name</td>
</tr>
<tr>
<td><code>color</code></td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td><code>contains</code></td>
<td>a callable function</td>
</tr>
<tr>
<td><code>edgecolor</code> or <code>edgecolors</code></td>
<td>matplotlib color spec or sequence of specs</td>
</tr>
<tr>
<td><code>facecolor</code> or <code>facecolors</code></td>
<td>matplotlib color spec or sequence of specs</td>
</tr>
<tr>
<td><code>figure</code></td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td><code>gid</code></td>
<td>an id string</td>
</tr>
<tr>
<td><code>hatch</code></td>
<td>[ ' '</td>
</tr>
<tr>
<td><code>label</code></td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td><code>linestyle</code> or <code>dashes</code> or <code>linestyles</code></td>
<td>['solid'</td>
</tr>
<tr>
<td><code>linewidth</code> or <code>linewidths</code></td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td><code>norm</code></td>
<td>unknown</td>
</tr>
</tbody>
</table>
Table 45.7 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>offset_position</td>
<td>unknown</td>
</tr>
<tr>
<td>offsets</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>pickradius</td>
<td>unknown</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>urls</td>
<td>unknown</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

Example:
Make a horizontal bar plot with rectangles bounded by:

- `left, left + width, bottom, bottom + height` (left, right, bottom and top edges)
- `bottom, width, height, and left` can be either scalars or sequences

**Parameters**
- `bottom` : scalar or array-like
  - the y coordinate(s) of the bars
- `width` : scalar or array-like
  - the width(s) of the bars
- `height` : sequence of scalars, optional, default: 0.8
  - the heights of the bars
- `left` : sequence of scalars

**Note:** In addition to the above described arguments, this function can take a `data` keyword argument. If such a `data` argument is given, the following arguments are replaced by `data[<arg>]`:

- All positional and all keyword arguments.
the x coordinates of the left sides of the bars

**Returns** `matplotlib.patches.Rectangle` instances.

**Other Parameters**
- **color**: scalar or array-like, optional
  the colors of the bars
- **edgecolor**: scalar or array-like, optional
  the colors of the bar edges
- **linewidth**: scalar or array-like, optional, default: None
  width of bar edge(s). If None, use default linewidth; If 0, don’t draw edges.
- **tick_label**: string or array-like, optional, default: None
  the tick labels of the bars
- **xerr**: scalar or array-like, optional, default: None
  if not None, will be used to generate errorbar(s) on the bar chart
- **yerr**: scalar or array-like, optional, default: None
  if not None, will be used to generate errorbar(s) on the bar chart
- **ecolor**: scalar or array-like, optional, default: None
  specifies the color of errorbar(s)
- **capsize**: scalar, optional
  determines the length in points of the error bar caps default: None, which will take the value from the `errorbar.capsize rcParam`.
- **error_kw**: dictionary of kwargs to be passed to errorbar method. ecolor and capsize may be specified here rather than as independent kwargs.
- **align**: {'center', 'edge'}, optional
  If ‘edge’, aligns bars by their left edges (for vertical bars) and by their bottom edges (for horizontal bars). If ‘center’, interpret the bottom argument as the coordinates of the centers of the bars. To align on the align bars on the top edge pass a negative ‘height’.
- **log**: boolean, optional, default: False
  If true, sets the axis to be log scale

**See also**

- **bar**: Plot a vertical bar plot.
Notes

The optional arguments color, edgecolor, linewidth, xerr, and yerr can be either scalars or sequences of length equal to the number of bars. This enables you to use bar as the basis for stacked bar charts, or candlestick plots. Detail: xerr and yerr are passed directly to errorbar(), so they can also have shape 2xN for independent specification of lower and upper errors.

Other optional kwargs:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor</td>
<td>mpl color spec, None, 'none', or 'auto'</td>
</tr>
<tr>
<td>facecolor</td>
<td>mpl color spec, or None for default, or 'none' for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>['/'</td>
</tr>
<tr>
<td>joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle</td>
<td>['solid'</td>
</tr>
<tr>
<td>linewidth</td>
<td>float or None for default</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

matplotlib.pyplot.bone()

set the default colormap to bone and apply to current image if any. See help(colormaps) for more information.
matplotlib.pyplot.box(on=None)

Turn the axes box on or off. on may be a boolean or a string, ‘on’ or ‘off’. If on is None, toggle state.

matplotlib.pyplot.boxplot(x, notch=None, sym=None, vert=None, whis=None, positions=None, widths=None, patch_artist=None, bootstrap=None, usermedians=None, conf_intervals=None, medianline=None, showmeans=None, showcaps=None, showbox=None, showfliers=None, boxprops=None, labels=None, flierprops=None, medianprops=None, meanprops=None, capprops=None, whiskerprops=None, manage_xticks=True, autorange=False, zorder=None, hold=None, data=None)

Make a box and whisker plot.

Make a box and whisker plot for each column of x or each vector in sequence x. The box extends from the lower to upper quartile values of the data, with a line at the median. The whiskers extend from the box to show the range of the data. Flier points are those past the end of the whiskers.

**Parameters x :** Array or a sequence of vectors.

The input data.

**notch :** bool, optional (False)

If True, will produce a notched box plot. Otherwise, a rectangular boxplot is produced. The notches represent the confidence interval (CI) around the median. See the entry for the bootstrap parameter for information regarding how the locations of the notches are computed.

**Note:** In cases where the values of the CI are less than the lower quartile or greater than the upper quartile, the notches will extend beyond the box, giving it a distinctive “flipped” appearance. This is expected behavior and consistent with other statistical visualization packages.

**sym :** str, optional

The default symbol for flier points. Enter an empty string ('') if you don’t want to show fliers. If None, then the fliers default to ‘b+’ If you want more control use the flierprops kwarg.

**vert :** bool, optional (True)

If True (default), makes the boxes vertical. If False, everything is drawn horizontally.

**whis :** float, sequence, or string (default = 1.5)

As a float, determines the reach of the whiskers to the beyond the first and third quartiles. In other words, where IQR is the interquartile range (Q3–Q1), the upper whisker will extend to last datum less than Q3 + \text{whis}\times\text{IQR}). Similarly, the lower whisker will extend to the first datum greater than Q1 -
whis*IQR. Beyond the whiskers, data are considered outliers and are plotted as individual points. Set this to an unreasonably high value to force the whiskers to show the min and max values. Alternatively, set this to an ascending sequence of percentile (e.g., [5, 95]) to set the whiskers at specific percentiles of the data. Finally, whis can be the string 'range' to force the whiskers to the min and max of the data.

**bootstrap** : int, optional

Specifies whether to bootstrap the confidence intervals around the median for notched boxplots. If bootstrap is None, no bootstrapping is performed, and notches are calculated using a Gaussian-based asymptotic approximation (see McGill, R., Tukey, J.W., and Larsen, W.A., 1978, and Kendall and Stuart, 1967). Otherwise, bootstrap specifies the number of times to bootstrap the median to determine its 95% confidence intervals. Values between 1000 and 10000 are recommended.

**usermedians** : array-like, optional

An array or sequence whose first dimension (or length) is compatible with x. This overrides the medians computed by matplotlib for each element of usermedians that is not None. When an element of usermedians is None, the median will be computed by matplotlib as normal.

**conf_intervals** : array-like, optional

Array or sequence whose first dimension (or length) is compatible with x and whose second dimension is 2. When an element of conf_intervals is not None, the notch locations computed by matplotlib are overridden (provided notch is True). When an element of conf_intervals is None, the notches are computed by the method specified by the other kwargs (e.g., bootstrap).

**positions** : array-like, optional

Sets the positions of the boxes. The ticks and limits are automatically set to match the positions. Defaults to range(1, N+1) where N is the number of boxes to be drawn.

**widths** : scalar or array-like

Sets the width of each box either with a scalar or a sequence. The default is 0.5, or 0.15*(distance between extreme positions), if that is smaller.

**patch_artist** : bool, optional (False)

If False produces boxes with the Line2D artist. Otherwise, boxes and drawn with Patch artists.

**labels** : sequence, optional

Labels for each dataset. Length must be compatible with dimensions of x.

**manage_xticks** : bool, optional (True)
If the function should adjust the xlim and xtick locations.

**autorange**: bool, optional (False)

When **True** and the data are distributed such that the 25th and 75th percentiles are equal, **whis** is set to **'range'** such that the whisker ends are at the minimum and maximum of the data.

**meanline**: bool, optional (False)

If **True** (and **showmeans** is **True**), will try to render the mean as a line spanning the full width of the box according to **meanprops** (see below). Not recommended if **shownotches** is also **True**. Otherwise, means will be shown as points.

**zorder**: scalar, optional (None)

Sets the zorder of the boxplot.

Returns **result**: dict

A dictionary mapping each component of the boxplot to a list of the **matplotlib.pyplot.Line2D** instances created. That dictionary has the following keys (assuming vertical boxplots):

- **boxes**: the main body of the boxplot showing the quartiles and the median’s confidence intervals if enabled.
- **medians**: horizontal lines at the median of each box.
- **whiskers**: the vertical lines extending to the most extreme, non-outlier data points.
- **caps**: the horizontal lines at the ends of the whiskers.
- **fliers**: points representing data that extend beyond the whiskers (fliers).
- **means**: points or lines representing the means.

Other Parameters **showcaps**: bool, optional (True)

Show the caps on the ends of whiskers.

**showbox**: bool, optional (True)

Show the central box.

**showfliers**: bool, optional (True)

Show the outliers beyond the caps.

**showmeans**: bool, optional (False)

Show the arithmetic means.

**capprops**: dict, optional (None)

Specifies the style of the caps.

**boxprops**: dict, optional (None)
Specifies the style of the box.

**whiskerprops**: dict, optional (None)

Specifies the style of the whiskers.

**flierprops**: dict, optional (None)

Specifies the style of the fliers.

**medianprops**: dict, optional (None)

Specifies the style of the median.

**meanprops**: dict, optional (None)

Specifies the style of the mean.
Examples

```
Default

showmeans=True

showmeans=True, meanline=True

Tufte Style
(showbox=False, showcaps=False)

notch=True, bootstrap=10000

showfliers=False
```
I never said they'd be pretty

<table>
<thead>
<tr>
<th>Custom boxprops</th>
<th>Custom medianprops and flierprops</th>
<th>whis=&quot;range&quot;</th>
</tr>
</thead>
</table>

| Custom mean as point | Custom mean as line | whis=[15, 85] #percentiles |

Note: In addition to the above described arguments, this function can take a data keyword argument. If such a data argument is given, the following arguments are replaced by data[<arg>]:

- All positional and all keyword arguments.

```python
matplotlib.pyplot.broken_barh(xranges, yrange, hold=None, data=None, **kwargs)
```

Plot horizontal bars.

A collection of horizontal bars spanning yrange with a sequence of xranges.

Required arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xranges</td>
<td>sequence of (xmin, xwidth)</td>
</tr>
<tr>
<td>yrange</td>
<td>sequence of (ymin, ywidth)</td>
</tr>
</tbody>
</table>
kwargs are `matplotlib.collections.BrokenBarHCollection` properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>agg_filter</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>alpha</code></td>
<td>float or None</td>
</tr>
<tr>
<td><code>animated</code></td>
<td>`[True</td>
</tr>
<tr>
<td><code>antialiased</code></td>
<td>Boolean or sequence of booleans</td>
</tr>
<tr>
<td><code>array</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>axes</code></td>
<td>an <code>Axes</code> instance</td>
</tr>
<tr>
<td><code>clim</code></td>
<td>a length 2 sequence of floats</td>
</tr>
<tr>
<td><code>clip_box</code></td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td><code>clip_on</code></td>
<td>`[True</td>
</tr>
<tr>
<td><code>clip_path</code></td>
<td>`(Path, Transform)</td>
</tr>
<tr>
<td><code>cmap</code></td>
<td>a colormap or registered colormap name</td>
</tr>
<tr>
<td><code>color</code></td>
<td><code>matplotlib</code> color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td><code>contains</code></td>
<td>a callable function</td>
</tr>
<tr>
<td><code>edgecolor</code> or <code>edgecolors</code></td>
<td><code>matplotlib</code> color spec or sequence of specs</td>
</tr>
<tr>
<td><code>facecolor</code> or <code>facecolors</code></td>
<td><code>matplotlib</code> color spec or sequence of specs</td>
</tr>
<tr>
<td><code>figure</code></td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td><code>gid</code></td>
<td>an id string</td>
</tr>
<tr>
<td><code>hatch</code></td>
<td>`['/', '', '</td>
</tr>
<tr>
<td><code>label</code></td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td><code>linestyle</code> or <code>dashes</code> or <code>linestyles</code></td>
<td><code>['solid', 'dashed', 'dashdot', 'dotted', (offset, on-off-dash-seq), '-', '--', '-.' ]</code></td>
</tr>
<tr>
<td><code>linewidth</code> or <code>linewidths</code> or <code>lw</code></td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td><code>norm</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>offset_position</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>offsets</code></td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td><code>path_effects</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>picker</code></td>
<td>`[None</td>
</tr>
<tr>
<td><code>pickradius</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>rasterized</code></td>
<td>`[True</td>
</tr>
<tr>
<td><code>sketch_params</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>snap</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>transform</code></td>
<td><code>Transform</code> instance</td>
</tr>
<tr>
<td><code>url</code></td>
<td>a url string</td>
</tr>
<tr>
<td><code>urls</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>visible</code></td>
<td>`[True</td>
</tr>
<tr>
<td><code>zorder</code></td>
<td>any number</td>
</tr>
</tbody>
</table>

these can either be a single argument, i.e.:

```
facecolors = 'black'
```

or a sequence of arguments for the various bars, i.e.:
facecolors = ('black', 'red', 'green')

Example:

Note: In addition to the above described arguments, this function can take a data keyword argument. If such a data argument is given, the following arguments are replaced by data[<arg>]:

- All positional and all keyword arguments.

matplotlib.pyplot.cla()
Clear the current axes.

matplotlib.pyplot.clabel(CS, *args, **kwargs)
Label a contour plot.

Call signature:

clabel(cs, **kwargs)

Adds labels to line contours in cs, where cs is a ContourSet object returned by contour.

clabel(cs, v, **kwargs)

only labels contours listed in v.
Optional keyword arguments:

**fontsize**: size in points or relative size e.g., ‘smaller’, ‘x-large’

**colors**:
- if *None*, the color of each label matches the color of the corresponding contour
- if one string color, e.g., *colors* = ‘r’ or *colors* = ‘red’, all labels will be plotted in this color
- if a tuple of matplotlib color args (string, float, rgb, etc), different labels will be plotted in different colors in the order specified

**inline**: controls whether the underlying contour is removed or not. Default is *True*.

**inline_spacing**: space in pixels to leave on each side of label when placing inline. Defaults to 5. This spacing will be exact for labels at locations where the contour is straight, less so for labels on curved contours.

**fmt**: a format string for the label. Default is ‘%1.3f’ Alternatively, this can be a dictionary matching contour levels with arbitrary strings to use for each contour level (i.e., fmt[level]=string), or it can be any callable, such as a *Formatter* instance, that returns a string when called with a numeric contour level.

**manual**: if *True*, contour labels will be placed manually using mouse clicks. Click the first button near a contour to add a label, click the second button (or potentially both mouse buttons at once) to finish adding labels. The third button can be used to remove the last label added, but only if labels are not inline. Alternatively, the keyboard can be used to select label locations (enter to end label placement, delete or backspace act like the third mouse button, and any other key will select a label location).

*manual* can be an iterable object of x,y tuples. Contour labels will be created as if mouse is clicked at each x,y positions.

**rightside_up**: if *True* (default), label rotations will always be plus or minus 90 degrees from level.

**use_clabeltext**: if *True* (default is False), ClabelText class (instead of matplotlib.Text) is used to create labels. ClabelText recalculates rotation angles of texts during the drawing time, therefore this can be used if aspect of the axes changes.
Simplest default with labels
Single color - negative contours dashed
Single color - negative contours solid
matplotlib.pyplot.clf()
Clear the current figure.

matplotlib.pyplot.clim(vmin=None, vmax=None)
Set the color limits of the current image.

To apply clim to all axes images do:

clim(0, 0.5)

If either vmin or vmax is None, the image min/max respectively will be used for color scaling.

If you want to set the clim of multiple images, use, for example:

```
for im in gca().get_images():
    im.set_clim(0, 0.05)
```

matplotlib.pyplot.close(*args)
Close a figure window.

close() by itself closes the current figure

close(h) where h is a Figure instance, closes that figure

close(num) closes figure number num

close(name) where name is a string, closes figure with that label
close('all') closes all the figure windows

matplotlib.pyplot.cohere(x, y, NFFT=256, Fs=2, Fc=0, detrend=<function detrend_none>, window=<function window_hanning>, noverlap=0, pad_to=None, sides='default', scale_by_freq=None, hold=None, data=None, **kwargs)

Plot the coherence between $x$ and $y$.

Plot the coherence between $x$ and $y$. Coherence is the normalized cross spectral density:

$$C_{xy} = \frac{|P_{xy}|^2}{P_{xx}P_{yy}}$$  \hspace{1cm} (45.1)

**Parameters**

- **Fs**: scalar
  
  The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.

- **window**: callable or ndarray
  
  A function or a vector of length $NFFT$. To create window vectors see `window_hanning()`, `window_none()`, `numpy.blackman()`, `numpy.hanning()`, `numpy.bartlett()`, `scipy.signal()`, and `scipy.signal.get_window()`, etc. The default is `window_hanning()`. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

- **sides**: [ ‘default’ | ‘onesided’ | ‘twosided’ ]
  
  Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. ‘onesided’ forces the return of a one-sided spectrum, while ‘twosided’ forces two-sided.

- **pad_to**: integer
  
  The number of points to which the data segment is padded when performing the FFT. This can be different from $NFFT$, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the $n$ parameter in the call to `fft()`. The default is None, which sets pad_to equal to $NFFT$.

- **NFFT**: integer
  
  The number of data points used in each block for the FFT. This can be different from $NFFT$, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the $n$ parameter in the call to `fft()`. The default is 256. This should NOT be used to get zero padding, or the scaling of the result will be incorrect. Use pad_to for this instead.

- **detrend**: {‘default’, ‘constant’, ‘mean’, ‘linear’, ‘none’} or callable
  
  The function applied to each segment before `fft-ing`, designed to remove the mean or linear trend. Unlike in MATLAB, where the detrend parameter is a vector, in matplotlib it is a function. The `pylab` module defines
detrend_none(), detrend_mean(), and detrend_linear(), but you can use a custom function as well. You can also use a string to choose one of the functions. 'default', 'constant', and 'mean' call detrend_mean(). 'linear' calls detrend_linear(). 'none' calls detrend_none().

scale_by_freq : boolean, optional

Specifies whether the resulting density values should be scaled by the scaling frequency, which gives density in units of Hz^-1. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

noverlap : integer

The number of points of overlap between blocks. The default value is 0 (no overlap).

Fc : integer

The center frequency of x (defaults to 0), which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

**kwargs :

Keyword arguments control the Line2D properties of the coherence plot:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>[‘butt’</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>[‘miter’</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>drawstyle</td>
<td>[‘default’</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.pyplot.figure.Figure instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>[‘full’</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>[‘solid’</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>marker</td>
<td>A valid marker style</td>
</tr>
</tbody>
</table>
Table 45.10 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>markeredgecolor</code> or <code>mec</code></td>
<td>any matplotlib color</td>
</tr>
<tr>
<td><code>markeredgewidth</code> or <code>mew</code></td>
<td>float value in points</td>
</tr>
<tr>
<td><code>markerfacecolor</code> or <code>mfc</code></td>
<td>any matplotlib color</td>
</tr>
<tr>
<td><code>markerfacecoloralt</code> or <code>mfcalt</code></td>
<td>any matplotlib color</td>
</tr>
<tr>
<td><code>markersize</code> or <code>ms</code></td>
<td>float</td>
</tr>
<tr>
<td><code>markevery</code></td>
<td>[None</td>
</tr>
<tr>
<td><code>path_effects</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>picker</code></td>
<td>float distance in points or callable pick function <code>fn(artist, event)</code></td>
</tr>
<tr>
<td><code>pickradius</code></td>
<td>float distance in points</td>
</tr>
<tr>
<td><code>rasterized</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>sketch_params</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>solid_capstyle</code></td>
<td>['butt'</td>
</tr>
<tr>
<td><code>solid_joinstyle</code></td>
<td>['miter'</td>
</tr>
<tr>
<td><code>transform</code></td>
<td>a <code>matplotlib.transforms.Transform</code> instance</td>
</tr>
<tr>
<td><code>url</code></td>
<td>a url string</td>
</tr>
<tr>
<td><code>visible</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>xdata</code></td>
<td>1D array</td>
</tr>
<tr>
<td><code>ydata</code></td>
<td>1D array</td>
</tr>
<tr>
<td><code>zorder</code></td>
<td>any number</td>
</tr>
</tbody>
</table>

**Returns** The return value is a tuple \((C_{xy}, f)\), where \(f\) are the frequencies of the coherence vector.

**kwargs are applied to the lines.

**References**

Note: In addition to the above described arguments, this function can take a data keyword argument. If such a data argument is given, the following arguments are replaced by data[<arg>]:

• All arguments with the following names: ‘x’, ‘y’.

matplotlib.pyplot.colorbar(mappable=None, cax=None, ax=None, **kw)

Add a colorbar to a plot.

Function signatures for the pyplot interface; all but the first are also method signatures for the colorbar() method:

- colorbar(**kwargs)
- colorbar(mappable, **kwargs)
- colorbar(mappable, cax=cax, **kwargs)
- colorbar(mappable, ax=ax, **kwargs)

arguments:

mappable the Image, ContourSet, etc. to which the colorbar applies; this argument is mandatory for the colorbar() method but optional for the colorbar() function, which sets the default to the current image.
keyword arguments:

- **cax**  
  None | axes object into which the colorbar will be drawn

- **ax**  
  None | parent axes object(s) from which space for a new colorbar axes will be stolen. If a list of axes is given they will all be resized to make room for the colorbar axes.

- **use_gridspec**  
  False | If cax is None, a new cax is created as an instance of Axes. If ax is an instance of Subplot and use_gridspec is True, cax is created as an instance of Subplot using the grid_spec module.

Additional keyword arguments are of two kinds:

**axes properties:**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>orientation</td>
<td>vertical or horizontal</td>
</tr>
<tr>
<td>fraction</td>
<td>0.15; fraction of original axes to use for colorbar</td>
</tr>
<tr>
<td>pad</td>
<td>0.05 if vertical, 0.15 if horizontal; fraction of original axes between colorbar and new image axes</td>
</tr>
<tr>
<td>shrink</td>
<td>1.0; fraction by which to shrink the colorbar</td>
</tr>
<tr>
<td>aspect</td>
<td>20; ratio of long to short dimensions</td>
</tr>
<tr>
<td>anchor</td>
<td>(0.0, 0.5) if vertical; (0.5, 1.0) if horizontal; the anchor point of the colorbar axes</td>
</tr>
<tr>
<td>pan-chor</td>
<td>(1.0, 0.5) if vertical; (0.5, 0.0) if horizontal; the anchor point of the colorbar parent axes. If False, the parent axes’ anchor will be unchanged</td>
</tr>
</tbody>
</table>

**colorbar properties:**
Matplotlib, Release 2.0.2

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>extend</code></td>
<td>[ ‘neither’</td>
</tr>
<tr>
<td><code>extendfrac</code></td>
<td>[ None</td>
</tr>
<tr>
<td><code>extendrect</code></td>
<td>[ False</td>
</tr>
<tr>
<td><code>spacing</code></td>
<td>[ ‘uniform’</td>
</tr>
<tr>
<td><code>ticks</code></td>
<td>[ None</td>
</tr>
<tr>
<td><code>format</code></td>
<td>[ None</td>
</tr>
<tr>
<td><code>drawedges</code></td>
<td>[ False</td>
</tr>
</tbody>
</table>

The following will probably be useful only in the context of indexed colors (that is, when the mappable has norm=NoNorm()), or other unusual circumstances.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>boundaries</code></td>
<td>None or a sequence</td>
</tr>
<tr>
<td><code>values</code></td>
<td>None or a sequence which must be of length 1 less than the sequence of boundaries. For each region delimited by adjacent entries in boundaries, the color mapped to the corresponding value in values will be used.</td>
</tr>
</tbody>
</table>

If mappable is a ContourSet, its extend kwarg is included automatically.

Note that the shrink kwarg provides a simple way to keep a vertical colorbar, for example, from being taller than the axes of the mappable to which the colorbar is attached; but it is a manual method requiring some trial and error. If the colorbar is too tall (or a horizontal colorbar is too wide) use a smaller value of shrink.

For more precise control, you can manually specify the positions of the axes objects in which the
mappable and the colorbar are drawn. In this case, do not use any of the axes properties kwargs.

It is known that some vector graphics viewer (svg and pdf) renders white gaps between segments of the colorbar. This is due to bugs in the viewers not matplotlib. As a workaround the colorbar can be rendered with overlapping segments:

```python
cbar = colorbar()
cbar.solids.set_edgecolor("face")
draw()
```

However this has negative consequences in other circumstances. Particularly with semi transparent images (alpha < 1) and colorbar extensions and is not enabled by default see (issue #1188).

**returns:** Colorbar instance; see also its base class, ColorbarBase. Call the set_label() method to label the colorbar.

```python
matplotlib.pyplot.colors()
```

This is a do-nothing function to provide you with help on how matplotlib handles colors.

Commands which take color arguments can use several formats to specify the colors. For the basic built-in colors, you can use a single letter

<table>
<thead>
<tr>
<th>Alias</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>'b'</td>
<td>blue</td>
</tr>
<tr>
<td>'g'</td>
<td>green</td>
</tr>
<tr>
<td>'r'</td>
<td>red</td>
</tr>
<tr>
<td>'c'</td>
<td>cyan</td>
</tr>
<tr>
<td>'m'</td>
<td>magenta</td>
</tr>
<tr>
<td>'y'</td>
<td>yellow</td>
</tr>
<tr>
<td>'k'</td>
<td>black</td>
</tr>
<tr>
<td>'w'</td>
<td>white</td>
</tr>
</tbody>
</table>

For a greater range of colors, you have two options. You can specify the color using an html hex string, as in:

```python
color = '#eefeff'
```

or you can pass an R,G,B tuple, where each of R,G,B are in the range [0,1].

You can also use any legal html name for a color, for example:

```python
color = 'red'
color = 'burlywood'
color = 'chartreuse'
```

The example below creates a subplot with a dark slate gray background:

```python
subplot(111, facecolor=(0.1843, 0.3098, 0.3098))
```

Here is an example that creates a pale turquoise title:

```python
title('Is this the best color?', color='#afeeee')
```
**matplotlib.pyplot.connect**(*s, func*)

Connect event with string *s* to *func*. The signature of *func* is:

```python
def func(event)
```

where *event* is a *matplotlib.backend_bases.Event*. The following events are recognized:

- `'button_press_event'`
- `'button_release_event'`
- `'draw_event'`
- `'key_press_event'`
- `'key_release_event'`
- `'motion_notify_event'`
- `'pick_event'`
- `'resize_event'`
- `'scroll_event'`
- `'figure_enter_event'`
- `'figure_leave_event'`
- `'axes_enter_event'`
- `'axes_leave_event'`
- `'close_event'`

For the location events (button and key press/release), if the mouse is over the axes, the variable *event.inaxes* will be set to the *Axes* the event occurs is over, and additionally, the variables *event.xdata* and *event.ydata* will be defined. This is the mouse location in data coords. See *KeyEvent* and *MouseEvent* for more info.

Return value is a connection id that can be used with *mpl_disconnect()*.

Example usage:

```python
def on_press(event):
    print('you pressed', event.button, event.xdata, event.ydata)

cid = canvas.mpl_connect('button_press_event', on_press)
```

**matplotlib.pyplot.contour**(*args, **kwargs*)

Plot contours.

*contour()* and *contourf()* draw contour lines and filled contours, respectively. Except as noted, function signatures and return values are the same for both versions.

*contourf()* differs from the MATLAB version in that it does not draw the polygon edges. To draw edges, add line contours with calls to *contour()*.

Call signatures:
contour(Z)

make a contour plot of an array Z. The level values are chosen automatically.

contour(X, Y, Z)

X, Y specify the (x, y) coordinates of the surface

contour(Z, N)
contour(X, Y, Z, N)

countour up to N automatically-chosen levels.

contour(Z, V)
contour(X, Y, Z, V)

draw contour lines at the values specified in sequence V, which must be in increasing order.

contourf(..., V)

fill the len(V) - 1 regions between the values in V, which must be in increasing order.

contour(Z, **kwargs)

Use keyword args to control colors, linewidth, origin, cmap ... see below for more details.

X and Y must both be 2-D with the same shape as Z, or they must both be 1-D such that len(X) is the number of columns in Z and len(Y) is the number of rows in Z.

C = contour(...) returns a QuadContourSet object.

Optional keyword arguments:

corner_mask: [ True | False | ‘legacy’ ] Enable/disable corner masking, which only has an effect if Z is a masked array. If False, any quad touching a masked point is masked out. If True, only the triangular corners of quads nearest those points are always masked out, other triangular corners comprising three unmasked points are contoured as usual. If ‘legacy’, the old contouring algorithm is used, which is equivalent to False and is deprecated, only remaining whilst the new algorithm is tested fully.

If not specified, the default is taken from rcParams[‘contour.corner_mask’], which is True unless it has been modified.

colors: [ None | string | (mpl_colors) ] If None, the colormap specified by cmap will be used.

If a string, like ‘r’ or ‘red’, all levels will be plotted in this color.

If a tuple of matplotlib color args (string, float, rgb, etc), different levels will be plotted in different colors in the order specified.

alpha: float The alpha blending value
**cmap**: [None | Colormap] A `cm` Colormap instance or None. If `cmap` is None and `colors` is None, a default Colormap is used.

**norm**: [None | Normalize] A `matplotlib.colors.Normalize` instance for scaling data values to colors. If `norm` is None and `colors` is None, the default linear scaling is used.

**vmin, vmax**: [None | scalar] If not None, either or both of these values will be supplied to the `matplotlib.colors.Normalize` instance, overriding the default color scaling based on `levels`.

**levels**: [level0, level1, ..., leveln] A list of floating point numbers indicating the level curves to draw, in increasing order; e.g., to draw just the zero contour pass `levels=[0]`

**origin**: [None | ‘upper’ | ‘lower’ | ‘image’] If None, the first value of Z will correspond to the lower left corner, location (0,0). If ‘image’, the rc value for `image.origin` will be used.

This keyword is not active if X and Y are specified in the call to `contour`.

**extent**: [None | (x0,x1,y0,y1)]

If `origin` is not None, then `extent` is interpreted as in `matplotlib.pyplot.imshow()`: it gives the outer pixel boundaries. In this case, the position of Z[0,0] is the center of the pixel, not a corner. If `origin` is None, then (x0, y0) is the position of Z[0,0], and (x1, y1) is the position of Z[-1,-1].

This keyword is not active if X and Y are specified in the call to `contour`.

**locator**: [None | ticker.Locator subclass] If `locator` is None, the default `MaxNLocator` is used. The locator is used to determine the contour levels if they are not given explicitly via the `V` argument.

**extend**: [‘neither’ | ‘both’ | ‘min’ | ‘max’] Unless this is ‘neither’, contour levels are automatically added to one or both ends of the range so that all data are included. These added ranges are then mapped to the special colormap values which default to the ends of the colormap range, but can be set via `matplotlib.colors.Colormap.set_under()` and `matplotlib.colors.Colormap.set_over()` methods.

**xunits, yunits**: [None | registered units] Override axis units by specifying an instance of a `matplotlib.units.ConversionInterface`.

**antialiased**: [True | False] enable antialiasing, overriding the defaults. For filled contours, the default is `True`. For line contours, it is taken from rc-Params['lines.antialiased'].

**nchunk**: [0 | integer] If 0, no subdivision of the domain. Specify a positive integer to divide the domain into subdomains of `nchunk` by `nchunk` quads. Chunking reduces the maximum length of polygons generated by the contouring algorithm which reduces the rendering workload passed on to the backend and also requires slightly less RAM. It can however introduce rendering artifacts at chunk boundaries depending on the backend, the `antialiased` flag and value of `alpha`. 

---

45.1. `matplotlib.pyplot` 1439
contour-only keyword arguments:

**linewidths:** [ None | number | tuple of numbers ] If linewidths is None, the default width in lines.linewidth in matplotlibrc is used.

If a number, all levels will be plotted with this linewidth.

If a tuple, different levels will be plotted with different linewidths in the order specified.

**linestyles:** [ None | ‘solid’ | ‘dashed’ | ‘dashdot’ | ‘dotted’ ] If linestyles is None, the default is ‘solid’ unless the lines are monochrome. In that case, negative contours will take their linestyle from the matplotlibrc contour.negative_linestyle setting.

linestyles can also be an iterable of the above strings specifying a set of linestyles to be used. If this iterable is shorter than the number of contour levels it will be repeated as necessary.

contourf-only keyword arguments:

**hatches:** A list of cross hatch patterns to use on the filled areas. If None, no hatching will be added to the contour. Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Note: contourf fills intervals that are closed at the top; that is, for boundaries \( z_1 \) and \( z_2 \), the filled region is:

\[
\{ z_1 < z \leq z_2 \}
\]

There is one exception: if the lowest boundary coincides with the minimum value of the \( z \) array, then that minimum value will be included in the lowest interval.

**Examples:**
Simplest default with labels
labels at selected locations
Single color - negative contours dashed
Single color - negative contours solid
Crazy lines
Lines with colorbar

-1.2  -0.8  -0.4   0.0   0.4   0.8   1.2

-3 -2 -1  0  1  2  3

-2.0  -1.5  -1.0  -0.5  0.0   0.5   1.0   1.5
Nonsense (3 masked regions)

Verbosity coefficient

Word length anomaly

Sentence length anomaly

Verbosity coefficient
Listed colors (3 masked regions)
extend = neither
extend = both
extend = min
extend = max
Matplotlib.pyplot.contourf(*args, **kwargs)

Plot contours.

contour() and contourf() draw contour lines and filled contours, respectively. Except as noted, function signatures and return values are the same for both versions.

contourf() differs from the MATLAB version in that it does not draw the polygon edges. To draw edges, add line contours with calls to contour().

Call signatures:

```
contour(Z)
```

make a contour plot of an array Z. The level values are chosen automatically.

```
contour(X,Y,Z)
```

X, Y specify the (x, y) coordinates of the surface

```
contour(Z,N)
contour(X,Y,Z,N)
```

contour up to N automatically-chosen levels.

```
contour(Z,V)
contour(X,Y,Z,V)
```
draw contour lines at the values specified in sequence \( V \), which must be in increasing order.

\[
\text{contour}(\ldots, V)
\]

fill the \( \text{len}(V) - 1 \) regions between the values in \( V \), which must be in increasing order.

\[
\text{contour}(Z, \ **\text{kwargs})
\]

Use keyword args to control colors, linewidth, origin, cmap ... see below for more details.

\( X \) and \( Y \) must both be 2-D with the same shape as \( Z \), or they must both be 1-D such that \( \text{len}(X) \) is the number of columns in \( Z \) and \( \text{len}(Y) \) is the number of rows in \( Z \).

\( C = \text{contour}(\ldots) \) returns a QuadContourSet object.

Optional keyword arguments:

- **corner mask**: [True | False | ‘legacy’] Enable/disable corner masking, which only has an effect if \( Z \) is a masked array. If False, any quad touching a masked point is masked out. If True, only the triangular corners of quads nearest those points are always masked out, other triangular corners comprising three unmasked points are contoured as usual. If ‘legacy’, the old contouring algorithm is used, which is equivalent to False and is deprecated, only remaining whilst the new algorithm is tested fully.

  If not specified, the default is taken from rcParams[‘contour.corner_mask’], which is True unless it has been modified.

- **colors**: [None | string | (mpl_colors)] If None, the colormap specified by cmap will be used.

  If a string, like ‘r’ or ‘red’, all levels will be plotted in this color.

  If a tuple of matplotlib color args (string, float, rgb, etc), different levels will be plotted in different colors in the order specified.

- **alpha**: float The alpha blending value

- **cmap**: [None | Colormap] A cm Colormap instance or None. If cmap is None and colors is None, a default Colormap is used.

- **norm**: [None | Normalize] A matplotlib.colors.Normalize instance for scaling data values to colors. If norm is None and colors is None, the default linear scaling is used.

- **vmin, vmax**: [None | scalar] If not None, either or both of these values will be supplied to the matplotlib.colors.Normalize instance, overriding the default color scaling based on levels.

- **levels**: [level0, level1, ..., leveln] A list of floating point numbers indicating the level curves to draw, in increasing order; e.g., to draw just the zero contour pass levels=[0]

- **origin**: [None | ‘upper’ | ‘lower’ | ‘image’] If None, the first value of \( Z \) will correspond to the lower left corner, location (0,0). If ‘image’, the rc value for image.origin will be used.
This keyword is not active if $X$ and $Y$ are specified in the call to contour.

`extent` : [ `None` | (x0,x1,y0,y1) ]

If `origin` is not `None`, then `extent` is interpreted as in `matplotlib.pyplot.imshow()`: it gives the outer pixel boundaries. In this case, the position of $Z[0,0]$ is the center of the pixel, not a corner. If `origin` is `None`, then $(x0, y0)$ is the position of $Z[0,0]$, and $(x1, y1)$ is the position of $Z[-1,-1]$.

This keyword is not active if $X$ and $Y$ are specified in the call to contour.

`locator` : [ `None` | `ticker.Locator subclass` ]

If `locator` is `None`, the default `MaxNLocator` is used. The locator is used to determine the contour levels if they are not given explicitly via the `V` argument.

`extend` : [ `‘neither’` | `‘both’` | `‘min’` | `‘max’` ]

Unless this is `‘neither’`, contour levels are automatically added to one or both ends of the range so that all data are included. These added ranges are then mapped to the special colormap values which default to the ends of the colormap range, but can be set via `matplotlib.colors.Colormap.set_under()` and `matplotlib.colors.Colormap.set_over()` methods.

`xunits`, `yunits` : [ `None` | `registered units` ]

Override axis units by specifying an instance of `matplotlib.units.ConversionInterface`.

`antialiased` : [ `True` | `False` ]

Enable antialiasing, overriding the defaults. For filled contours, the default is `True`. For line contours, it is taken from `rcParams['lines.antialiased']`.

`nchunk` : [ `0` | `integer` ]

If `0`, no subdivision of the domain. Specify a positive integer to divide the domain into subdomains of `nchunk` by `nchunk` quads. Chunking reduces the maximum length of polygons generated by the contouring algorithm which reduces the rendering workload passed on to the backend and also requires slightly less RAM. It can however introduce rendering artifacts at chunk boundaries depending on the backend, the `antialiased` flag and value of `alpha`.

countour-only keyword arguments:

`linewidths` : [ `None` | `number` | `tuple of numbers` ]

If `linewidths` is `None`, the default width in `lines.linewidth` in `matplotlibrc` is used.

If a number, all levels will be plotted with this linewidth.

If a tuple, different levels will be plotted with different linewidths in the order specified.

`linestyles` : [ `None` | `‘solid’` | `‘dashed’` | `‘dashdot’` | `‘dotted’` ]

If `linestyles` is `None`, the default is `‘solid’` unless the lines are monochrome. In that case, negative contours will take their linestyle from the `matplotlibrc contour.negative_linestyle` setting.

`linestyles` can also be an iterable of the above strings specifying a set of linestyles to be used. If this iterable is shorter than the number of contour levels it will be repeated as necessary.
contourf-only keyword arguments:

**hatches**: A list of cross hatch patterns to use on the filled areas. If None, no hatching will be added to the contour. Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Note: contourf fills intervals that are closed at the top; that is, for boundaries $z_1$ and $z_2$, the filled region is:

$$z_1 < z \leq z_2$$

There is one exception: if the lowest boundary coincides with the minimum value of the $z$ array, then that minimum value will be included in the lowest interval.

**Examples:**
labels at selected locations
Single color - negative contours dashed
Single color - negative contours solid
Crazy lines

-3 -2 -1 0 1 2
-2.0 -1.5 -1.0 -0.5 0.0 0.5 1.0 1.5 2.0
-1.000 -0.500 0.000 0.500 1.000 1.500

0.000 0.500 1.000 1.500

matplotlib.pyplot

45.1. matplotlib.pyplot

1457
Lines with colorbar
Nonsense (3 masked regions)
Listed colors (3 masked regions)
extend = neither

extend = both

extend = min

extend = max
matplotlib.pyplot.cool()

set the default colormap to cool and apply to current image if any. See help(colormaps) for more information

matplotlib.pyplot.copper()

set the default colormap to copper and apply to current image if any. See help(colormaps) for more information

matplotlib.pyplot.csd(x, y, NFFT=None, Fs=None, Fc=None, detrend=mlab.detrend_none, window=mlab.hanning, noverlap=0, pad_to=None, sides='default', scale_by_freq=None, return_line=None, hold=None, data=None, **kwargs)

Plot the cross-spectral density.

Call signature:

```
csd(x, y, NFFT=256, Fs=2, Fc=0, detrend=mlab.detrend_none, window=mlab.hanning, noverlap=0, pad_to=None, sides='default', scale_by_freq=None, return_line=None, hold=None, data=None, **kwargs)
```

The cross spectral density $P_{xy}$ by Welch’s average periodogram method. The vectors $x$ and $y$ are divided into $NFFT$ length segments. Each segment is detrended by function `detrend` and windowed by function `window`. `noverlap` gives the length of the overlap between segments. The product of the direct FFTs of $x$ and $y$ are averaged over each segment to compute $P_{xy}$, with a scaling to correct for power loss due to windowing.

If len($x$) < $NFFT$ or len($y$) < $NFFT$, they will be zero padded to $NFFT$. 
Parameters x, y : 1-D arrays or sequences

Arrays or sequences containing the data

Fs : scalar

The sampling frequency (samples per time unit). It is used to calculate the
Fourier frequencies, freqs, in cycles per time unit. The default value is 2.

window : callable or ndarray

A function or a vector of length NFFT. To create window vectors
see window_hanning(), window_none(), numpy.blackman(), numpy.
hamming(), numpy.bartlett(), scipy.signal(), scipy.signal.
get_window(), etc. The default is window_hanning(). If a function is
passed as the argument, it must take a data segment as an argument and re-
turn the windowed version of the segment.

sides : [‘default’ | ‘onesided’ | ‘twosided’ ]

Specifies which sides of the spectrum to return. Default gives the default
behavior, which returns one-sided for real data and both for complex data.
‘onesided’ forces the return of a one-sided spectrum, while ‘twosided’ forces
two-sided.

pad_to : integer

The number of points to which the data segment is padded when performing
the FFT. This can be different from NFFT, which specifies the number of data
points used. While not increasing the actual resolution of the spectrum (the
minimum distance between resolvable peaks), this can give more points in
the plot, allowing for more detail. This corresponds to the n parameter in the
call to fft(). The default is None, which sets pad_to equal to NFFT

NFFT : integer

The number of data points used in each block for the FFT. A power 2 is most
efficient. The default value is 256. This should NOT be used to get zero
padding, or the scaling of the result will be incorrect. Use pad_to for this
instead.

detrend : {‘default’, ‘constant’, ‘mean’, ‘linear’, ‘none’} or callable

The function applied to each segment before fft-ing, designed to remove
the mean or linear trend. Unlike in MATLAB, where the detrend param-
eter is a vector, in matplotlib is it a function. The pylab module defines
detrend_none(), detrend_mean(), and detrend_linear(), but you can
use a custom function as well. You can also use a string to choose one of the
functions. ‘default’, ‘constant’, and ‘mean’ call detrend_mean(). ‘linear’
calls detrend_linear(). ‘none’ calls detrend_none().

scale_by_freq : boolean, optional

 Specifies whether the resulting density values should be scaled by the scaling
frequency, which gives density in units of Hz^-1. This allows for integra-
tion over the returned frequency values. The default is True for MATLAB compatibility.

**noverlap**: integer
The number of points of overlap between segments. The default value is 0 (no overlap).

**Fc**: integer
The center frequency of \( x \) (defaults to 0), which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

**return_line**: bool
Whether to include the line object plotted in the returned values. Default is False.

**kwargs**
Keyword arguments control the Line2D properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>drawstyle</td>
<td>['default'</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>['full'</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['solid'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>marker</td>
<td>A valid marker style</td>
</tr>
<tr>
<td>markeredgecolor or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
</tbody>
</table>

Continued on next page
Table 45.11 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>markevery</td>
<td>[None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

**Returns** $P_{xy}$ : 1-D array

The values for the cross spectrum $P_{xy}$ before scaling (complex valued)

**freqs** : 1-D array

The frequencies corresponding to the elements in $P_{xy}$

**line** : a Line2D instance

The line created by this function. Only returned if `return_line` is True.

**See also:**

`psd()` `psd()` is the equivalent to setting $y=x$.

In addition to the above described arguments, this function can take a data keyword argument. If such a data argument is given, the following arguments are replaced by `data[<arg>]`: * All arguments with the following names: ‘x’, ‘y’.

**Notes**

For plotting, the power is plotted as $10 \log_{10}(P_{xy})$ for decibels, though $P_{xy}$ itself is returned.

**References**

Examples

```
import matplotlib.pyplot as plt

# Example 1: CSD (cross-spectral density)
fig, (ax1, ax2) = plt.subplots(2, 1, sharex=True)

# Generate data
fs = 1000  # Sampling frequency
frq = np.arange(0, 0.5, 1.0 / fs)  # Frequency vector

# CSD example
ax1.plot(frq, 20 * np.log10(np.abs(np.fft.fftshift(np.fft.fft(s1 - s2)))))
ax1.set_xlabel('Frequency (Hz)')
ax1.set_ylabel('CSD (db)')

# Time series example
ax2.plot(np.arange(len(s1)), s1)
ax2.set_xlabel('Time (s)')
ax2.set_ylabel('Signal')

plt.show()
```

```
# Example 2: Disconnecting callbacks

cid = canvas.mpl_connect('button_press_event', on_press)

# ...later ...

canvas.mpl_disconnect(cid)
```

```
# Example 3: Redrawing the figure

delaxes()  # Remove the current axes

draw()    # Redraw the figure
```

matplotlib.pyplot.delaxes(*args)

Remove an axes from the current figure. If ax doesn’t exist, an error will be raised.

delaxes(): delete the current axes

matplotlib.pyplot.disconnect(cid)

Disconnect callback id cid

Example usage:

```
cid = canvas.mpl_connect('button_press_event', on_press)
#...later

canvas.mpl_disconnect(cid)
```

matplotlib.pyplot.draw()

Redraw the current figure.

This is used to update a figure that has been altered, but not automatically re-drawn. If interactive mode is on (ion()), this should be only rarely needed, but there may be ways to modify the state of a figure without marking it as stale. Please report these cases as bugs.

A more object-oriented alternative, given any Figure instance, fig, that was created using a pyplot function, is:
matplotlib.pyplot.errorbar(x, y, yerr=None, xerr=None, fmt='', ecolor=None, elinewidth=None, capsize=None, barsabove=False, lolims=False, uplims=False, xlimits=False, ylimits=False, errorevery=1, capthick=None, hold=None, data=None, **kwargs)

Plot an errorbar graph.

Plot x versus y with error deltas in yerr and xerr. Vertical errorbars are plotted if yerr is not None. Horizontal errorbars are plotted if xerr is not None.

x, y, xerr, and yerr can all be scalars, which plots a single error bar at x, y.

**Parameters**

- **x** : scalar or array-like
- **y** : scalar or array-like
- **xerr/yerr** : scalar or array-like, shape(N,) or shape(2,N), optional
  - If a scalar number, len(N) array-like object, or a N-element array-like object, errorbars are drawn at +/-value relative to the data. Default is None.
  - If a sequence of shape 2xN, errorbars are drawn at -row1 and +row2 relative to the data.
- **fmt** : plot format string, optional, default: None
  - The plot format symbol. If fmt is ‘none’ (case-insensitive), only the errorbars are plotted. This is used for adding errorbars to a bar plot, for example. Default is ‘’, an empty plot format string; properties are then identical to the defaults for `plot()`.
- **ecolor** : mpl color, optional, default: None
  - A matplotlib color arg which gives the color the errorbar lines; if None, use the color of the line connecting the markers.
- **elinewidth** : scalar, optional, default: None
  - The linewidth of the errorbar lines. If None, use the linewidth.
- **capsize** : scalar, optional, default: None
  - The length of the error bar caps in points; if None, it will take the value from `errorbar.capsize rcParam`.
- **capthick** : scalar, optional, default: None
  - An alias kwarg to markeredgewidth (a.k.a. - mew). This setting is a more sensible name for the property that controls the thickness of the error bar cap in points. For backwards compatibility, if mew or markeredgewidth are given, then they will over-ride capthick. This may change in future releases.
- **barsabove** : bool, optional, default: False
  - If True, will plot the errorbars above the plot symbols. Default is below.
lolims / uplims / xlolims / xuplims : bool, optional, default:None

These arguments can be used to indicate that a value gives only upper/lower limits. In that case a caret symbol is used to indicate this. lims-arguments may be of the same type as xerr and yerr. To use limits with inverted axes, set_xlim() or set_ylim() must be called before errorbar().

errorevery : positive integer, optional, default:1

subsamples the errorbars. e.g., if errorevery=5, errorbars for every 5-th data-point will be plotted. The data plot itself still shows all data points.

Returns plotline : Line2D instance

x, y plot markers and/or line
caplines : list of Line2D instances

error bar cap
barlinecols : list of LineCollection

horizontal and vertical error ranges.

Other Parameters kwargs : All other keyword arguments are passed on to the plot command for the markers. For example, this code makes big red squares with thick green edges:

```python
x, y, yerr = rand(3, 10)
errorbar(x, y, yerr, marker='s', mfc='red', mec='green', ms=20, mew=4)
```

where mfc, mec, ms and mew are aliases for the longer property names, markerfacecolor, markeredgecolor, markersize and markeredgewidth.

valid kwargs for the marker properties are

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>drawstyle</td>
<td>['default'</td>
</tr>
</tbody>
</table>
Table 45.12 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>figure</code></td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td><code>fillstyle</code></td>
<td>['full', 'left', 'right', 'bottom', 'top', 'none']</td>
</tr>
<tr>
<td><code>gid</code></td>
<td>an id string</td>
</tr>
<tr>
<td><code>label</code></td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td><code>linestyle</code> or <code>ls</code></td>
<td>['solid', 'dashed', 'dashdot', 'dotted']</td>
</tr>
<tr>
<td><code>linewidth</code> or <code>lw</code></td>
<td>float value in points</td>
</tr>
<tr>
<td><code>marker</code></td>
<td><em>A valid marker style</em></td>
</tr>
<tr>
<td><code>markeredgecolor</code> or <code>mec</code></td>
<td>any matplotlib color</td>
</tr>
<tr>
<td><code>markeredgewidth</code> or <code>mew</code></td>
<td>float value in points</td>
</tr>
<tr>
<td><code>markerfacecolor</code> or <code>mfc</code></td>
<td>any matplotlib color</td>
</tr>
<tr>
<td><code>markerfacecoloralt</code> or <code>mfcalt</code></td>
<td>any matplotlib color</td>
</tr>
<tr>
<td><code>markersize</code> or <code>ms</code></td>
<td>float</td>
</tr>
<tr>
<td><code>markevery</code></td>
<td>[None</td>
</tr>
<tr>
<td><code>path_effects</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>picker</code></td>
<td>float distance in points or callable pick function <code>fn(artist, event)</code></td>
</tr>
<tr>
<td><code>pickradius</code></td>
<td>float distance in points</td>
</tr>
<tr>
<td><code>rasterized</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>sketch_params</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>snap</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>solid_capstyle</code></td>
<td>['butt', 'round', 'projecting']</td>
</tr>
<tr>
<td><code>solid_joinstyle</code></td>
<td>['miter', 'round', 'bevel']</td>
</tr>
<tr>
<td><code>transform</code></td>
<td>a <code>matplotlib.transforms.Transform</code> instance</td>
</tr>
<tr>
<td><code>url</code></td>
<td>a url string</td>
</tr>
<tr>
<td><code>visible</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>xdata</code></td>
<td>1D array</td>
</tr>
<tr>
<td><code>ydata</code></td>
<td>1D array</td>
</tr>
<tr>
<td><code>zorder</code></td>
<td>any number</td>
</tr>
</tbody>
</table>
**Examples**

```
matplotlib.pyplot.eventplot(positions, orientation='horizontal', lineoffsets=1, line
lengths=1, linewidths=None, colors=None, linestyles='solid', hold=None, data=None, **kwargs)
```

Plot identical parallel lines at specific positions.

Plot parallel lines at the given positions. positions should be a 1D or 2D array-like object, with each row corresponding to a row or column of lines.

This type of plot is commonly used in neuroscience for representing neural events, where it is commonly called a spike raster, dot raster, or raster plot.

However, it is useful in any situation where you wish to show the timing or position of multiple sets of discrete events, such as the arrival times of people to a business on each day of the month or the date of hurricanes each year of the last century.

**orientation** [[ ‘horizontal’ | ‘vertical’ ]] ‘horizontal’ : the lines will be vertical and arranged in rows
‘vertical’ : lines will be horizontal and arranged in columns

**Note:** In addition to the above described arguments, this function can take a `data` keyword argument. If such a `data` argument is given, the following arguments are replaced by `data[<arg>]`:

- All arguments with the following names: ‘x’, ‘xerr’, ‘y’, ‘yerr’.
**lineoffsets** : A float or array-like containing floats.

**lineoffsets** : A float or array-like containing floats.

**linewidths** : A float or array-like containing floats.

**colors** must be a sequence of RGBA tuples (e.g., arbitrary color strings, etc, not allowed) or a list of such sequences

**linestyles** : ['solid' | 'dashed' | 'dashdot' | 'dotted'] or an array of these values

For linelengths, linewidths, colors, and linestyles, if only a single value is given, that value is applied to all lines. If an array-like is given, it must have the same length as positions, and each value will be applied to the corresponding row or column in positions.

Returns a list of `matplotlib.collections.EventCollection` objects that were added.

**kwargs** are `LineCollection` properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>agg_filter</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>alpha</code></td>
<td>float or None</td>
</tr>
<tr>
<td><code>animated</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>antialiased</code> or antialiaseds</td>
<td>Boolean or sequence of booleans</td>
</tr>
<tr>
<td><code>array</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>axes</code></td>
<td>an <code>Axes</code> instance</td>
</tr>
<tr>
<td><code>clim</code></td>
<td>a length 2 sequence of floats</td>
</tr>
<tr>
<td><code>clip_box</code></td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td><code>clip_on</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>clip_path</code></td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td><code>cmap</code></td>
<td>a colormap or registered colormap name</td>
</tr>
<tr>
<td><code>color</code></td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td><code>contains</code></td>
<td>a callable function</td>
</tr>
<tr>
<td><code>edgecolor</code> or edgecolors</td>
<td>matplotlib color spec or sequence of specs</td>
</tr>
<tr>
<td><code>facecolor</code> or facecolors</td>
<td>matplotlib color spec or sequence of specs</td>
</tr>
<tr>
<td><code>figure</code></td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td><code>gid</code></td>
<td>an id string</td>
</tr>
<tr>
<td><code>hatch</code></td>
<td>['/'</td>
</tr>
<tr>
<td><code>label</code></td>
<td>string or anything printable with '%s' conversion.</td>
</tr>
<tr>
<td><code>linestyle</code> or dashes or linestyles</td>
<td>['solid'</td>
</tr>
<tr>
<td><code>linewidth</code> or linewidths or lw</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td><code>norm</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>offset_position</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>offsets</code></td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td><code>path_effects</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>paths</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>picker</code></td>
<td>[None</td>
</tr>
<tr>
<td><code>pickradius</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>rasterized</code></td>
<td>[True</td>
</tr>
</tbody>
</table>
Table 45.13 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>segments</td>
<td>unknown</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>urls</td>
<td>unknown</td>
</tr>
<tr>
<td>verts</td>
<td>unknown</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

Example:

![Example Image]

**Note:** In addition to the above described arguments, this function can take a `data` keyword argument. If such a `data` argument is given, the following arguments are replaced by `data[<arg>]`:


---

Chapter 45. pyplot
Matplotlib.pyplot.figimage(*args, **kwargs)

Adds a non-resampled image to the figure.

call signatures:

```
figimage(X, **kwargs)
```

adds a non-resampled array $X$ to the figure.

```
figimage(X, xo, yo)
```

with pixel offsets $xo$, $yo$.

$X$ must be a float array:

- If $X$ is MxN, assume luminance (grayscale)
- If $X$ is MxNx3, assume RGB
- If $X$ is MxNx4, assume RGBA

Optional keyword arguments:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>resize</td>
<td>a boolean, True or False. If “True”, then re-size the Figure to match the given image size.</td>
</tr>
<tr>
<td>xo or yo</td>
<td>An integer, the $x$ and $y$ image offset in pixels</td>
</tr>
<tr>
<td>cmap</td>
<td>a <code>matplotlib.colors.Colormap</code> instance, e.g., cm.jet. If <code>None</code>, default to the rc <code>image.cmap</code> value</td>
</tr>
<tr>
<td>norm</td>
<td>a <code>matplotlib.colors.Normalize</code> instance. The default is normalization(). This scales luminance -&gt; 0-1</td>
</tr>
<tr>
<td>vmin</td>
<td><code>vmin</code> used to scale a luminance image to 0-1. If either is <code>None</code>, the min and max of the luminance values will be used. Note if you pass a norm instance, the settings for <code>vmin</code> and <code>vmax</code> will be ignored.</td>
</tr>
<tr>
<td>vmax</td>
<td><code>vmax</code> used to scale a luminance image to 0-1. If either is <code>None</code>, the min and max of the luminance values will be used. Note if you pass a norm instance, the settings for <code>vmin</code> and <code>vmax</code> will be ignored.</td>
</tr>
<tr>
<td>alpha</td>
<td>the alpha blending value, default is <code>None</code></td>
</tr>
<tr>
<td>origin</td>
<td>[‘upper’, ‘lower’] Indicates where the [0,0] index of the array is in the upper left or lower left corner of the axes. Defaults to the rc <code>image.origin</code> value</td>
</tr>
</tbody>
</table>

figimage complements the axes image (`imshow()`) which will be resampled to fit the current axes. If you want a resampled image to fill the entire figure, you can define an `Axes` with extent [0,0,1,1].

An `matplotlib.image.FigureImage` instance is returned.
Additional kwargs are Artist kwargs passed on to `FigureImage`

```python
matplotlib.pyplot.figlegend(handles, labels, loc, **kwargs)
```
Place a legend in the figure.

- `labels` a sequence of strings
- `handles` a sequence of `Line2D` or `Patch` instances
- `loc` can be a string or an integer specifying the legend location

A `matplotlib.legend.Legend` instance is returned.

Example:

```python
figlegend((line1, line2, line3),
          ('label1', 'label2', 'label3'),
          'upper right')
```

See also:

- `legend()`
- `fignum_exists(num)`
- `figtext(*args, **kwargs)`
  Add text to figure.
Call signature:

\[
\text{text}(x, y, s, fontdict=None, **kwargs)
\]

Add text to figure at location \( x, y \) (relative 0-1 coords). See \texttt{text()} for the meaning of the other arguments.

\texttt{kwargs} control the \texttt{Text} properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an \texttt{Axes} instance</td>
</tr>
<tr>
<td>backgroundcolor</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>bbox</td>
<td>FancyBboxPatch prop dict</td>
</tr>
<tr>
<td>clip_box</td>
<td>a \texttt{matplotlib.transforms.Bbox} instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(\texttt{Path}, \texttt{Transform})</td>
</tr>
<tr>
<td>color</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>family or fontfamily or fontname or name</td>
<td>[\texttt{FONTNAME}</td>
</tr>
<tr>
<td>figure</td>
<td>a \texttt{matplotlib.figure.Figure} instance</td>
</tr>
<tr>
<td>fontproperties or font_properties</td>
<td>a \texttt{matplotlib.font_manager.FontProperties} instance</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>horizontalalignment or ha</td>
<td>[‘center’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linespacing</td>
<td>float (multiple of font size)</td>
</tr>
<tr>
<td>multialignment</td>
<td>[‘left’</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>position</td>
<td>(x,y)</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>rotation</td>
<td>[ angle in degrees</td>
</tr>
<tr>
<td>rotation_mode</td>
<td>unknown</td>
</tr>
<tr>
<td>size or fontsize</td>
<td>[size in points</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>stretch or fontstretch</td>
<td>[a numeric value in range 0-1000</td>
</tr>
<tr>
<td>style or fontstyle</td>
<td>[ ‘normal’</td>
</tr>
<tr>
<td>text</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>transform</td>
<td>\texttt{Transform} instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>usetex</td>
<td>unknown</td>
</tr>
<tr>
<td>variant or fontvariant</td>
<td>[ ‘normal’</td>
</tr>
<tr>
<td>verticalalignment or ma or va</td>
<td>[ ‘center’</td>
</tr>
</tbody>
</table>

45.1. matplotlib.pyplot 1475
Table 45.14 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>wrap</td>
<td>unknown</td>
</tr>
<tr>
<td>x</td>
<td>float</td>
</tr>
<tr>
<td>y</td>
<td>float</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

`matplotlib.pyplot.figure(num=None, figsize=None, dpi=None, facecolor=None, edgecolor=None, frameon=True, FigureClass=<class 'matplotlib.figure.Figure'>, **kwargs)`

Creates a new figure.

**Parameters**

- **num** : integer or string, optional, default: none
  - If not provided, a new figure will be created, and the figure number will be incremented. The figure objects holds this number in a number attribute. If num is provided, and a figure with this id already exists, make it active, and returns a reference to it. If this figure does not exists, create it and returns it. If num is a string, the window title will be set to this figure’s num.

- **figsize** : tuple of integers, optional, default: None
  - width, height in inches. If not provided, defaults to rc figure.figsize.

- **dpi** : integer, optional, default: None
  - resolution of the figure. If not provided, defaults to rc figure.dpi.

- **facecolor** :
  - the background color. If not provided, defaults to rc figure.facecolor

- **edgecolor** :
  - the border color. If not provided, defaults to rc figure.edgecolor

**Returns** **figure** : Figure

- The Figure instance returned will also be passed to new_figure_manager in the backends, which allows to hook custom Figure classes into the pylab interface. Additional kwargs will be passed to the figure init function.

**Notes**

- If you are creating many figures, make sure you explicitly call “close” on the figures you are not using, because this will enable pylab to properly clean up the memory.

- rcParams defines the default values, which can be modified in the matplotlibrc file
matplotlib.pyplot.fill(*args, **kwargs)
Plot filled polygons.

**Parameters**

arg : a variable length argument

It allows for multiple x, y pairs with an optional color format string; see
`plot()` for details on the argument parsing. For example, each of the follow-
ing is legal:

```python
ax.fill(x, y)
ax.fill(x, y, "b")
ax.fill(x, y, "b", x, y, "r")
```

An arbitrary number of x, y, color groups can be specified:
```
ax.fill(x1, y1, 'g',
x2, y2, 'r')
```

**Returns** a list of `Patch`

**Other Parameters**

kwarg : `Polygon` properties

**Notes**

The same color strings that `plot()` supports are supported by the fill format string.

If you would like to fill below a curve, e.g., shade a region between 0 and y along x, use
`fill_between()`
Examples

Note: In addition to the above described arguments, this function can take a data keyword argument. If such a data argument is given, the following arguments are replaced by data[<arg>]:

- All arguments with the following names: ‘x’, ‘y’.

```
matplotlib.pyplot.fill_between(x, y1, y2=0, where=None, interpolate=False, step=None, hold=None, data=None, **kwargs)
```

Make filled polygons between two curves.

Create a PolyCollection filling the regions between $y1$ and $y2$ where $where==True$

**Parameters**

- **x**: array
  An N-length array of the x data
- **y1**: array
  An N-length array (or scalar) of the y data
- **y2**: array
  An N-length array (or scalar) of the y data
- **where**: array, optional
If `None`, default to fill between everywhere. If not `None`, it is an N-length numpy boolean array and the fill will only happen over the regions where `where==True`.

**interpolate**: bool, optional

If `True`, interpolate between the two lines to find the precise point of intersection. Otherwise, the start and end points of the filled region will only occur on explicit values in the `x` array.

**step**: `{'pre', 'post', 'mid'}`, optional

If not `None`, fill with step logic.

**See also:**

In addition to the above described arguments, this function can take a `data` keyword argument. If such a `data` argument is given, the following arguments are replaced by `data[<arg>]:` * All arguments with the following names: `where`, `x`, `y1`, `y2`.

**Notes**

Additional keyword args passed on to the `PolyCollection`.

**kwargs** control the `Polygon` properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>agg_filter</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>alpha</code></td>
<td>float or None</td>
</tr>
<tr>
<td><code>animated</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>antialiased</code> or <code>antialiaseds</code></td>
<td>Boolean or sequence of booleans</td>
</tr>
<tr>
<td><code>array</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>axes</code></td>
<td>an <code>Axes</code> instance</td>
</tr>
<tr>
<td><code>clim</code></td>
<td>a length 2 sequence of floats</td>
</tr>
<tr>
<td><code>clip_box</code></td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td><code>clip_on</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>clip_path</code></td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td><code>cmap</code></td>
<td>a colormap or registered colormap name</td>
</tr>
<tr>
<td><code>color</code></td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td><code>contains</code></td>
<td>a callable function</td>
</tr>
<tr>
<td><code>edgecolor</code> or <code>edgecolors</code></td>
<td>matplotlib color spec or sequence of specs</td>
</tr>
<tr>
<td><code>facecolor</code> or <code>facecolors</code></td>
<td>matplotlib color spec or sequence of specs</td>
</tr>
<tr>
<td><code>figure</code></td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td><code>gid</code></td>
<td>an id string</td>
</tr>
<tr>
<td><code>hatch</code></td>
<td>['/'</td>
</tr>
<tr>
<td><code>label</code></td>
<td>string or anything printable with <code>%s</code> conversion.</td>
</tr>
<tr>
<td><code>linestyle</code> or <code>dashes</code> or <code>linestyles</code></td>
<td>['solid'</td>
</tr>
<tr>
<td><code>linewidth</code> or <code>linewights</code> or <code>lw</code></td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td><code>norm</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>offset_position</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>offsets</code></td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td><code>path_effects</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>picker</code></td>
<td>[None</td>
</tr>
<tr>
<td><code>pickradius</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>rasterized</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>sketch_params</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>snap</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>transform</code></td>
<td><code>Transform</code> instance</td>
</tr>
<tr>
<td><code>url</code></td>
<td>a url string</td>
</tr>
<tr>
<td><code>urls</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>visible</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>zorder</code></td>
<td>any number</td>
</tr>
</tbody>
</table>

**Examples**

![Graphs](image-url)
fill between where

Now regions with $y_2 > 1$ are masked
Matplotlib.pyplot.fill_betweenx(y, x1, x2=None, where=True, step=None, hold=None, data=None, **kwargs)

Make filled polygons between two horizontal curves.

Create a PolyCollection filling the regions between x1 and x2 where where=True

Parameters

- **y**: array
  - An N-length array of the y data
- **x1**: array
  - An N-length array (or scalar) of the x data
- **x2**: array, optional
  - An N-length array (or scalar) of the x data
- **where**: array, optional
  - If None, default to fill between everywhere. If not None, it is a N length numpy boolean array and the fill will only happen over the regions where where==True
- **step**: {'pre', 'post', 'mid'}, optional
  - If not None, fill with step logic.

See also:
In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**: * All arguments with the following names: ‘where’, ‘x1’, ‘x2’, ‘y’.

**Notes**

**keyword args passed on to the** *PolyCollection*

kwargs control the *Polygon* properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>agg_filter</strong></td>
<td>unknown</td>
</tr>
<tr>
<td><strong>alpha</strong></td>
<td>float or None</td>
</tr>
<tr>
<td><strong>animated</strong></td>
<td>[True</td>
</tr>
<tr>
<td><strong>antialiased</strong> or antialiaseds</td>
<td>Boolean or sequence of booleans</td>
</tr>
<tr>
<td><strong>array</strong></td>
<td>unknown</td>
</tr>
<tr>
<td><strong>axes</strong></td>
<td>an <em>Axes</em> instance</td>
</tr>
<tr>
<td><strong>clim</strong></td>
<td>a length 2 sequence of floats</td>
</tr>
<tr>
<td><strong>clip_box</strong></td>
<td>a <em>matplotlib.transforms.Bbox</em> instance</td>
</tr>
<tr>
<td><strong>clip_on</strong></td>
<td>[True</td>
</tr>
<tr>
<td><strong>clip_path</strong></td>
<td>[ *(Path, Transform)</td>
</tr>
<tr>
<td><strong>cmap</strong></td>
<td>a colormap or registered colormap name</td>
</tr>
<tr>
<td><strong>color</strong></td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td><strong>contains</strong></td>
<td>a callable function</td>
</tr>
<tr>
<td><strong>edgecolor</strong> or edgecolors</td>
<td>matplotlib color spec or sequence of specs</td>
</tr>
<tr>
<td><strong>facecolor</strong> or facecolors</td>
<td>matplotlib color spec or sequence of specs</td>
</tr>
<tr>
<td><strong>figure</strong></td>
<td>a <em>matplotlib.figure.Figure</em> instance</td>
</tr>
<tr>
<td><strong>gid</strong></td>
<td>an id string</td>
</tr>
<tr>
<td><strong>hatch</strong></td>
<td>[ ‘/’</td>
</tr>
<tr>
<td><strong>label</strong></td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td><strong>linestyle</strong> or dashes or linestyles</td>
<td>[‘solid’</td>
</tr>
<tr>
<td><strong>linewidth</strong> or linewidths or lw</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td><strong>norm</strong></td>
<td>unknown</td>
</tr>
<tr>
<td><strong>offset_position</strong></td>
<td>unknown</td>
</tr>
<tr>
<td><strong>offsets</strong></td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td><strong>path_effects</strong></td>
<td>unknown</td>
</tr>
<tr>
<td><strong>picker</strong></td>
<td>[None</td>
</tr>
<tr>
<td><strong>pickradius</strong></td>
<td>unknown</td>
</tr>
<tr>
<td><strong>rasterized</strong></td>
<td>[True</td>
</tr>
<tr>
<td><strong>sketch_params</strong></td>
<td>unknown</td>
</tr>
<tr>
<td><strong>snap</strong></td>
<td>unknown</td>
</tr>
<tr>
<td><strong>transform</strong></td>
<td><em>Transform</em> instance</td>
</tr>
<tr>
<td><strong>url</strong></td>
<td>a url string</td>
</tr>
<tr>
<td><strong>urls</strong></td>
<td>unknown</td>
</tr>
<tr>
<td><strong>visible</strong></td>
<td>[True</td>
</tr>
</tbody>
</table>
Table 45.16 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

Examples

![Graph examples](image_url)
**Find artist objects.**

Recursively find all *Artist* instances contained in self.

*match* can be

- None: return all objects contained in artist.
- function with signature `boolean = match(artist)` used to filter matches
- class instance: e.g., Line2D. Only return artists of class type.

If `include_self` is True (default), include self in the list to be checked for a match.

**flag()**

set the default colormap to flag and apply to current image if any. See help(colormaps) for more information

**gca(****kwargs**)**

Get the current *Axes* instance on the current figure matching the given keyword args, or create one.

See also:

- `matplotlib.figure.Figure.gca` The figure’s gca method.
Examples

To get the current polar axes on the current figure:

```python
plt.gca(projection='polar')
```

If the current axes doesn’t exist, or isn’t a polar one, the appropriate axes will be created and then returned.

```python
matplotlib.pyplot.gcf()
```

Get a reference to the current figure.

```python
matplotlib.pyplot.gci()
```

Get the current colorable artist. Specifically, returns the current `ScalarMappable` instance (image or patch collection), or `None` if no images or patch collections have been defined. The commands `imshow()` and `figimage()` create `Image` instances, and the commands `pcolor()` and `scatter()` create `Collection` instances. The current image is an attribute of the current axes, or the nearest earlier axes in the current figure that contains an image.

```python
matplotlib.pyplot.get_current_fig_manager()
```

```python
matplotlib.pyplot.get_fignums()
```

Return a list of existing figure numbers.

```python
matplotlib.pyplot.get_plot_commands()
```

Get a sorted list of all of the plotting commands.

```python
matplotlib.pyplot.ginput(*args, **kwargs)
```

Blocking call to interact with the figure.

This will wait for $n$ clicks from the user and return a list of the coordinates of each click.

If `timeout` is zero or negative, does not timeout.

If $n$ is zero or negative, accumulate clicks until a middle click (or potentially both mouse buttons at once) terminates the input.

Right clicking cancels last input.

The buttons used for the various actions (adding points, removing points, terminating the inputs) can be overridden via the arguments `mouse_add`, `mouse_pop` and `mouse_stop`, that give the associated mouse button: 1 for left, 2 for middle, 3 for right.

The keyboard can also be used to select points in case your mouse does not have one or more of the buttons. The delete and backspace keys act like right clicking (i.e., remove last point), the enter key terminates input and any other key (not already used by the window manager) selects a point.

```python
matplotlib.pyplot.gray()
```

Set the default colormap to gray and apply to current image if any. See `help(colormaps)` for more information.
matplotlib.pyplot.grid(b=None, which='major', axis='both', **kwargs)

Turn the axes grids on or off.

Set the axes grids on or off; b is a boolean. (For MATLAB compatibility, b may also be a string, ‘on’ or ‘off’.)

If b is None and len(kwargs)==0, toggle the grid state. If kwargs are supplied, it is assumed that you want a grid and b is thus set to True.

which can be ‘major’ (default), ‘minor’, or ‘both’ to control whether major tick grids, minor tick grids, or both are affected.

axis can be ‘both’ (default), ‘x’, or ‘y’ to control which set of gridlines are drawn.

kwargs are used to set the grid line properties, e.g.:

ax.grid(color='r', linestyle='-', linewidth=2)

Valid Line2D kwargs are:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>drawstyle</td>
<td>['default'</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>['full'</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['solid'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>marker</td>
<td>A valid marker style</td>
</tr>
<tr>
<td>markeredgcolor or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgwidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>[None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
</tbody>
</table>
Table 45.17 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function ( fn(\text{artist}, \text{event}) )</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a \texttt{matplotlib.transforms.Transform} instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

\texttt{matplotlib.pyplot.hexbin}(x, y, \( C=\text{None} \), gridsize=100, bins=\text{None}, xscale='linear', yscale='linear', extent=\text{None}, cmap=\text{None}, norm=\text{None}, vmin=\text{None}, vmax=\text{None}, alpha=\text{None}, linewidths=\text{None}, edgecolors=\text{none}', reduce_C_function=\text{function mean}, mincnt=\text{None}, marginals=\text{False}, hold=\text{None}, data=\text{None}, **kwargs)

Make a hexagonal binning plot.

Make a hexagonal binning plot of \( x \) versus \( y \), where \( x, y \) are 1-D sequences of the same length, \( N \). If \( C \) is \text{None} (the default), this is a histogram of the number of occurrences of the observations at \( (x[i],y[i]) \).

If \( C \) is specified, it specifies values at the coordinate \( (x[i],y[i]) \). These values are accumulated for each hexagonal bin and then reduced according to \text{reduce}_\text{C}_\text{function}, which defaults to numpy’s mean function (np.mean). (If \( C \) is specified, it must also be a 1-D sequence of the same length as \( x \) and \( y \).)

**Parameters** \( x, y \) : array or masked array

- \( C \) : array or masked array, optional, default is \text{None}
- \text{gridsize} : int or (int, int), optional, default is 100

The number of hexagons in the \( x \)-direction, default is 100. The corresponding number of hexagons in the \( y \)-direction is chosen such that the hexagons are approximately regular. Alternatively, \text{gridsize} can be a tuple with two elements specifying the number of hexagons in the \( x \)-direction and the \( y \)-direction.

- \text{bins} : {'log'} or int or sequence, optional, default is \text{None}

If \text{None}, no binning is applied; the color of each hexagon directly corresponds to its count value.

If ‘log’, use a logarithmic scale for the color map. Internally, \( \log_{10}(i + 1) \) is used to determine the hexagon color.

If an integer, divide the counts in the specified number of bins, and color the hexagons accordingly.
If a sequence of values, the values of the lower bound of the bins to be used.

**xscale**: {‘linear’, ‘log’}, optional, default is ‘linear’

Use a linear or log10 scale on the horizontal axis.

**yscale**: {‘linear’, ‘log’}, optional, default is ‘linear’

Use a linear or log10 scale on the vertical axis.

**mincnt**: int > 0, optional, default is None

If not None, only display cells with more than mincnt number of points in the cell

**marginals**: bool, optional, default is False

if marginals is True, plot the marginal density as colormapped rectangles along the bottom of the x-axis and left of the y-axis

**extent**: scalar, optional, default is None

The limits of the bins. The default assigns the limits based on gridsize, x, y, xscale and yscale.

If xscale or yscale is set to ‘log’, the limits are expected to be the exponent for a power of 10. E.g. for x-limits of 1 and 50 in ‘linear’ scale and y-limits of 10 and 1000 in ‘log’ scale, enter (1, 50, 1, 3).

Order of scalars is (left, right, bottom, top).

**Returns** object

a *PolyCollection* instance; use *get_array()* on this *PolyCollection* to get the counts in each hexagon.

If marginals is True, horizontal bar and vertical bar (both PolyCollections) will be attached to the return collection as attributes hbar and vbar.

**Other Parameters** *cmap*: object, optional, default is None

a *matplotlib.colors.Colormap* instance. If None, defaults to rc image.cmap.

**norm**: object, optional, default is None

*matplotlib.colors.Normalize* instance is used to scale luminance data to 0,1.

**vmin, vmax**: scalar, optional, default is None

vmin and vmax are used in conjunction with norm to normalize luminance data. If None, the min and max of the color array C are used. Note if you pass a norm instance your settings for vmin and vmax will be ignored.

**alpha**: scalar between 0 and 1, optional, default is None

the alpha value for the patches

**linewidths**: scalar, optional, default is None
If `None`, defaults to 1.0.

**edgecolors**: {'none'} or mpl color, optional, default is ‘none’

If ‘none’, draws the edges in the same color as the fill color. This is the default, as it avoids unsightly unpainted pixels between the hexagons.

If `None`, draws outlines in the default color.

If a matplotlib color arg, draws outlines in the specified color.

**Notes**

The standard descriptions of all the `Collection` parameters:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>agg_filter</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>alpha</code></td>
<td>float or <code>None</code></td>
</tr>
<tr>
<td><code>animated</code></td>
<td>`[True</td>
</tr>
<tr>
<td><code>antialiased</code></td>
<td>Boolean or sequence of booleans</td>
</tr>
<tr>
<td><code>array</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>axes</code></td>
<td>an <code>Axes</code> instance</td>
</tr>
<tr>
<td><code>clim</code></td>
<td>a length 2 sequence of floats</td>
</tr>
<tr>
<td><code>clip_box</code></td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td><code>clip_on</code></td>
<td>`[True</td>
</tr>
<tr>
<td><code>clip_path</code></td>
<td>`[ (Path, Transform)</td>
</tr>
<tr>
<td><code>cmap</code></td>
<td>a colormap or registered colormap name</td>
</tr>
<tr>
<td><code>color</code></td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td><code>contains</code></td>
<td>a callable function</td>
</tr>
<tr>
<td><code>edgecolor</code></td>
<td>matplotlib color spec or sequence of specs</td>
</tr>
<tr>
<td><code>facecolor</code></td>
<td>matplotlib color spec or sequence of specs</td>
</tr>
<tr>
<td><code>figure</code></td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td><code>gid</code></td>
<td>an id string</td>
</tr>
<tr>
<td><code>hatch</code></td>
<td>`[ ‘</td>
</tr>
<tr>
<td><code>label</code></td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td><code>linestyle</code></td>
<td>`‘solid’</td>
</tr>
<tr>
<td><code>linewidth</code></td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td><code>norm</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>offset_position</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>offsets</code></td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td><code>path_effects</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>picker</code></td>
<td>`[None</td>
</tr>
<tr>
<td><code>pickradius</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>rasterized</code></td>
<td>`[True</td>
</tr>
<tr>
<td><code>sketch_params</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>snap</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>transform</code></td>
<td><code>Transform</code> instance</td>
</tr>
</tbody>
</table>
Table 45.18 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>urls</td>
<td>unknown</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

**Note:** In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]:**

- All arguments with the following names: ‘x’, ‘y’.

**Examples**

```python
import matplotlib.pyplot as plt

# Hexagon binning
plt.hist(x, bins=None, range=None, normed=False, weights=None, cumulative=False, bottom=None, histtype='bar', align='mid', orientation='vertical', rwidth=None, log=False, color=None, label=None, stacked=False, hold=None, data=None, **kwargs)

# With a log color scale
```

Plot a histogram.

Compute and draw the histogram of `x`. The return value is a tuple `(n, bins, patches)` or `([n0, n1, ...], bins, [patches0, patches1,...])` if the input contains multiple data.

Multiple data can be provided via `x` as a list of datasets of potentially different length (`[x0, x1, ...]`), or as a 2-D ndarray in which each column is a dataset. Note that the ndarray form is transposed relative to the input.
to the list form.

Masked arrays are not supported at present.

**Parameters**

- **x**: (n,) array or sequence of (n,) arrays
  
  Input values, this takes either a single array or a sequence of arrays which are not required to be of the same length

- **bins**: integer or array_like or ‘auto’, optional
  
  If an integer is given, bins + 1 bin edges are returned, consistently with `numpy.histogram()` for numpy version >= 1.3.
  
  Unequally spaced bins are supported if bins is a sequence.
  
  If Numpy 1.11 is installed, may also be 'auto'.
  
  Default is taken from the rcParam `hist.bins`.

- **range**: tuple or None, optional
  
  The lower and upper range of the bins. Lower and upper outliers are ignored.
  
  If not provided, range is (x.min(), x.max()). Range has no effect if bins is a sequence.
  
  If bins is a sequence or range is specified, autoscaling is based on the specified bin range instead of the range of x.
  
  Default is None

- **normed**: boolean, optional
  
  If True, the first element of the return tuple will be the counts normalized to form a probability density, i.e., \(\frac{n}{\text{len}(x) \cdot \text{dbin}}\), i.e., the integral of the histogram will sum to 1. If stacked is also True, the sum of the histograms is normalized to 1.
  
  Default is False

- **weights**: (n,) array_like or None, optional
  
  An array of weights, of the same shape as x. Each value in x only contributes its associated weight towards the bin count (instead of 1). If normed is True, the weights are normalized, so that the integral of the density over the range remains 1.
  
  Default is None

- **cumulative**: boolean, optional
  
  If True, then a histogram is computed where each bin gives the counts in that bin plus all bins for smaller values. The last bin gives the total number of datapoints. If normed is also True then the histogram is normalized such that the last bin equals 1. If cumulative evaluates to less than 0 (e.g., -1), the direction of accumulation is reversed. In this case, if normed is also True, then the histogram is normalized such that the first bin equals 1.
Default is False

**bottom**: array_like, scalar, or None

Location of the bottom baseline of each bin. If a scalar, the base line for each bin is shifted by the same amount. If an array, each bin is shifted independently and the length of bottom must match the number of bins. If None, defaults to 0.

Default is None

**histtype**: {'bar', 'barstacked', 'step', 'stepfilled'}, optional

The type of histogram to draw.
- 'bar' is a traditional bar-type histogram. If multiple data are given the bars are arranged side by side.
- 'barstacked' is a bar-type histogram where multiple data are stacked on top of each other.
- 'step' generates a lineplot that is by default unfilled.
- 'stepfilled' generates a lineplot that is by default filled.

Default is 'bar'

**align**: {'left', 'mid', 'right'}, optional

Controls how the histogram is plotted.
- 'left': bars are centered on the left bin edges.
- 'mid': bars are centered between the bin edges.
- 'right': bars are centered on the right bin edges.

Default is 'mid'

**orientation**: {'horizontal', 'vertical'}, optional

If 'horizontal', *barh* will be used for bar-type histograms and the *bottom* kwarg will be the left edges.

**rwidth**: scalar or None, optional

The relative width of the bars as a fraction of the bin width. If None, automatically compute the width.

Ignored if *histtype* is 'step' or 'stepfilled'.

Default is None

**log**: boolean, optional

If True, the histogram axis will be set to a log scale. If log is True and x is a 1D array, empty bins will be filtered out and only the non-empty (n, bins, patches) will be returned.

Default is False
MATPLOTLIB, RELEASE 2.0.2

**color**: color or array_like of colors or None, optional

Color spec or sequence of color specs, one per dataset. Default (None) uses the standard line color sequence.

Default is None

**label**: string or None, optional

String, or sequence of strings to match multiple datasets. Bar charts yield multiple patches per dataset, but only the first gets the label, so that the legend command will work as expected.

Default is None

**stacked**: boolean, optional

If True, multiple data are stacked on top of each other. If False, multiple data are arranged side by side if histtype is ‘bar’ or on top of each other if histtype is ‘step’

Default is False

**Returns**

- **n** : array or list of arrays

  The values of the histogram bins. See **normed** and **weights** for a description of the possible semantics. If input x is an array, then this is an array of length nbins. If input is a sequence arrays [data1, data2, ...], then this is a list of arrays with the values of the histograms for each of the arrays in the same order.

- **bins** : array

  The edges of the bins. Length nbins + 1 (nbins left edges and right edge of last bin). Always a single array even when multiple data sets are passed in.

- **patches** : list or list of lists

  Silent list of individual patches used to create the histogram or list of such list if multiple input datasets.

**Other Parameters**

- **kwargs**: **Patch** properties

**See also**:

- **hist2d** 2D histograms

**Notes**

Until numpy release 1.5, the underlying numpy histogram function was incorrect with normed=`True` if bin sizes were unequal. MPL inherited that error. It is now corrected within MPL when using earlier numpy versions.
Examples

Histogram of IQ: $\mu = 100, \sigma = 15$

Note: In addition to the above described arguments, this function can take a data keyword argument. If such a data argument is given, the following arguments are replaced by data[<arg>]:

• All arguments with the following names: ‘weights’, ‘x’.

```
matplotlib.pyplot.hist2d(x, y, bins=10, range=None, normed=False, weights=None,
                          cmin=None, cmax=None, hold=None, data=None, **kwargs)
```

Make a 2D histogram plot.

**Parameters x, y: array_like, shape (n, )**

Input values

**bins: [None | int | [int, int] | array_like | [array, array]]**

The bin specification:

• If int, the number of bins for the two dimensions (nx=ny=bins).

• If [int, int], the number of bins in each dimension (nx, ny = bins).

• If array_like, the bin edges for the two dimensions (x_edges=y_edges=bins).
If [array, array], the bin edges in each dimension (x_edges, y_edges = bins).

The default value is 10.

**range** : array_like shape(2, 2), optional, default: None

The leftmost and rightmost edges of the bins along each dimension (if not specified explicitly in the bins parameters): [[xmin, xmax], [ymin, ymax]]. All values outside of this range will be considered outliers and not tallied in the histogram.

**normed** : boolean, optional, default: False

Normalize histogram.

**weights** : array_like, shape (n,), optional, default: None

An array of values w_i weighing each sample (x_i, y_i).

**cmin** : scalar, optional, default: None

All bins that have count less than cmin will not be displayed and these count values in the return value count histogram will also be set to nan upon return.

**cmax** : scalar, optional, default: None

All bins that have count more than cmax will not be displayed (set to none before passing to imshow) and these count values in the return value count histogram will also be set to nan upon return.

**Returns**
The return value is (counts, xedges, yedges, Image).

**Other Parameters**

**cmap** : {Colormap, string}, optional


**norm** : Normalize, optional

A *matplotlib.colors.Normalize* instance is used to scale luminance data to [0, 1]. If not set, defaults to Normalize().

**vmin/vmax** : {None, scalar}, optional

Arguments passed to the Normalize instance.

**alpha** : 0 <= scalar <= 1 or None, optional

The alpha blending value.

**See also:**

**hist** 1D histogram
Notes

Rendering the histogram with a logarithmic color scale is accomplished by passing a colors.LogNorm instance to the norm keyword argument. Likewise, power-law normalization (similar in effect to gamma correction) can be accomplished with colors.PowerNorm.

Examples

![Histogram Example](image)

**Note:** In addition to the above described arguments, this function can take a data keyword argument. If such a data argument is given, the following arguments are replaced by data[arg]:

- All arguments with the following names: ‘weights’, ‘x’, ‘y’.

```python
matplotlib.pyplot.hlines(y, xmin, xmax, colors='k', linestyles='solid', label='', hold=None, data=None, **kwargs)
```

Plot horizontal lines at each y from xmin to xmax.

**Parameters**

- **y**: scalar or sequence of scalar
  y-indexes where to plot the lines.
- **xmin, xmax**: scalar or 1D array_like
Respective beginning and end of each line. If scalars are provided, all lines will have same length.

**colors** : array_like of colors, optional, default: ‘k’

**linestyles** : [‘solid’ | ‘dashed’ | ‘dashdot’ | ‘dotted’], optional

**label** : string, optional, default: ‘’

**Returns** lines : LineCollection

**Other Parameters** **kwargs** : LineCollection properties.

**See also:**

`vlines` vertical lines

**Examples**

![Vertical lines demo](image1)

![Horizontal lines demo](image2)

**Note:** In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All arguments with the following names: ‘xmax’, ‘xmin’, ‘y’.

**matplotlib.pyplot.hold**(b=None)

Deprecated since version 2.0: pyplot.hold is deprecated. Future behavior will be consistent with the long-time default: plot commands add elements without first clearing the Axes and/or Figure.

Set the hold state. If `b` is None (default), toggle the hold state, else set the hold state to boolean value `b`:
### hold()

# toggle hold
hold()

# hold is on
hold(True)

# hold is off
hold(False)

When `hold` is `True`, subsequent plot commands will add elements to the current axes. When `hold` is `False`, the current axes and figure will be cleared on the next plot command.

```python
matplotlib.pyplot.hot()
```

set the default colormap to hot and apply to current image if any. See `help(colormaps)` for more information

```python
matplotlib.pyplot.hsv()
```

set the default colormap to hsv and apply to current image if any. See `help(colormaps)` for more information

```python
matplotlib.pyplot.imread(*args, **kwargs)
```

Read an image from a file into an array.

`fname` may be a string path, a valid URL, or a Python file-like object. If using a file object, it must be opened in binary mode.

If `format` is provided, will try to read file of that type, otherwise the format is deduced from the filename. If nothing can be deduced, PNG is tried.

Return value is a `numpy.array`. For grayscale images, the return array is MxN. For RGB images, the return value is MxNx3. For RGBA images the return value is MxNx4.

Matplotlib can only read PNGs natively, but if PIL is installed, it will use it to load the image and return an array (if possible) which can be used with `imshow()`. Note, URL strings may not be compatible with PIL. Check the PIL documentation for more information.

```python
matplotlib.pyplot.imsave(*args, **kwargs)
```

Save an array as an image file.

The output formats available depend on the backend being used.

#### Arguments:

- **fname**: A string containing a path to a filename, or a Python file-like object. If `format` is `None` and `fname` is a string, the output format is deduced from the extension of the filename.

- **arr**: An MxN (luminance), MxNx3 (RGB) or MxNx4 (RGBA) array.

#### Keyword arguments:

- **vmin/vmax**: `[ None | scalar ]` `vmin` and `vmax` set the color scaling for the image by fixing the values that map to the colormap color limits. If either `vmin` or `vmax` is `None`, that limit is determined from the `arr` `min/max` value.

- **cmap**: `cmap` is a `colors.Colormap` instance, e.g., `cm.jet`. If `None`, default to the `rc.image.cmap` value.

- **format**: One of the file extensions supported by the active backend. Most backends support `png`, `pdf`, `ps`, `eps` and `svg`.  

---

45.1. matplotlib.pyplot  1499
**origin** [‘upper’ | ‘lower’] Indicates where the [0,0] index of the array is in the upper left or lower left corner of the axes. Defaults to the rc image.origin value.

**dpi** The DPI to store in the metadata of the file. This does not affect the resolution of the output image.

```python
matplotlib.pyplot.imshow(X, cmap=None, norm=None, aspect=None, interpolation=None, alpha=None, vmin=None, vmax=None, origin=None, extent=None, shape=None, filternorm=1, filterrad=4.0, imlim=None, resample=None, url=None, hold=None, data=None, **kwargs)
```

Display an image on the axes.

**Parameters**

**X** : array_like, shape (n, m) or (n, m, 3) or (n, m, 4)

Display the image in X to current axes. X may be an array or a PIL image. If X is an array, it can have the following shapes and types:

- MxN – values to be mapped (float or int)
- MxNx3 – RGB (float or uint8)
- MxNx4 – RGBA (float or uint8)

The value for each component of MxNx3 and MxNx4 float arrays should be in the range 0.0 to 1.0. MxN arrays are mapped to colors based on the norm (mapping scalar to scalar) and the cmap (mapping the normed scalar to a color).

**cmap** : `Colormap`, optional, default: None

If None, default to rc `image.cmap` value. cmap is ignored if X is 3-D, directly specifying RGB(A) values.

**aspect** : [‘auto’ | ‘equal’ | scalar], optional, default: None

If ‘auto’, changes the image aspect ratio to match that of the axes.

If ‘equal’, and extent is None, changes the axes aspect ratio to match that of the image. If extent is not None, the axes aspect ratio is changed to match that of the extent.

If None, default to rc `image.aspect` value.

**interpolation** : string, optional, default: None


If interpolation is None, default to rc `image.interpolation`. See also the filternorm and filterrad parameters. If interpolation is ‘none’, then no interpolation is performed on the Agg, ps and pdf backends. Other backends will fall back to ‘nearest’.

**norm** : `Normalize`, optional, default: None
A `Normalize` instance is used to scale a 2-D float `X` input to the (0, 1) range for input to the `cmap`. If `norm` is None, use the default function `normalize`. If `norm` is an instance of `NoNorm`, `X` must be an array of integers that index directly into the lookup table of the `cmap`.

**vmin, vmax**: scalar, optional, default: None

- `vmin` and `vmax` are used in conjunction with `norm` to normalize luminance data. Note if you pass a `norm` instance, your settings for `vmin` and `vmax` will be ignored.

**alpha**: scalar, optional, default: None

- The alpha blending value, between 0 (transparent) and 1 (opaque)

**origin**: `[‘upper’ | ‘lower’], optional, default: None

- Place the [0,0] index of the array in the upper left or lower left corner of the axes. If None, default to rc `image.origin`.

**extent**: scalars (left, right, bottom, top), optional, default: None

- The location, in data-coordinates, of the lower-left and upper-right corners. If None, the image is positioned such that the pixel centers fall on zero-based (row, column) indices.

**shape**: scalars (columns, rows), optional, default: None

- For raw buffer images

**filternorm**: scalar, optional, default: 1

- A parameter for the antigrain image resize filter. From the antigrain documentation, if `filternorm = 1`, the filter normalizes integer values and corrects the rounding errors. It doesn’t do anything with the source floating point values, it corrects only integers according to the rule of 1.0 which means that any sum of pixel weights must be equal to 1.0. So, the filter function must produce a graph of the proper shape.

**filterrad**: scalar, optional, default: 4.0

- The filter radius for filters that have a radius parameter, i.e. when interpolation is one of: ‘sinc’, ‘lanczos’ or ‘blackman’

**Returns image**: `AxesImage`

**Other Parameters kwarg**: `Artist` properties.

**See also**:

- `matshow` Plot a matrix or an array as an image.
Notes

Unless `extent` is used, pixel centers will be located at integer coordinates. In other words: the origin will coincide with the center of pixel (0, 0).

Examples

```
3
 2
 1
 0 1 2 3
3
2
1
0
1
2
3
```

**Note:** In addition to the above described arguments, this function can take a `data` keyword argument. If such a `data` argument is given, the following arguments are replaced by `data[<arg>]`:

- All positional and all keyword arguments.

```python
import matplotlib.pyplot as plt

# Set the default colormap to inferno and apply to current image if any.
plt.inferno()

# Install a repl display hook so that any stale figure are automatically redrawn when control is returned to the repl.
plt.install_repl_displayhook()
```

This works with IPython terminals and kernels, as well as vanilla python shells.
matplotlib.pyplot.ioff()
    Turn interactive mode off.

matplotlib.pyplot.ion()
    Turn interactive mode on.

matplotlib.pyplot.ishold()
    Deprecated since version 2.0: pyplot.hold is deprecated. Future behavior will be consistent with the
    long-time default: plot commands add elements without first clearing the Axes and/or Figure.

    Return the hold status of the current axes.

matplotlib.pyplot.isinteractive()
    Return status of interactive mode.

matplotlib.pyplot.jet()
    set the default colormap to jet and apply to current image if any. See help(colormaps) for more
    information

matplotlib.pyplot.legend(*args, **kwargs)
    Places a legend on the axes.

    To make a legend for lines which already exist on the axes (via plot for instance), simply call this
    function with an iterable of strings, one for each legend item. For example:

    ```python
    ax.plot([1, 2, 3])
    ax.legend(['A simple line'])
    ```

    However, in order to keep the “label” and the legend element instance together, it is preferable to
    specify the label either at artist creation, or by calling the set_label() method on the artist:

    ```python
    line, = ax.plot([1, 2, 3], label='Inline label')
    # Overwrite the label by calling the method.
    line.set_label('Label via method')
    ax.legend()
    ```

    Specific lines can be excluded from the automatic legend element selection by defining a label starting
    with an underscore. This is default for all artists, so calling legend() without any arguments and
    without setting the labels manually will result in no legend being drawn.

    For full control of which artists have a legend entry, it is possible to pass an iterable of legend artists
    followed by an iterable of legend labels respectively:

    ```python
    legend((line1, line2, line3), ('label1', 'label2', 'label3'))
    ```

    **Parameters**

    - **loc**: int or string or pair of floats, default: ‘upper right’

    The location of the legend. Possible codes are:

    - ‘best’
    - ‘lower center’
    - ‘lower left’
    - ‘lower right’
    - ‘lower outside’
    - ‘outside’
    - ‘tight’
    - ‘upper center’
    - ‘upper left’
    - ‘upper right’
    - ‘upper outside’
    - ‘upper right’
    - ‘center’
<table>
<thead>
<tr>
<th>Location String</th>
<th>Location Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘best’</td>
<td>0</td>
</tr>
<tr>
<td>‘upper right’</td>
<td>1</td>
</tr>
<tr>
<td>‘upper left’</td>
<td>2</td>
</tr>
<tr>
<td>‘lower left’</td>
<td>3</td>
</tr>
<tr>
<td>‘lower right’</td>
<td>4</td>
</tr>
<tr>
<td>‘right’</td>
<td>5</td>
</tr>
<tr>
<td>‘center left’</td>
<td>6</td>
</tr>
<tr>
<td>‘center right’</td>
<td>7</td>
</tr>
<tr>
<td>‘lower center’</td>
<td>8</td>
</tr>
<tr>
<td>‘upper center’</td>
<td>9</td>
</tr>
<tr>
<td>‘center’</td>
<td>10</td>
</tr>
</tbody>
</table>

Alternatively can be a 2-tuple giving \( x, y \) of the lower-left corner of the legend in axes coordinates (in which case \( bbox\_to\_anchor \) will be ignored).

**bbox\_to\_anchor**: `matplotlib.transforms.BboxBase` instance or tuple of floats

Specify any arbitrary location for the legend in \( bbox\_transform \) coordinates (default Axes coordinates).

For example, to put the legend’s upper right hand corner in the center of the axes the following keywords can be used:

```python
loc='upper right', bbox_to_anchor=(0.5, 0.5)
```

**ncol**: integer

The number of columns that the legend has. Default is 1.

**prop**: None or `matplotlib.font_manager.FontProperties` or dict

The font properties of the legend. If None (default), the current `matplotlib.rcParams` will be used.

**fontsize**: int or float or {'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large'}

Controls the font size of the legend. If the value is numeric the size will be the absolute font size in points. String values are relative to the current default font size. This argument is only used if **prop** is not specified.

**numpoints**: None or int

The number of marker points in the legend when creating a legend entry for a `line/matplotlib.lines.Line2D`. Default is None which will take the value from the `legend.numpoints` rcParam.

**scatterpoints**: None or int

The number of marker points in the legend when creating a legend entry for a scatter plot/`matplotlib.collections.PathCollection`. Default is None which will take the value from the `legend.scatterpoints` rcParam.

**scatteryoffsets**: iterable of floats
The vertical offset (relative to the font size) for the markers created for a scatter plot legend entry. 0.0 is at the base the legend text, and 1.0 is at the top. To draw all markers at the same height, set to [0.5]. Default [0.375, 0.5, 0.3125].

**markerscale**: None or int or float

The relative size of legend markers compared with the originally drawn ones. Default is None which will take the value from the `legend.markerscale` rcParam.

**markerfirst**: bool

If True, legend marker is placed to the left of the legend label if False, legend marker is placed to the right of the legend label

**frameon**: None or bool

Control whether the legend should be drawn on a patch (frame). Default is None which will take the value from the `legend.frameon` rcParam.

**fancybox**: None or bool

Control whether round edges should be enabled around the `FancyBboxPatch` which makes up the legend’s background. Default is None which will take the value from the `legend.fancybox` rcParam.

**shadow**: None or bool

Control whether to draw a shadow behind the legend. Default is None which will take the value from the `legend.shadow` rcParam.

**framealpha**: None or float

Control the alpha transparency of the legend’s background. Default is None which will take the value from the `legend.framealpha` rcParam.

**facecolor**: None or “inherit” or a color spec

Control the legend’s background color. Default is None which will take the value from the `legend.facecolor` rcParam. If "inherit", it will take the axes.facecolor rcParam.

**edgecolor**: None or “inherit” or a color spec

Control the legend’s background patch edge color. Default is None which will take the value from the `legend.edgecolor` rcParam. If "inherit", it will take the axes.edgecolor rcParam.

**mode**: {“expand”, None}

If mode is set to "expand" the legend will be horizontally expanded to fill the axes area (or bbox_to_anchor if defines the legend’s size).

**bbox_transform**: None or `matplotlib.transforms.Transform`

The transform for the bounding box (bbox_to_anchor). For a value of None (default) the Axes’ transAxes transform will be used.
**title** : str or None

The legend’s title. Default is no title (None).

**borderpad** : float or None

The fractional whitespace inside the legend border. Measured in font-size units. Default is None which will take the value from the `legend.borderpad` `rcParam`.

**labelspacing** : float or None

The vertical space between the legend entries. Measured in font-size units. Default is None which will take the value from the `legend.labelspacing` `rcParam`.

**handlelength** : float or None

The length of the legend handles. Measured in font-size units. Default is None which will take the value from the `legend.handlelength` `rcParam`.

**handletextpad** : float or None

The pad between the legend handle and text. Measured in font-size units. Default is None which will take the value from the `legend.handletextpad` `rcParam`.

**borderaxespad** : float or None

The pad between the axes and legend border. Measured in font-size units. Default is None which will take the value from the `legend.borderaxespad` `rcParam`.

**columnspacing** : float or None

The spacing between columns. Measured in font-size units. Default is None which will take the value from the `legend.columnspacing` `rcParam`.

**handler_map** : dict or None

The custom dictionary mapping instances or types to a legend handler. This `handler_map` updates the default handler map found at `matplotlib.legend.Legend.get_legend_handler_map()`.

**Notes**

Not all kinds of artist are supported by the legend command. See *Legend guide* for details.
Matplotlib.pyplot.locator_params(axis='both', tight=None, **kwargs)

Control behavior of tick locators.

Keyword arguments:

axis ['x' | 'y' | 'both'] Axis on which to operate; default is ‘both’.

tight [True | False | None] Parameter passed to autoscale_view(). Default is None, for no change.

Remaining keyword arguments are passed to directly to the set_params() method.

Typically one might want to reduce the maximum number of ticks and use tight bounds when plotting small subplots, for example:

```
ax.locator_params(tight=True, nbins=4)
```

Because the locator is involved in autoscaling, autoscale_view() is called automatically after the parameters are changed.

This presently works only for the MaxNLocator used by default on linear axes, but it may be generalized.

Matplotlib.pyplot.loglog(*args, **kwargs)

Make a plot with log scaling on both the x and y axis.
loglog() supports all the keyword arguments of plot() and matplotlib.axes.Axes.set_xscale() / matplotlib.axes.Axes.set_yscale().

Notable keyword arguments:

- **basex/basey**: scalar > 1 Base of the x/y logarithm
- **subsx/subsy**: [ None | sequence ] The location of the minor x/y ticks; None defaults to autosubs, which depend on the number of decades in the plot; see matplotlib.axes.Axes.set_xscale() / matplotlib.axes.Axes.set_yscale() for details
- **nonposx/nonposy**: ['mask' | 'clip'] Non-positive values in x or y can be masked as invalid, or clipped to a very small positive number

The remaining valid kwarg is Line2D properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>drawstyle</td>
<td>['default'</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>['full'</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['solid'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>marker</td>
<td>A valid marker style</td>
</tr>
<tr>
<td>markeredgecolor or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>[None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
</tbody>
</table>
Table 45.19 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

Example:

```
matplotlib.pyplot.magma()
```

set the default colormap to magma and apply to current image if any. See help(colormaps) for more information

```
matplotlib.pyplot.magnitude_spectrum(x, Fs=None, Fc=None, window=None, pad_to=None, sides=None, scale=None, hold=None, data=None, **kwargs)
```
Plot the magnitude spectrum.

Call signature:

```
magnitude_spectrum(x, Fs=2, Fc=0, window=mlab.window_hanning,
                  pad_to=None, sides='default', **kwargs)
```

Compute the magnitude spectrum of `x`. Data is padded to a length of `pad_to` and the windowing function `window` is applied to the signal.

**Parameters**

- **x**: 1-D array or sequence
  - Array or sequence containing the data
- **Fs**: scalar
  - The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, `freqs`, in cycles per time unit. The default value is 2.
- **window**: callable or ndarray
  - A function or a vector of length `NFFT`. To create window vectors see `window_hanning()`, `window_none()`, `numpy.blackman()`, `numpy.hanning()`, `numpy.bartlett()`, `scipy.signal()`, `scipy.signal.get_window()`, etc. The default is `window_hanning()`. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.
- **sides**: [‘default’ | ‘onesided’ | ‘twosided’]
  - Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. ‘onesided’ forces the return of a one-sided spectrum, while ‘twosided’ forces two-sided.
- **pad_to**: integer
  - The number of points to which the data segment is padded when performing the FFT. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the `n` parameter in the call to `fft()`. The default is `None`, which sets `pad_to` equal to the length of the input signal (i.e. no padding).
- **scale**: [‘default’ | ‘linear’ | ‘dB’]
  - The scaling of the values in the `spec`. ‘linear’ is no scaling. ‘dB’ returns the values in dB scale. When `mode` is ‘density’, this is dB power (`10 * log10`). Otherwise this is dB amplitude (`20 * log10`). ‘default’ is ‘linear’.
- **Fc**: integer
  - The center frequency of `x` (defaults to 0), which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.
Keyword arguments control the `Line2D` properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>agg_filter</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>alpha</code></td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td><code>animated</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>antialiased</code> or <code>aa</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>axes</code></td>
<td>an <code>Axes</code> instance</td>
</tr>
<tr>
<td><code>clip_box</code></td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td><code>clip_on</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>clip_path</code></td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td><code>color</code> or <code>c</code></td>
<td>any matplotlib color</td>
</tr>
<tr>
<td><code>contains</code></td>
<td>a callable function</td>
</tr>
<tr>
<td><code>dash_capstyle</code></td>
<td>[‘butt’</td>
</tr>
<tr>
<td><code>dash_joinstyle</code></td>
<td>[‘miter’</td>
</tr>
<tr>
<td><code>dashes</code></td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td><code>drawstyle</code></td>
<td>[‘default’</td>
</tr>
<tr>
<td><code>figure</code></td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td><code>fillstyle</code></td>
<td>[‘full’</td>
</tr>
<tr>
<td><code>gid</code></td>
<td>an id string</td>
</tr>
<tr>
<td><code>label</code></td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td><code>linestyle</code> or <code>ls</code></td>
<td>[‘solid’</td>
</tr>
<tr>
<td><code>linewidth</code> or <code>lw</code></td>
<td>float value in points</td>
</tr>
<tr>
<td><code>marker</code></td>
<td>A valid marker style</td>
</tr>
<tr>
<td><code>markeredgecolor</code> or <code>mec</code></td>
<td>any matplotlib color</td>
</tr>
<tr>
<td><code>markeredgewidth</code> or <code>mew</code></td>
<td>float value in points</td>
</tr>
<tr>
<td><code>markerfacecolor</code> or <code>mfc</code></td>
<td>any matplotlib color</td>
</tr>
<tr>
<td><code>markerfacecoloralt</code> or <code>mfcalt</code></td>
<td>any matplotlib color</td>
</tr>
<tr>
<td><code>markersize</code> or <code>ms</code></td>
<td>float</td>
</tr>
<tr>
<td><code>markevery</code></td>
<td>[None</td>
</tr>
<tr>
<td><code>path_effects</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>picker</code></td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td><code>pickradius</code></td>
<td>float distance in points</td>
</tr>
<tr>
<td><code>rasterized</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>sketch_params</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>snap</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>solid_capstyle</code></td>
<td>[‘butt’</td>
</tr>
<tr>
<td><code>solid_joinstyle</code></td>
<td>[‘miter’</td>
</tr>
<tr>
<td><code>transform</code></td>
<td>a <code>matplotlib.transforms.Transform</code> instance</td>
</tr>
<tr>
<td><code>url</code></td>
<td>a url string</td>
</tr>
<tr>
<td><code>visible</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>xdata</code></td>
<td>1D array</td>
</tr>
<tr>
<td><code>ydata</code></td>
<td>1D array</td>
</tr>
<tr>
<td><code>zorder</code></td>
<td>any number</td>
</tr>
</tbody>
</table>
Returns spectrum : 1-D array

The values for the magnitude spectrum before scaling (real valued)

freqs : 1-D array

The frequencies corresponding to the elements in spectrum

line : a Line2D instance

The line created by this function

See also:

psd() plots the power spectral density.

angle_spectrum() plots the angles of the corresponding frequencies.

phase_spectrum() plots the phase (unwrapped angle) of the corresponding frequencies.

specgram() can plot the magnitude spectrum of segments within the signal in a colormap.

In addition to the above described arguments, this function can take a data keyword argument. If such a data argument is given, the following arguments are replaced by data[arg]: * All arguments with the following names: ‘x’.
Examples

```python
matplotlib.pyplot.margins(*args, **kw)

Set or retrieve autoscaling margins.

signatures:

margins()
returns xmargin, ymargin

margins(margin)
margins(xmargin, ymargin)
margins(x=xmargin, y=ymargin)
margins(..., tight=False)

All three forms above set the xmargin and ymargin parameters. All keyword parameters are optional. A single argument specifies both xmargin and ymargin. The tight parameter is passed to autoscale_view(), which is executed after a margin is changed; the default here is True, on the assumption that when margins are specified, no additional padding to match tick marks is usually desired. Setting tight to None will preserve the previous setting.
```
Specifying any margin changes only the autoscaling: for example, if xmargin is not None, then xmargin times the X data interval will be added to each end of that interval before it is used in autoscaling.

```python
matplotlib.pyplot.matshow(A, fignum=None, **kw)
```

Display an array as a matrix in a new figure window.

The origin is set at the upper left hand corner and rows (first dimension of the array) are displayed horizontally. The aspect ratio of the figure window is that of the array, unless this would make an excessively short or narrow figure.

Tick labels for the xaxis are placed on top.

With the exception of fignum, keyword arguments are passed to imshow(). You may set the origin kwarg to “lower” if you want the first row in the array to be at the bottom instead of the top.

**fignum:** [ None | integer | False ] By default, matshow() creates a new figure window with automatic numbering. If fignum is given as an integer, the created figure will use this figure number. Because of how matshow() tries to set the figure aspect ratio to be the one of the array, if you provide the number of an already existing figure, strange things may happen.

If fignum is False or 0, a new figure window will **NOT** be created.

```python
matplotlib.pyplot.minorticks_off()
```

Remove minor ticks from the current plot.

```python
matplotlib.pyplot.minorticks_on()
```

Display minor ticks on the current plot.

Displaying minor ticks reduces performance; turn them off using minorticks_off() if drawing speed is a problem.

```python
matplotlib.pyplot.nipy_spectral()
```

set the default colormap to nipy_spectral and apply to current image if any. See help(colormaps) for more information

```python
matplotlib.pyplot.over(func, *args, **kwargs)
```

Deprecated since version 2.0: pyplot.hold is deprecated. Future behavior will be consistent with the long-time default: plot commands add elements without first clearing the Axes and/or Figure.

Call a function with hold(True).

Calls:

```python
func(*args, **kwargs)
```

with hold(True) and then restores the hold state.

```python
matplotlib.pyplot.pause(interval)
```

Pause for interval seconds.

If there is an active figure it will be updated and displayed, and the GUI event loop will run during the pause.

If there is no active figure, or if a non-interactive backend is in use, this executes time.sleep(interval).

This can be used for crude animation. For more complex animation, see matplotlib.animation.
This function is experimental; its behavior may be changed or extended in a future release.

```python
matplotlib.pyplot.pcolor(*args, **kwargs)
```

Create a pseudocolor plot of a 2-D array.

**Note:** pcolor can be very slow for large arrays; consider using the similar but much faster `pcolormesh()` instead.

Call signatures:

```python
pcolor(C, **kwargs)
pcolor(X, Y, C, **kwargs)
```

*C* is the array of color values.

*X* and *Y*, if given, specify the (*x*, *y*) coordinates of the colored quadrilaterals; the quadrilateral for *C*[i,j] has corners at:

```python
(X[i, j], Y[i, j]),
(X[i, j+1], Y[i, j+1]),
(X[i+1, j], Y[i+1, j]),
(X[i+1, j+1], Y[i+1, j+1]).
```

Ideally the dimensions of *X* and *Y* should be one greater than those of *C*; if the dimensions are the same, then the last row and column of *C* will be ignored.

Note that the column index corresponds to the *x*-coordinate, and the row index corresponds to *y*; for details, see the Grid Orientation section below.

If either or both of *X* and *Y* are 1-D arrays or column vectors, they will be expanded as needed into the appropriate 2-D arrays, making a rectangular grid.

*X*, *Y* and *C* may be masked arrays. If either *C*[i, j], or one of the vertices surrounding *C*[i,j] (*X* or *Y* at [i, j], [i+1, j], [i, j+1],[i+1, j+1]) is masked, nothing is plotted.

Keyword arguments:

- **cmap:** [None | Colormap] A matplotlib.colors.Colormap instance. If None, use rc settings.
- **norm:** [None | Normalize] An matplotlib.colors.Normalize instance is used to scale luminance data to 0,1. If None, defaults to normalize().
- **vmin/vmax:** [None | scalar] vmin and vmax are used in conjunction with norm to normalize luminance data. If either is None, it is autoscaled to the respective min or max of the color array *C*. If not None, vmin or vmax passed in here override any pre-existing values supplied in the norm instance.
- **shading:** [‘flat’ | ‘faceted’] If ‘faceted’, a black grid is drawn around each rectangle; if ‘flat’, edges are not drawn. Default is ‘flat’, contrary to MATLAB.

This kwarg is deprecated; please use ‘edgecolors’ instead:

- shading='flat' – edgecolors='none'
• shading='faceted' – edgecolors='k'

**edgecolors:** [None | 'none' | color | color sequence] If None, the rc setting is used by default.
If 'none', edges will not be visible.
An mpl color or sequence of colors will set the edge color

**alpha:** 0 <= scalar <= 1 or None the alpha blending value

**snap:** bool Whether to snap the mesh to pixel boundaries.

Return value is a `matplotlib.collections.Collection` instance. The grid orientation follows the MATLAB convention: an array `C` with shape (nrows, ncolumns) is plotted with the column number as X and the row number as Y, increasing up; hence it is plotted the way the array would be printed, except that the Y axis is reversed. That is, `C` is taken as \( C^\top(y, x) \).

Similarly for `meshgrid()`:

```python
x = np.arange(5)
y = np.arange(3)
X, Y = np.meshgrid(x, y)
```

is equivalent to:

```python
X = np.array([[0, 1, 2, 3, 4],
              [0, 1, 2, 3, 4],
              [0, 1, 2, 3, 4]])
Y = np.array([[0, 0, 0, 0, 0],
              [1, 1, 1, 1, 1],
              [2, 2, 2, 2, 2]])
```

so if you have:

```python
C = rand(len(x), len(y))
```

then you need to transpose `C`:

```python
pcolor(X, Y, C.T)
```
or:

```python
pcolor(C.T)
```

MATLAB `pcolor()` always discards the last row and column of `C`, but matplotlib displays the last row and column if X and Y are not specified, or if X and Y have one more row and column than C.

`**kwargs` can be used to control the `PolyCollection` properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>agg_filter</code></td>
<td>unknown</td>
</tr>
</tbody>
</table>

Continued on next page
### Table 45.21 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or antialiaseds</td>
<td>Boolean or sequence of booleans</td>
</tr>
<tr>
<td>array</td>
<td>unknown</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clim</td>
<td>a length 2 sequence of floats</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>cmap</td>
<td>a colormap or registered colormap name</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or edgecolors</td>
<td>matplotlib color spec or sequence of specs</td>
</tr>
<tr>
<td>facecolor or facecolors</td>
<td>matplotlib color spec or sequence of specs</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[‘/’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or dashes or linestyles</td>
<td>[‘solid’</td>
</tr>
<tr>
<td>linewidth or linewidths or lw</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>norm</td>
<td>unknown</td>
</tr>
<tr>
<td>offset_position</td>
<td>unknown</td>
</tr>
<tr>
<td>offsets</td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>pickradius</td>
<td>unknown</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>urls</td>
<td>unknown</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

**Note:** The default antialiaseds is False if the default edgecolors*=“none” is used. This eliminates artificial lines at patch boundaries, and works regardless of the value of alpha. If *edgecolors is not “none”, then the default antialiaseds is taken from rcParams[‘patch.antialiased’], which defaults to True. Stroking the edges may be preferred if alpha is 1, but will cause artifacts otherwise.

See also:
**pcolormesh()** For an explanation of the differences between pcolor and pcolormesh.

**Note:** In addition to the above described arguments, this function can take a data keyword argument. If such a data argument is given, the following arguments are replaced by data[<arg>]:

- All positional and all keyword arguments.

```
matplotlib.pyplot.pcolormesh(*args, **kwargs)
```

Plot a quadrilateral mesh.

Call signatures:

```
pcolormesh(C)
pcolormesh(X, Y, C)
pcolormesh(C, **kwargs)
```

Create a pseudocolor plot of a 2-D array.

pcolormesh is similar to pcolor(), but uses a different mechanism and returns a different object; pcolor returns a PolyCollection but pcolormesh returns a QuadMesh. It is much faster, so it is almost always preferred for large arrays.

C may be a masked array, but X and Y may not. Masked array support is implemented via cmap and norm; in contrast, pcolor() simply does not draw quadrilaterals with masked colors or vertices.

Keyword arguments:

- **cmap:** [None | Colormap] A matplotlib.colors.Colormap instance. If None, use rc settings.
- **norm:** [None | Normalize] A matplotlib.colors.Normalize instance is used to scale luminance data to 0,1. If None, defaults to normalize().
- **vmin/vmax:** [None | scalar] vmin and vmax are used in conjunction with norm to normalize luminance data. If either is None, it is autoscaled to the respective min or max of the color array C. If not None, vmin or vmax passed in here override any pre-existing values supplied in the norm instance.
- **shading:** [‘flat’ | ‘gouraud’] ‘flat’ indicates a solid color for each quad. When ‘gouraud’, each quad will be Gouraud shaded. When gouraud shading, edgecolors is ignored.
- **edgecolors:** [None | ’None’ | ’face’ | color | color sequence]
  - If None, the rc setting is used by default.
  - If ’None’, edges will not be visible.
  - If ’face’, edges will have the same color as the faces.
  - An mpl color or sequence of colors will set the edge color
- **alpha:** 0 <= scalar <= 1 or None the alpha blending value
Return value is a `matplotlib.collections.QuadMesh` object.

**kwargs can be used to control the `matplotlib.collections.QuadMesh` properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>agg_filter</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>alpha</code></td>
<td>float or None</td>
</tr>
<tr>
<td><code>animated</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>antialiased</code> or <code>antialiaseds</code></td>
<td>Boolean or sequence of booleans</td>
</tr>
<tr>
<td><code>array</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>axes</code></td>
<td>an <code>Axes</code> instance</td>
</tr>
<tr>
<td><code>clim</code></td>
<td>a length 2 sequence of floats</td>
</tr>
<tr>
<td><code>clip_box</code></td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td><code>clip_on</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>clip_path</code></td>
<td>[(Path, Transform), Patch</td>
</tr>
<tr>
<td><code>cmap</code></td>
<td>a colormap or registered colormap name</td>
</tr>
<tr>
<td><code>color</code></td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td><code>contains</code></td>
<td>a callable function</td>
</tr>
<tr>
<td><code>edgecolor</code> or <code>edgecolors</code></td>
<td>matplotlib color spec or sequence of specs</td>
</tr>
<tr>
<td><code>facecolor</code> or <code>facecolors</code></td>
<td>matplotlib color spec or sequence of specs</td>
</tr>
<tr>
<td><code>figure</code></td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td><code>gid</code></td>
<td>an id string</td>
</tr>
<tr>
<td><code>hatch</code></td>
<td>['-', '</td>
</tr>
<tr>
<td><code>label</code></td>
<td>string or anything printable with '%s' conversion.</td>
</tr>
<tr>
<td><code>linestyle</code> or <code>dashes</code> or <code>linestyles</code></td>
<td>['solid', 'dashed', 'dashdot', 'dotted']</td>
</tr>
<tr>
<td><code>linewidth</code> or <code>linewidths</code> or <code>lw</code></td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td><code>norm</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>offset_position</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>offsets</code></td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td><code>path_effects</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>picker</code></td>
<td>[None, float, boolean, callable]</td>
</tr>
<tr>
<td><code>pickradius</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>rasterized</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>sketch_params</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>snap</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>transform</code></td>
<td><code>Transform</code> instance</td>
</tr>
<tr>
<td><code>url</code></td>
<td>a url string</td>
</tr>
<tr>
<td><code>urls</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>visible</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>zorder</code></td>
<td>any number</td>
</tr>
</tbody>
</table>

**See also:**

`pcolor()` For an explanation of the grid orientation ([Grid Orientation](https://matplotlib.org/3.1.1/tutorials/toolkits/axes_grid.html)) and the expansion of 1-D $X$ and/or $Y$ to 2-D arrays.
**Note:** In addition to the above described arguments, this function can take a data keyword argument. If such a data argument is given, the following arguments are replaced by data[arg]:

- All positional and all keyword arguments.

```python
matplotlib.pyplot.phase_spectrum(x, Fs=None, Fc=None, window=None, pad_to=None, sides=None, hold=None, data=None, **kwargs)
```

Plot the phase spectrum.

Call signature:

```python
phase_spectrum(x, Fs=2, Fc=0, window=mlab.window_hanning,
              pad_to=None, sides='default', **kwargs)
```

Compute the phase spectrum (unwrapped angle spectrum) of x. Data is padded to a length of pad_to and the windowing function window is applied to the signal.

**Parameters**

- **x**: 1-D array or sequence
  
  Array or sequence containing the data

- **Fs**: scalar
  
  The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.

- **window**: callable or ndarray
  
  A function or a vector of length NFFT. To create window vectors see window_hanning(), window_none(), numpy.blackman(), numpy.hanning(), numpy.bartlett(), scipy.signal(), scipy.signal.get_window(), etc. The default is window_hanning(). If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

- **sides**: ['default' | 'onesided' | 'twosided']
  
  Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. ‘onesided’ forces the return of a one-sided spectrum, while ‘twosided’ forces two-sided.

- **pad_to**: integer
  
  The number of points to which the data segment is padded when performing the FFT. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the n parameter in the call to fft(). The default is None, which sets pad_to equal to the length of the input signal (i.e. no padding).

- **Fc**: integer
The center frequency of \( x \) (defaults to 0), which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

**kwargs:

Keyword arguments control the Line2D properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>drawstyle</td>
<td>['default'</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>['full'</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with <code>%%s</code> conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['solid'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>marker</td>
<td>A valid marker style</td>
</tr>
<tr>
<td>markeredgecolor or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>[None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
</tbody>
</table>
Table 45.23 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

**Returns spectrum**: 1-D array

The values for the phase spectrum in radians (real valued)

**freqs**: 1-D array

The frequencies corresponding to the elements in *spectrum*

**line**: a *Line2D* instance

The line created by this function

**See also**:  

`magnitude_spectrum()` *magnitude_spectrum()* plots the magnitudes of the corresponding frequencies.  

`angle_spectrum()` *angle_spectrum()* plots the wrapped version of this function.  

`specgram()` *specgram()* can plot the phase spectrum of segments within the signal in a colormap.

In addition to the above described arguments, this function can take a *data* keyword argument. If such a *data* argument is given, the following arguments are replaced by *data*[<arg>]: * All arguments with the following names: ‘x’.
Examples

```
matplotlib.pyplot.pie(x, explode=None, labels=None, colors=None, autopct=None, pctdistance=0.6, shadow=False, labeldistance=1.1, startangle=None, radius=None, counterclock=True, wedgeprops=None, textprops=None, center=(0, 0), frame=False, hold=None, data=None)
```

Plot a pie chart.

Make a pie chart of array \( x \). The fractional area of each wedge is given by \( x / \text{sum}(x) \). If \( \text{sum}(x) \leq 1 \), then the values of \( x \) give the fractional area directly and the array will not be normalized. The wedges are plotted counterclockwise, by default starting from the \( x \)-axis.

**Parameters**

- **\( x \):** array-like
  
  The input array used to make the pie chart.

- **\( \text{explode} \):** array-like, optional, default: None
  
  If not \( \text{None} \), is a \( \text{len}(x) \) array which specifies the fraction of the radius with which to offset each wedge.

- **\( \text{labels} \):** list, optional, default: None
  
  A sequence of strings providing the labels for each wedge

- **\( \text{colors} \):** array-like, optional, default: None
A sequence of matplotlib color args through which the pie chart will cycle. If None, will use the colors in the currently active cycle.

**autopct :** None (default), string, or function, optional

If not None, is a string or function used to label the wedges with their numeric value. The label will be placed inside the wedge. If it is a format string, the label will be `fmt%pct`. If it is a function, it will be called.

**pctdistance :** float, optional, default: 0.6

The ratio between the center of each pie slice and the start of the text generated by `autopct`. Ignored if `autopct` is None.

**shadow :** bool, optional, default: False

Draw a shadow beneath the pie.

**labeldistance :** float, optional, default: 1.1

The radial distance at which the pie labels are drawn.

**startangle :** float, optional, default: None

If not None, rotates the start of the pie chart by `angle` degrees counterclockwise from the x-axis.

**radius :** float, optional, default: None

The radius of the pie, if `radius` is None it will be set to 1.

**counterclock :** bool, optional, default: True

Specify fractions direction, clockwise or counterclockwise.

**wedgeprops :** dict, optional, default: None

Dict of arguments passed to the wedge objects making the pie. For example, you can pass in `'wedgeprops = {‘linewidth’: 3}` to set the width of the wedge border lines equal to 3. For more details, look at the doc/arguments of the wedge object. By default clip_on=False.

**textprops :** dict, optional, default: None

Dict of arguments to pass to the text objects.

**center :** list of float, optional, default: (0, 0)

Center position of the chart. Takes value (0, 0) or is a sequence of 2 scalars.

**frame :** bool, optional, default: False

Plot axes frame with the chart if true.

Returns **patches :** list

A sequence of `matplotlib.patches.Wedge` instances

**texts :** list

A is a list of the label `matplotlib.text.Text` instances.
autotexts : list

A is a list of Text instances for the numeric labels. Is returned only if parameter autopct is not None.

Notes

The pie chart will probably look best if the figure and axes are square, or the Axes aspect is equal.

Examples

matplotlib.pyplot.pink()

set the default colormap to pink and apply to current image if any. See help(colormaps) for more information

Note: In addition to the above described arguments, this function can take a data keyword argument. If such a data argument is given, the following arguments are replaced by data[<arg>]:

• All arguments with the following names: ‘colors’, ‘explode’, ‘labels’, ‘x’.

45.1. matplotlib.pyplot
matplotlib.pyplot.plasma()
set the default colormap to plasma and apply to current image if any. See help(colormaps) for more information

matplotlib.pyplot.plot(*args, **kwargs)
Plot lines and/or markers to the Axes. *args is a variable length argument, allowing for multiple x, y pairs with an optional format string. For example, each of the following is legal:

```
plot(x, y)           # plot x and y using default line style and color
plot(x, y, 'bo')    # plot x and y using blue circle markers
plot(y)             # plot y using x as index array 0..N-1
plot(y, 'r+')       # ditto, but with red plusses
```

If x and/or y is 2-dimensional, then the corresponding columns will be plotted.

If used with labeled data, make sure that the color spec is not included as an element in data, as otherwise the last case plot("v","r", data={"v":..., "r":...}) can be interpreted as the first case which would do plot(v, r) using the default line style and color.

If not used with labeled data (i.e., without a data argument), an arbitrary number of x, y, fmt groups can be specified, as in:

```
a.plot(x1, y1, 'g^', x2, y2, 'g-')
```

Return value is a list of lines that were added.

By default, each line is assigned a different style specified by a 'style cycle'. To change this behavior, you can edit the axes.prop_cycle rcParam.

The following format string characters are accepted to control the line style or marker:
### Line Styles

<table>
<thead>
<tr>
<th>character</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'-'</td>
<td>solid line style</td>
</tr>
<tr>
<td>'--'</td>
<td>dashed line style</td>
</tr>
<tr>
<td>'-.'</td>
<td>dash-dot line style</td>
</tr>
<tr>
<td>':'</td>
<td>dotted line style</td>
</tr>
<tr>
<td>','</td>
<td>point marker</td>
</tr>
<tr>
<td>'o'</td>
<td>circle marker</td>
</tr>
<tr>
<td>'v'</td>
<td>triangle_down marker</td>
</tr>
<tr>
<td>'^'</td>
<td>triangle_up marker</td>
</tr>
<tr>
<td>'&lt;'</td>
<td>triangle_left marker</td>
</tr>
<tr>
<td>'&gt;'</td>
<td>triangle_right marker</td>
</tr>
<tr>
<td>'1'</td>
<td>tri_down marker</td>
</tr>
<tr>
<td>'2'</td>
<td>tri_up marker</td>
</tr>
<tr>
<td>'3'</td>
<td>tri_left marker</td>
</tr>
<tr>
<td>'4'</td>
<td>tri_right marker</td>
</tr>
<tr>
<td>'s'</td>
<td>square marker</td>
</tr>
<tr>
<td>'p'</td>
<td>pentagon marker</td>
</tr>
<tr>
<td>'*'</td>
<td>star marker</td>
</tr>
<tr>
<td>'h'</td>
<td>hexagon1 marker</td>
</tr>
<tr>
<td>'H'</td>
<td>hexagon2 marker</td>
</tr>
<tr>
<td>'+'</td>
<td>plus marker</td>
</tr>
<tr>
<td>'x'</td>
<td>x marker</td>
</tr>
<tr>
<td>'D'</td>
<td>diamond marker</td>
</tr>
<tr>
<td>'d'</td>
<td>thin_diamond marker</td>
</tr>
<tr>
<td>'</td>
<td>'</td>
</tr>
<tr>
<td>'_'</td>
<td>hline marker</td>
</tr>
</tbody>
</table>

### Color Abbreviations

<table>
<thead>
<tr>
<th>character</th>
<th>color</th>
</tr>
</thead>
<tbody>
<tr>
<td>'b'</td>
<td>blue</td>
</tr>
<tr>
<td>'g'</td>
<td>green</td>
</tr>
<tr>
<td>'r'</td>
<td>red</td>
</tr>
<tr>
<td>'c'</td>
<td>cyan</td>
</tr>
<tr>
<td>'m'</td>
<td>magenta</td>
</tr>
<tr>
<td>'y'</td>
<td>yellow</td>
</tr>
<tr>
<td>'k'</td>
<td>black</td>
</tr>
<tr>
<td>'w'</td>
<td>white</td>
</tr>
</tbody>
</table>

The following color abbreviations are supported:

In addition, you can specify colors in many weird and wonderful ways, including full names ('green'), hex strings ('#008000'), RGB or RGBA tuples ((0, 1, 0, 1)) or grayscale intensities as a string (0.8). Of these, the string specifications can be used in place of a **fmt** group, but the tuple forms can be used only as **kwargs**.

Line styles and colors are combined in a single format string, as in 'bo' for blue circles.

The **kwargs** can be used to set line properties (any property that has a `set_*` method). You can use this to set a line label (for auto legends), linewidth, anitialising, marker face color, etc. Here is an
example:

```python
plot([1,2,3], [1,2,3], 'go-', label='line 1', linewidth=2)
plot([1,2,3], [1,4,9], 'rs', label='line 2')
axis([0, 4, 0, 10])
legend()
```

If you make multiple lines with one plot command, the kwargs apply to all those lines, e.g.:

```python
plot(x1, y1, x2, y2, antialiased=False)
```

Neither line will be antialiased.

You do not need to use format strings, which are just abbreviations. All of the line properties can be controlled by keyword arguments. For example, you can set the color, marker, linestyle, and marker color with:

```python
plot(x, y, color='green', linestyle='dashed', marker='o',
     markerfacecolor='blue', markersize=12).
```

See `Line2D` for details.

The kwargs are `Line2D` properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an <code>Axes</code> instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>drawstyle</td>
<td>['default'</td>
</tr>
<tr>
<td>figure</td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>['full'</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyles or ls</td>
<td>['solid'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>marker</td>
<td>A valid marker style</td>
</tr>
<tr>
<td>markeredgecolor or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgedwidth or mew</td>
<td>float value in points</td>
</tr>
</tbody>
</table>
Table 45.24 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>[None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>solid_joistyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

kwars `scalex` and `scaley`, if defined, are passed on to `autoscale_view()` to determine whether the x and y axes are autoscaled; the default is `True`.

**Note:** In addition to the above described arguments, this function can take a `data` keyword argument. If such a `data` argument is given, the following arguments are replaced by `data[<arg>]`:

- All arguments with the following names: ‘x’, ‘y’.

```python
matplotlib.pyplot.plot_date(x, y, fmt='o', tz=None, xdate=True, ydate=False, hold=None, data=None, **kwargs)
```

A plot with data that contains dates.

Similar to the `plot()` command, except the x or y (or both) data is considered to be dates, and the axis is labeled accordingly.

x and/or y can be a sequence of dates represented as float days since 0001-01-01 UTC.

Note if you are using custom date tickers and formatters, it may be necessary to set the formatters/locators after the call to method `plot_date` since method `plot_date` will set the default tick locator to class: `matplotlib.dates.AutoDateLocator` (if the tick locator is not already set to a class: `matplotlib.dates.DateLocator` instance) and the default tick formatter to class: `matplotlib.dates.AutoDateFormatter` (if the tick formatter is not already set to a class: `matplotlib.dates.DateFormatter` instance).

**Parameters** `fmt` : string
The plot format string.

**tz**: [``None`` | timezone string | tzinfo instance]

The time zone to use in labeling dates. If `None`, defaults to rc value.

**xdate**: boolean

If `True`, the x-axis will be labeled with dates.

**ydate**: boolean

If `True`, the y-axis will be labeled with dates.

**Returns** lines

**Other Parameters** `kwargs`: `matplotlib.lines.Line2D` properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>agg_filter</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>alpha</code></td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td><code>animated</code></td>
<td>[<code>True</code></td>
</tr>
<tr>
<td><code>antialiased</code> or <code>aa</code></td>
<td>[<code>True</code></td>
</tr>
<tr>
<td><code>axes</code></td>
<td>an <code>Axes</code> instance</td>
</tr>
<tr>
<td><code>clip_box</code></td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td><code>clip_on</code></td>
<td>[<code>True</code></td>
</tr>
<tr>
<td><code>clip_path</code></td>
<td>[(<code>Path</code></td>
</tr>
<tr>
<td><code>color</code> or <code>c</code></td>
<td>any <code>matplotlib</code> color</td>
</tr>
<tr>
<td><code>contains</code></td>
<td>a callable function</td>
</tr>
<tr>
<td><code>dash_capstyle</code></td>
<td>[<code>butt</code></td>
</tr>
<tr>
<td><code>dash_joinstyle</code></td>
<td>[<code>miter</code></td>
</tr>
<tr>
<td><code>dashes</code></td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td><code>drawstyle</code></td>
<td>[<code>default</code></td>
</tr>
<tr>
<td><code>figure</code></td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td><code>fillstyle</code></td>
<td>[<code>full</code></td>
</tr>
<tr>
<td><code>gid</code></td>
<td>an id string</td>
</tr>
<tr>
<td><code>label</code></td>
<td>string or anything printable with <code>%%s</code> conversion.</td>
</tr>
<tr>
<td><code>linestyle</code> or <code>ls</code></td>
<td>[<code>solid</code></td>
</tr>
<tr>
<td><code>linewidth</code> or <code>lw</code></td>
<td>float value in points</td>
</tr>
<tr>
<td><code>marker</code></td>
<td>A valid marker style</td>
</tr>
<tr>
<td><code>markeredgecolor</code> or <code>mec</code></td>
<td>any <code>matplotlib</code> color</td>
</tr>
<tr>
<td><code>markeredgewidth</code> or <code>mew</code></td>
<td>float value in points</td>
</tr>
<tr>
<td><code>markerfacecolor</code> or <code>mfc</code></td>
<td>any <code>matplotlib</code> color</td>
</tr>
<tr>
<td><code>markerfacecoloralt</code> or <code>mfcalt</code></td>
<td>any <code>matplotlib</code> color</td>
</tr>
<tr>
<td><code>markersize</code> or <code>ms</code></td>
<td>float</td>
</tr>
<tr>
<td><code>markevery</code></td>
<td>[<code>None</code></td>
</tr>
<tr>
<td><code>path_effects</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>picker</code></td>
<td>float distance in points or callable pick function <code>fn(artist, event)</code></td>
</tr>
</tbody>
</table>
.. note::

In addition to the above described arguments, this function can take a `data` keyword argument. If such a `data` argument is given, the following arguments are replaced by `data[<arg>]`:

- All arguments with the following names: ‘x’, ‘y’.

See also:

- `matplotlib.dates` helper functions on dates
- `matplotlib.dates.date2num` how to convert dates to num
- `matplotlib.dates.num2date` how to convert num to dates
- `matplotlib.dates.drange` how floating point dates

`matplotlib.pyplot.plotfile` *(fname, cols=(0,), plotfuncs=None, comments='#', skiprows=0, checkrows=5, delimiter=' ', names=None, subplots=True, newfig=True, **kwargs)*

Plot the data in a file.

cols is a sequence of column identifiers to plot. An identifier is either an int or a string. If it is an int, it indicates the column number. If it is a string, it indicates the column header. `matplotlib` will make column headers lower case, replace spaces with underscores, and remove all illegal characters; so 'Adj Close*' will have name 'adj_close'.

- If `len(cols) == 1`, only that column will be plotted on the y axis.
- If `len(cols) > 1`, the first element will be an identifier for data for the x axis and the remaining elements will be the column indexes for multiple subplots if `subplots` is `True` (the default), or for lines in a single subplot if `subplots` is `False`.
**plotfuncs**, if not `None`, is a dictionary mapping identifier to an `Axes` plotting function as a string. Default is ‘plot’, other choices are ‘semilogy’, ‘fill’, ‘bar’, etc. You must use the same type of identifier in the `cols` vector as you use in the `plotfuncs` dictionary, e.g., integer column numbers in both or column names in both. If `subplots` is `False`, then including any function such as ‘semilogy’ that changes the axis scaling will set the scaling for all columns.

`comments`, `skiprows`, `checkrows`, `delimiter`, and `names` are all passed on to `matplotlib.pylab.csv2rec()` to load the data into a record array.

If `newfig` is `True`, the plot always will be made in a new figure; if `False`, it will be made in the current figure if one exists, else in a new figure.

`kwargs` are passed on to plotting functions.

Example usage:

```python
# plot the 2nd and 4th column against the 1st in two subplots
plotfile(fname, (0,1,3))

# plot using column names; specify an alternate plot type for volume
plotfile(fname, ('date', 'volume', 'adj_close'),
         plotfuncs={'volume': 'semilogy'})
```

Note: plotfile is intended as a convenience for quickly plotting data from flat files; it is not intended as an alternative interface to general plotting with pyplot or matplotlib.

**matplotlib.pyplot.polar(**args, **kwargs)**

Make a polar plot.

```python
polar(theta, r, **kwargs)
```

Multiple `theta`, `r` arguments are supported, with format strings, as in `plot()`.

**matplotlib.pyplot.prism()**

set the default colormap to prism and apply to current image if any. See `help(colormaps)` for more information

**matplotlib.pyplot.psd(x, NFFT=None, Fs=None, Fc=None, detrend=None, window=None, noverlap=None, pad_to=None, sides='default', scale_by_freq=None, return_line=None, hold=None, data=None, **kwargs)**

Plot the power spectral density.

Call signature:

```python
psd(x, NFFT=256, Fs=2, Fc=0, detrend=mlab.detrend_none, window=mlab.window_hanning, noverlap=0, pad_to=None, sides='default', scale_by_freq=None, return_line=None, hold=None, data=None, **kwargs)
```

The power spectral density $P_x$, by Welch’s average periodogram method. The vector $x$ is divided into $NFFT$ length segments. Each segment is detrended by function `detrend` and windowed by function `window`. `noverlap` gives the length of the overlap between segments. The $|\text{fft}(i)|^2$ of each segment $i$ are averaged to compute $P_x$, with a scaling to correct for power loss due to windowing.
If \( \text{len}(x) < NFFT \), it will be zero padded to \( NFFT \).

**Parameters**
- **x**: 1-D array or sequence
  - Array or sequence containing the data
- **Fs**: scalar
  - The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.
- **window**: callable or ndarray
  - A function or a vector of length \( NFFT \). To create window vectors see `window_hanning()`, `window_none()`, `numpy.blackman()`, `numpy.hamming()`, `numpy.bartlett()`, `scipy.signal()`, `scipy.signal.get_window()`, etc. The default is `window_hanning()`. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.
- **sides**: [‘default’ | ‘onesided’ | ‘twosided’ ]
  - Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. ‘onesided’ forces the return of a one-sided spectrum, while ‘twosided’ forces two-sided.
- **pad_to**: integer
  - The number of points to which the data segment is padded when performing the FFT. This can be different from \( NFFT \), which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the \( n \) parameter in the call to `fft()`. The default is None, which sets `pad_to` equal to \( NFFT \)
- **NFFT**: integer
  - The number of data points used in each block for the FFT. A power 2 is most efficient. The default value is 256. This should NOT be used to get zero padding, or the scaling of the result will be incorrect. Use `pad_to` for this instead.
- **detrend**: {'default’, ‘constant’, ‘mean’, ‘linear’, ‘none’} or callable
  - The function applied to each segment before fft-ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the `detrend` parameter is a vector, in matplotlib is it a function. The `pylab` module defines `detrend_none()`, `detrend_mean()`, and `detrend_linear()`, but you can use a custom function as well. You can also use a string to choose one of the functions. ‘default’, ‘constant’, and ‘mean’ call `detrend_mean()`. ‘linear’ calls `detrend_linear()`. ‘none’ calls `detrend_none()`.
- **scale_by_freq**: boolean, optional
Specifies whether the resulting density values should be scaled by the scaling frequency, which gives density in units of Hz^-1. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

noverlap : integer

The number of points of overlap between segments. The default value is 0 (no overlap).

Fc : integer

The center frequency of x (defaults to 0), which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

return_line : bool

Whether to include the line object plotted in the returned values. Default is False.

**kwargs :

Keyword arguments control the Line2D properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color or c</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>dash_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>dash_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>dashes</td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td>drawstyle</td>
<td>['default'</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fillstyle</td>
<td>['full'</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>['solid'</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float value in points</td>
</tr>
<tr>
<td>marker</td>
<td>A valid marker style</td>
</tr>
<tr>
<td>markeredgecolor or mec</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markeredgewidth or mew</td>
<td>float value in points</td>
</tr>
<tr>
<td>markerfacecolor or mfc</td>
<td>any matplotlib color</td>
</tr>
</tbody>
</table>
Table 45.26 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>markerfacecoloralt or mfcalt</td>
<td>any matplotlib color</td>
</tr>
<tr>
<td>markersize or ms</td>
<td>float</td>
</tr>
<tr>
<td>markevery</td>
<td>[None</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

**Returns**

\( P_{xx} \) : 1-D array

The values for the power spectrum \( P_{xx} \) before scaling (real valued)

\( \text{freqs} \) : 1-D array

The frequencies corresponding to the elements in \( P_{xx} \)

\( \text{line} \) : a Line2D instance

The line created by this function. Only returned if \( \text{return\_line} \) is True.

**See also:**

\( \text{specgram()} \) \( \text{specgram()} \) differs in the default overlap; in not returning the mean of the segment periodograms; in returning the times of the segments; and in plotting a colormap instead of a line.

\( \text{magnitude\_spectrum()} \) \( \text{magnitude\_spectrum()} \) plots the magnitude spectrum.

\( \text{csd()} \) \( \text{csd()} \) plots the spectral density between two signals.

In addition to the above described arguments, this function can take a \( \text{data} \) keyword argument. If such a \( \text{data} \) argument is given, the following arguments are replaced by \( \text{data}[<\text{arg}>] \) : * All arguments with the following names: ‘x’.

**Notes**

For plotting, the power is plotted as \( 10 \log_{10}(P_{xx}) \) for decibels, though \( P_{xx} \) itself is returned.
References


Examples

![Graph showing a 2-D field of arrows and a Power Spectral Density plot.

```python
matplotlib.pyplot.quiver(*args, **kw)
```

Plot a 2-D field of arrows.

Call signatures:

```python
quiver(U, V, **kw)
quiver(U, V, C, **kw)
quiver(X, Y, U, V, **kw)
quiver(X, Y, U, V, C, **kw)
```

$U$ and $V$ are the arrow data, $X$ and $Y$ set the location of the arrows, and $C$ sets the color of the arrows. These arguments may be 1-D or 2-D arrays or sequences.

If $X$ and $Y$ are absent, they will be generated as a uniform grid. If $U$ and $V$ are 2-D arrays and $X$ and $Y$ are 1-D, and if len($X$) and len($Y$) match the column and row dimensions of $U$, then $X$ and $Y$ will be expanded with `numpy.meshgrid()`.
The default settings auto-scales the length of the arrows to a reasonable size. To change this behavior see the `scale` and `scale_units` kwargs.

The defaults give a slightly swept-back arrow; to make the head a triangle, make `headaxislength` the same as `headlength`. To make the arrow more pointed, reduce `headwidth` or increase `headlength` and `headaxislength`. To make the head smaller relative to the shaft, scale down all the head parameters. You will probably do best to leave minshaft alone.

`linewdths` and `edgecolors` can be used to customize the arrow outlines.

**Parameters**

- **X**: 1D or 2D array, sequence, optional
  - The x coordinates of the arrow locations

- **Y**: 1D or 2D array, sequence, optional
  - The y coordinates of the arrow locations

- **U**: 1D or 2D array or masked array, sequence
  - The x components of the arrow vectors

- **V**: 1D or 2D array or masked array, sequence
  - The y components of the arrow vectors

- **C**: 1D or 2D array, sequence, optional
  - The arrow colors

- **units**: [‘width’ | ‘height’ | ‘dots’ | ‘inches’ | ‘x’ | ‘y’ | ‘xy’]
  - The arrow dimensions (except for `length`) are measured in multiples of this unit.
  - ‘width’ or ‘height’: the width or height of the axis
  - ‘dots’ or ‘inches’: pixels or inches, based on the figure dpi
  - ‘x’, ‘y’, or ‘xy’: respectively X, Y, or $\sqrt{X^2 + Y^2}$ in data units

The arrows scale differently depending on the units. For ‘x’ or ‘y’, the arrows get larger as one zooms in; for other units, the arrow size is independent of the zoom state. For ‘width’ or ‘height’, the arrow size increases with the width and height of the axes, respectively, when the window is resized; for ‘dots’ or ‘inches’, resizing does not change the arrows.

- **angles**: [‘uv’ | ‘xy’], array, optional
  - Method for determining the angle of the arrows. Default is ‘uv’.
  - ‘uv’: the arrow axis aspect ratio is 1 so that if $U == V$ the orientation of the arrow on the plot is 45 degrees counter-clockwise from the horizontal axis (positive to the right).
  - ‘xy’: arrows point from (x,y) to (x+u, y+v). Use this for plotting a gradient field, for example.
Alternatively, arbitrary angles may be specified as an array of values in degrees, counter-clockwise from the horizontal axis.

Note: inverting a data axis will correspondingly invert the arrows only with angles='xy'.

**scale** : None, float, optional

Number of data units per arrow length unit, e.g., m/s per plot width; a smaller scale parameter makes the arrow longer. Default is None.

If None, a simple autoscaling algorithm is used, based on the average vector length and the number of vectors. The arrow length unit is given by the `scale_units` parameter

**scale_units** : ['width' | 'height' | 'dots' | 'inches' | 'x' | 'y' | 'xy'], None, optional

If the `scale` kwarg is None, the arrow length unit. Default is None.

e.g. `scale_units` is 'inches', `scale` is 2.0, and \((u, v) = (1, 0)\), then the vector will be 0.5 inches long.

If `scale_units` is 'width'/height’, then the vector will be half the width/height of the axes.

If `scale_units` is 'x' then the vector will be 0.5 x-axis units. To plot vectors in the x-y plane, with u and v having the same units as x and y, use angles='xy', `scale_units='xy'`, scale=1.

**width** : scalar, optional

Shaft width in arrow units; default depends on choice of units, above, and number of vectors; a typical starting value is about 0.005 times the width of the plot.

**headwidth** : scalar, optional

Head width as multiple of shaft width, default is 3

**headlength** : scalar, optional

Head length as multiple of shaft width, default is 5

**headaxislength** : scalar, optional

Head length at shaft intersection, default is 4.5

**minshaft** : scalar, optional

Length below which arrow scales, in units of head length. Do not set this to less than 1, or small arrows will look terrible! Default is 1

**minlength** : scalar, optional

Minimum length as a multiple of shaft width; if an arrow length is less than this, plot a dot (hexagon) of this diameter instead. Default is 1.

**pivot** : ['tail' | 'mid' | 'middle' | 'tip'], optional
The part of the arrow that is at the grid point; the arrow rotates about this point, hence the name *pivot*.

**color** : [ color | color sequence ], optional

This is a synonym for the `PolyCollection` facecolor kwarg. If `C` has been set, `color` has no effect.

**See also:**

*quiverkey* Add a key to a quiver plot

**Notes**

Additional `PolyCollection` keyword arguments:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>agg_filter</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>alpha</code></td>
<td>float or None</td>
</tr>
<tr>
<td><code>animated</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>antialiased</code></td>
<td>Boolean or sequence of bools</td>
</tr>
<tr>
<td><code>array</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>axes</code></td>
<td>an <em>Axes</em> instance</td>
</tr>
<tr>
<td><code>clim</code></td>
<td>a length 2 sequence of floats</td>
</tr>
<tr>
<td><code>clip_box</code></td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td><code>clip_on</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>clip_path</code></td>
<td>[ (<code>Path</code>, <code>Transform</code>)</td>
</tr>
<tr>
<td><code>cmap</code></td>
<td>a colormap or registered colormap name</td>
</tr>
<tr>
<td><code>color</code></td>
<td>matplotlib color arg or sequence of rgba tuples</td>
</tr>
<tr>
<td><code>contains</code></td>
<td>a callable function</td>
</tr>
<tr>
<td><code>edgecolor</code></td>
<td>matplotlib color spec or sequence of specs</td>
</tr>
<tr>
<td><code>facecolor</code></td>
<td>matplotlib color spec or sequence of specs</td>
</tr>
<tr>
<td><code>figure</code></td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td><code>gid</code></td>
<td>an id string</td>
</tr>
<tr>
<td><code>hatch</code></td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td><code>label</code></td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td><code>linestyle</code></td>
<td>[‘solid’</td>
</tr>
<tr>
<td><code>linewidth</code></td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td><code>norm</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>offset_position</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>offsets</code></td>
<td>float or sequence of floats</td>
</tr>
<tr>
<td><code>path_effects</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>picker</code></td>
<td>[None</td>
</tr>
<tr>
<td><code>pickradius</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>rasterized</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>sketch_params</code></td>
<td>unknown</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>snap</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>transform</code></td>
<td><code>Transform</code> instance</td>
</tr>
<tr>
<td><code>url</code></td>
<td>a url string</td>
</tr>
<tr>
<td><code>urls</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>visible</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>zorder</code></td>
<td>any number</td>
</tr>
</tbody>
</table>

**Examples**

```
→ Quiver key, length = 10
```

```python
matplotlib.pyplot.quiverkey(*args, **kw)
```

Add a key to a quiver plot.

Call signature:

```
quiverkey(Q, X, Y, U, label, **kw)
```

Arguments:

* `Q`: The Quiver instance returned by a call to quiver.
$X$, $Y$: The location of the key; additional explanation follows.

$U$: The length of the key

**label**: A string with the length and units of the key

Keyword arguments:

- **coordinates** = [‘axes’ | ‘figure’ | ‘data’ | ‘inches’] Coordinate system and units for $X$, $Y$: ‘axes’ and ‘figure’ are normalized coordinate systems with 0,0 in the lower left and 1,1 in the upper right; ‘data’ are the axes data coordinates (used for the locations of the vectors in the quiver plot itself); ‘inches’ is position in the figure in inches, with 0,0 at the lower left corner.

- **color**: overrides face and edge colors from $Q$.

- **labellpos** = [‘N’ | ‘S’ | ‘E’ | ‘W’] Position the label above, below, to the right, to the left of the arrow, respectively.

- **labelsep**: Distance in inches between the arrow and the label. Default is 0.1

- **labelcolor**: defaults to default Text color.

- **fontproperties**: A dictionary with keyword arguments accepted by the FontProperties initializer: family, style, variant, size, weight

Any additional keyword arguments are used to override vector properties taken from $Q$.

The positioning of the key depends on $X$, $Y$, **coordinates**, and **labellpos**. If **labellpos** is ‘N’ or ‘S’, $X$, $Y$ give the position of the middle of the key arrow. If **labellpos** is ‘E’, $X$, $Y$ positions the head, and if **labellpos** is ‘W’, $X$, $Y$ positions the tail; in either of these two cases, $X$, $Y$ is somewhere in the middle of the arrow+label key object.

**matplotlib.pyplot.rc**(*args, **kwargs)

Set the current rc params. Group is the grouping for the rc, e.g., for **lines.linewidth** the group is **lines**, for **axes.facecolor**, the group is **axes**, and so on. Group may also be a list or tuple of group names, e.g., (xtick, ytick). **kwargs** is a dictionary attribute name/value pairs, e.g.:

```python
rc('lines', linewidth=2, color='r')
```

sets the current rc params and is equivalent to:

```python
rcParams['lines.linewidth'] = 2
rcParams['lines.color'] = 'r'
```

The following aliases are available to save typing for interactive users:

<table>
<thead>
<tr>
<th>Alias</th>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘lw’</td>
<td>‘linewidth’</td>
</tr>
<tr>
<td>‘ls’</td>
<td>‘linestyle’</td>
</tr>
<tr>
<td>‘c’</td>
<td>‘color’</td>
</tr>
<tr>
<td>‘fc’</td>
<td>‘facecolor’</td>
</tr>
<tr>
<td>‘ec’</td>
<td>‘edgecolor’</td>
</tr>
<tr>
<td>‘mew’</td>
<td>‘markeredgewidth’</td>
</tr>
<tr>
<td>‘aa’</td>
<td>‘antialiased’</td>
</tr>
</tbody>
</table>
Thus you could abbreviate the above rc command as:

```python
rc('lines', lw=2, c='r')
```

Note you can use python’s kwargs dictionary facility to store dictionaries of default parameters. e.g., you can customize the font rc as follows:

```python
font = {'family': 'monospace',  
        'weight': 'bold',  
        'size': 'larger'}
rc('font', **font)  # pass in the font dict as kwargs
```

This enables you to easily switch between several configurations. Use matplotlib.style.use('default') or rcdefaults() to restore the default rc params after changes.

```python
matplotlib.pyplot.rc_context(rc=None, fname=None)
```

Return a context manager for managing rc settings.

This allows one to do:

```python
with mpl.rc_context(fname='screen.rc'):
    plt.plot(x, a)
with mpl.rc_context(fname='print.rc'):
    plt.plot(x, b)
plt.plot(x, c)
```

The ‘a’ vs ‘x’ and ‘c’ vs ‘x’ plots would have settings from ‘screen.rc’, while the ‘b’ vs ‘x’ plot would have settings from ‘print.rc’.

A dictionary can also be passed to the context manager:

```python
with mpl.rc_context(rc={'text.usetex': True}, fname='screen.rc'):
    plt.plot(x, a)
```

The ‘rc’ dictionary takes precedence over the settings loaded from ‘fname’. Passing a dictionary only is also valid.

```python
matplotlib.pyplot.rcdefaults()
```

Restore the rc params from Matplotlib’s internal defaults.

See also:

- `rc_file_defaults` Restore the rc params from the rc file originally loaded by Matplotlib.
- `matplotlib.style.use` Use a specific style file. Call style.use('default') to restore the default style.

```python
matplotlib.pyplot.rgrids(*args, **kwargs)
```

Get or set the radial gridlines on a polar plot.

call signatures:
```python
lines, labels = rgrids()
lines, labels = rgrids(radii, labels=None, angle=22.5, **kwargs)
```

When called with no arguments, rgrid() simply returns the tuple (lines, labels), where lines is an array of radial gridlines (Line2D instances) and labels is an array of tick labels (Text instances). When called with arguments, the labels will appear at the specified radial distances and angles.

labels, if not None, is a len(radii) list of strings of the labels to use at each angle.

If labels is None, the rformatter will be used.

Examples:

```python
# set the locations of the radial gridlines and labels
lines, labels = rgrids( (0.25, 0.5, 1.0) )

# set the locations and labels of the radial gridlines and labels
lines, labels = rgrids( (0.25, 0.5, 1.0), ('Tom', 'Dick', 'Harry') )
```

```python
matplotlib.pyplot.savefig(*args, **kwargs)
```

Save the current figure.

Call signature:

```python
savefig(fname, dpi=None, facecolor='w', edgecolor='w', orientation='portrait', papertype=None, format=None, transparent=False, bbox_inches=None, pad_inches=0.1, frameon=None)
```

The output formats available depend on the backend being used.

Arguments:

- **fname**: A string containing a path to a filename, or a Python file-like object, or possibly some backend-dependent object such as PdfPages.

  If format is None and fname is a string, the output format is deduced from the extension of the filename. If the filename has no extension, the value of the rc parameter savefig.format is used.

  If fname is not a string, remember to specify format to ensure that the correct backend is used.

Keyword arguments:

- **dpi**: [None | scalar > 0 | ‘figure’] The resolution in dots per inch. If None it will default to the value savefig.dpi in the matplotlibrc file. If ‘figure’ it will set the dpi to be the value of the figure.

- **facecolor, edgecolor**: the colors of the figure rectangle

- **orientation**: ['landscape' | 'portrait'] not supported on all backends; currently only on postscript output

\textbf{format}: One of the file extensions supported by the active backend. Most backends support png, pdf, ps, eps and svg.

\textbf{transparent}: If True, the axes patches will all be transparent; the figure patch will also be transparent unless facecolor and/or edgecolor are specified via kwargs. This is useful, for example, for displaying a plot on top of a colored background on a web page. The transparency of these patches will be restored to their original values upon exit of this function.

\textbf{frameon}: If True, the figure patch will be colored, if False, the figure background will be transparent. If not provided, the rcParam `savefig.frameon` will be used.

\textbf{bbox_inches}: Bbox in inches. Only the given portion of the figure is saved. If ‘tight’, try to figure out the tight bbox of the figure.

\textbf{pad_inches}: Amount of padding around the figure when bbox_inches is ‘tight’.

\textbf{bbox_extra_artists}: A list of extra artists that will be considered when the tight bbox is calculated.

\begin{verbatim}
matplotlib.pyplot.sca(ax)
Set the current Axes instance to ax.

The current Figure is updated to the parent of ax.

matplotlib.pyplot.scatter(x, y, s=None, c=None, marker=None, cmap=None, norm=None, vmin=None, vmax=None, alpha=None, linewidths=None, verts=None, edgecolors=None, hold=None, data=None, **kwargs)
Make a scatter plot of x vs y

Marker size is scaled by s and marker color is mapped to c

Parameters x, y : array_like, shape (n, )

Input data

s : scalar or array_like, shape (n,), optional
    size in points^2. Default is rcParams[‘lines.markersize’] ** 2.

c : color, sequence, or sequence of color, optional, default: ‘b’

c can be a single color format string, or a sequence of color specifications of length n, or a sequence of N numbers to be mapped to colors using the cmap and norm specified via kwargs (see below). Note that c should not be a single numeric RGB or RGBA sequence because that is indistinguishable from an array of values to be colormapped. c can be a 2-D array in which the rows are RGB or RGBA, however, including the case of a single row to specify the same color for all points.

marker : MarkerStyle, optional, default: ‘o’

\end{verbatim}
See markers for more information on the different styles of markers scatter supports. marker can be either an instance of the class or the text shorthand for a particular marker.

cmap : Colormap, optional, default: None

A Colormap instance or registered name. cmap is only used if c is an array of floats. If None, defaults to rc image.cmap.

norm : Normalize, optional, default: None

A Normalize instance is used to scale luminance data to 0, 1. norm is only used if c is an array of floats. If None, use the default normalize().

vmin, vmax : scalar, optional, default: None

vmin and vmax are used in conjunction with norm to normalize luminance data. If either are None, the min and max of the color array is used. Note if you pass a norm instance, your settings for vmin and vmax will be ignored.

alpha : scalar, optional, default: None

The alpha blending value, between 0 (transparent) and 1 (opaque)

linewidths : scalar or array_like, optional, default: None

If None, defaults to (lines.linewidth,).

verts : sequence of (x, y), optional

If marker is None, these vertices will be used to construct the marker. The center of the marker is located at (0,0) in normalized units. The overall marker is rescaled by s.

edgecolors : color or sequence of color, optional, default: None

If None, defaults to ‘face’

If ‘face’, the edge color will always be the same as the face color.

If it is ‘none’, the patch boundary will not be drawn.

For non-filled markers, the edgecolors kwarg is ignored and forced to ‘face’ internally.

Returns paths : PathCollection

Other Parameters kwargs : Collection properties

See also:

plot to plot scatter plots when markers are identical in size and color

Notes

• The plot function will be faster for scatterplots where markers don’t vary in size or color.
• Any or all of x, y, s, and c may be masked arrays, in which case all masks will be combined and only unmasked points will be plotted.

Fundamentally, scatter works with 1-D arrays; x, y, s, and c may be input as 2-D arrays, but within scatter they will be flattened. The exception is c, which will be flattened only if its size matches the size of x and y.

Examples

![Scatter plot example]

**Note:** In addition to the above described arguments, this function can take a `data` keyword argument. If such a `data` argument is given, the following arguments are replaced by `data[<arg>]`:


**matplotlib.pyplot.sci**(im)

Set the current image. This image will be the target of colormap commands like `jet()`, `hot()` or `clim()`). The current image is an attribute of the current axes.

**matplotlib.pyplot.semilogx**(args, **kwargs)

Make a plot with log scaling on the x axis.
Parameters **basex** : float, optional

Base of the x logarithm. The scalar should be larger than 1.

**subsx** : array_like, optional

The location of the minor xticks; *None* defaults to autosubs, which depend on
the number of decades in the plot; see `set_xscale()` for details.

**nonposx** : string, optional, {'mask', 'clip'}

Non-positive values in x can be masked as invalid, or clipped to a very small
positive number.

Returns **plot**

Log-scaled plot on the x axis.

Other Parameters :class:`~matplotlib.lines.Line2D` properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>agg_filter</strong></td>
<td>unknown</td>
</tr>
<tr>
<td><strong>alpha</strong></td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td><strong>animated</strong></td>
<td>[True</td>
</tr>
<tr>
<td><strong>antialiased</strong></td>
<td>[True</td>
</tr>
<tr>
<td><strong>axes</strong></td>
<td>an Axes instance</td>
</tr>
<tr>
<td><strong>clip_box</strong></td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td><strong>clip_on</strong></td>
<td>[True</td>
</tr>
<tr>
<td><strong>clip_path</strong></td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td><strong>color</strong> or <strong>c</strong></td>
<td>any matplotlib color</td>
</tr>
<tr>
<td><strong>contains</strong></td>
<td>a callable function</td>
</tr>
<tr>
<td><strong>dash_capstyle</strong></td>
<td>['butt'</td>
</tr>
<tr>
<td><strong>dash_jjoinstyle</strong></td>
<td>['miter'</td>
</tr>
<tr>
<td><strong>dashes</strong></td>
<td>sequence of on/off ink in points</td>
</tr>
<tr>
<td><strong>drawstyle</strong></td>
<td>['default'</td>
</tr>
<tr>
<td><strong>figure</strong></td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td><strong>fillstyle</strong></td>
<td>['full'</td>
</tr>
<tr>
<td><strong>gid</strong></td>
<td>an id string</td>
</tr>
<tr>
<td><strong>label</strong></td>
<td>string or anything printable with %s conversion.</td>
</tr>
<tr>
<td><strong>linestyle</strong> or <strong>ls</strong></td>
<td>['solid'</td>
</tr>
<tr>
<td><strong>linewidth</strong> or <strong>lw</strong></td>
<td>float value in points</td>
</tr>
<tr>
<td><strong>marker</strong></td>
<td>A valid marker style</td>
</tr>
<tr>
<td><strong>markeredgecolor</strong> or <strong>mec</strong></td>
<td>any matplotlib color</td>
</tr>
<tr>
<td><strong>markeredgewidth</strong> or <strong>mew</strong></td>
<td>float value in points</td>
</tr>
<tr>
<td><strong>markerfacecolor</strong> or <strong>mfc</strong></td>
<td>any matplotlib color</td>
</tr>
<tr>
<td><strong>markerfacecoloralt</strong> or <strong>mfcalt</strong></td>
<td>any matplotlib color</td>
</tr>
<tr>
<td><strong>markersize</strong> or <strong>ms</strong></td>
<td>float</td>
</tr>
<tr>
<td><strong>markevery</strong></td>
<td>[None</td>
</tr>
<tr>
<td><strong>path_effects</strong></td>
<td>unknown</td>
</tr>
<tr>
<td><strong>picker</strong></td>
<td>float distance in points or callable pick function fn(artist, event)</td>
</tr>
</tbody>
</table>
Table 45.28 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pickradius</td>
<td>float distance in points</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>solid_capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>solid_joinstyle</td>
<td>['miter'</td>
</tr>
<tr>
<td>transform</td>
<td>a matplotlib.transforms.Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xdata</td>
<td>1D array</td>
</tr>
<tr>
<td>ydata</td>
<td>1D array</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

See also:

`loglog` For example code and figure.

Notes

This function supports all the keyword arguments of `plot()` and `matplotlib.axes.Axes.set_xscale()`.

`matplotlib.pyplot.semilogy(*args, **kwargs)`

Make a plot with log scaling on the y axis.

Parameters basey : scalar > 1

Base of the y logarithm.

subsy [None or iterable] The location of the minor yticks. None defaults to autosubs, which depend on the number of decades in the plot. See `set_yticks()` for details.

nonposy [{‘mask’ | ‘clip’}] str Non-positive values in y can be masked as invalid, or clipped to a very small positive number.

Returns Line2D

Line instance of the plot.

Other Parameters kwars : Line2D properties.

plot and `matplotlib.axes.Axes.set_yticks` arguments.
Property Description

================================================================================

:meth:`agg_filter <matplotlib.artist.Artist.set_agg_filter>`\  unknown

:meth:`alpha <matplotlib.artist.Artist.set_alpha>`\  float (0.0 transparent through 1.0 opaque)

:meth:`animated <matplotlib.artist.Artist.set_animated>`\  [True | False]

:meth:`antialiased <matplotlib.lines.Line2D.set_antialiased>`\  or aa [True | False]


:meth:`clip_on <matplotlib.artist.Artist.set_clip_on>`\  [True | False]


:meth:`color <matplotlib.lines.Line2D.set_color>`\  or c any matplotlib color

:meth:`contains <matplotlib.artist.Artist.set_contains>`\  a callable function

:meth:`dash_capstyle <matplotlib.lines.Line2D.set_dash_capstyle>`\  ['butt' | 'round' | 'projecting']

:meth:`dash_joinstyle <matplotlib.lines.Line2D.set_dash_joinstyle>`\  ['miter' | 'round' | 'bevel']

:meth:`dashes <matplotlib.lines.Line2D.set_dashes>`\  sequence of on/off ink in points

:meth:`drawstyle <matplotlib.lines.Line2D.set_drawstyle>`\  ['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post']

:meth:`figure <matplotlib.artist.Artist.set_figure>`\  a :class:`matplotlib.figure.Figure` instance

:meth:`fillstyle <matplotlib.lines.Line2D.set_fillstyle>`\  ['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']

:meth:`gid <matplotlib.artist.Artist.set_gid>`\  an id string

:meth:`label <matplotlib.artist.Artist.set_label>`\  string or anything printable with \%s conversion.

:meth:`linestyle <matplotlib.lines.Line2D.set_linestyle>`\  or ls ['solid' | 'dashed', 'dashdot', 'dotted'] \ (offset, on-off-dash-seq) \ | 'None' | ' ' \ | ' ' ' ']

45.1. matplotlib.pyplot 1549
:method:`linewidth <matplotlib.lines.Line2D.set_linewidth>` or `lw` float value in points


:method:`markeredgecolor <matplotlib.lines.Line2D.set_markeredgecolor>` or `mec` any matplotlib color

:method:`markeredgewidth <matplotlib.lines.Line2D.set_markeredgewidth>` or `mew` float value in points

:method:`markerfacecolor <matplotlib.lines.Line2D.set_markerfacecolor>` or `mfc` any matplotlib color

:method:`markerfacecoloralt <matplotlib.lines.Line2D.set_markerfacecoloralt>` or `mfcalt` any matplotlib color

:method:`markersize <matplotlib.lines.Line2D.set_markersize>` or `ms` float

:method:`markevery <matplotlib.lines.Line2D.set_markevery>` [None | int | length-2 tuple of int | slice | list/array of int | float | length-2 tuple of float]

:method:`path_effects <matplotlib.artist.Artist.set_path_effects>` unknown

:method:`picker <matplotlib.lines.Line2D.set_picker>` float distance in points or callable pick function ``fn(artist, event)``

:method:`pickradius <matplotlib.lines.Line2D.set_pickradius>` float distance in points

:method:`rasterized <matplotlib.artist.Artist.set_rasterized>` [True | False | None]

:method:`sketch_params <matplotlib.artist.Artist.set_sketch_params>` unknown

:method:`snap <matplotlib.artist.Artist.set_snap>` unknown

:method:`solid_capstyle <matplotlib.lines.Line2D.set_solid_capstyle>` ['butt' | 'round' | 'projecting']

:method:`solid_joinstyle <matplotlib.lines.Line2D.set_solid_joinstyle>` ['miter' | 'round' | 'bevel']


:method:`url <matplotlib.artist.Artist.set_url>` a url string

:method:`visible <matplotlib.artist.Artist.set_visible>` [True | False]

:method:`xdata <matplotlib.lines.Line2D.set_xdata>` 1D array

:method:`ydata <matplotlib.lines.Line2D.set_ydata>` 1D array

:method:`zorder <matplotlib.artist.Artist.set_zorder>` any number

================================================================================================================================================================================
See also:

loglog() For example code and figure.

matplotlib.pyplot.set_cmap(cmap)
Set the default colormap. Applies to the current image if any. See help(colormaps) for more information.

cmap must be a Colormap instance, or the name of a registered colormap.

See matplotlib.cm.register_cmap() and matplotlib.cm.get_cmap().

matplotlib.pyplot.setp(*args, **kwargs)
Set a property on an artist object.

matplotlib supports the use of setp() (“set property”) and getp() to set and get object properties, as well as to do introspection on the object. For example, to set the linestyle of a line to be dashed, you can do:

```python
>>> line, = plot([1,2,3])
>>> setp(line, linestyle='--')
```

If you want to know the valid types of arguments, you can provide the name of the property you want to set without a value:

```python
>>> setp(line, 'linestyle')
linestyle: [ '-' | '--' | ':' | 'steps' | 'None' ]
```

If you want to see all the properties that can be set, and their possible values, you can do:

```python
>>> setp(line)
... long output listing omitted
```

setp() operates on a single instance or a list of instances. If you are in query mode introspecting the possible values, only the first instance in the sequence is used. When actually setting values, all the instances will be set. e.g., suppose you have a list of two lines, the following will make both lines thicker and red:

```python
>>> x = arange(0,1.0,0.01)
>>> y1 = sin(2*pi*x)
>>> y2 = sin(4*pi*x)
>>> lines = plot(x, y1, x, y2)
>>> setp(lines, linewidth=2, color='r')
```

setp() works with the MATLAB style string/value pairs or with python kwargs. For example, the following are equivalent:

```python
>>> setp(lines, 'linewidth', 2, 'color', 'r')  # MATLAB style
>>> setp(lines, linewidth=2, color='r')     # python style
```

matplotlib.pyplot.show(*args, **kw)
Display a figure. When running in ipython with its pylab mode, display all figures and return to the ipython prompt.
In non-interactive mode, display all figures and block until the figures have been closed; in interactive mode it has no effect unless figures were created prior to a change from non-interactive to interactive mode (not recommended). In that case it displays the figures but does not block.

A single experimental keyword argument, block, may be set to True or False to override the blocking behavior described above.

```
matplotlib.pyplot.specgram(x, NFFT=None, Fs=None, Fc=None, detrend=None, window=None, noverlap=None, cmap=None, xextent=None, pad_to=None, sides=None, scale_by_freq=None, mode=None, scale=None, vmin=None, vmax=None, hold=None, data=None, **kwargs)
```

Plot a spectrogram.

Call signature:

```
specgram(x, NFFT=256, Fs=2, Fc=0, detrend=mlab.detrend_none,
         window=mlab.window_hanning, noverlap=128,
         cmap=None, xextent=None, pad_to=None, sides='default',
         scale_by_freq=None, mode='default', scale='default',
         **kwargs)
```

Compute and plot a spectrogram of data in x. Data are split into NFFT length segments and the spectrum of each section is computed. The windowing function window is applied to each segment, and the amount of overlap of each segment is specified with noverlap. The spectrogram is plotted as a colormap (using imshow).

**Parameters**

- **x** : 1-D array or sequence
  Array or sequence containing the data.

- **Fs** : scalar
  The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.

- **window** : callable or ndarray
  A function or a vector of length NFFT. To create window vectors see window_hanning(), window_none(), numpy.blackman(), numpy.hanning(), numpy.bartlett(), scipy.signal(), scipy.signal.get_window(), etc. The default is window_hanning(). If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

- **sides** : [‘default’ | ‘onesided’ | ‘twosided’]
  Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. ‘onesided’ forces the return of a one-sided spectrum, while ‘twosided’ forces two-sided.

- **pad_to** : integer
The number of points to which the data segment is padded when performing
the FFT. This can be different from NFFT, which specifies the number of data
points used. While not increasing the actual resolution of the spectrum (the
minimum distance between resolvable peaks), this can give more points in
the plot, allowing for more detail. This corresponds to the n parameter in the
call to fft(). The default is None, which sets pad_to equal to NFFT

NFFT : integer

The number of data points used in each block for the FFT. A power 2 is most
efficient. The default value is 256. This should NOT be used to get zero
padding, or the scaling of the result will be incorrect. Use pad_to for this
instead.

detrend : {'default', 'constant', 'mean', 'linear', 'none'} or callable

The function applied to each segment before fft-ing, designed to remove
the mean or linear trend. Unlike in MATLAB, where the detrend param-
eter is a vector, in matplotlib it is a function. The pylab module defines
detrend_none(), detrend_mean(), and detrend_linear(), but you can
use a custom function as well. You can also use a string to choose one of the
functions. 'default', 'constant', and 'mean' call detrend_mean(). 'linear'
calls detrend_linear(). 'none' calls detrend_none().

scale_by_freq : boolean, optional

Specifies whether the resulting density values should be scaled by the scaling
frequency, which gives density in units of Hz^-1. This allows for integra-
tion over the returned frequency values. The default is True for MATLAB
compatibility.

mode : [ 'default' | 'psd' | 'magnitude' | 'angle' | 'phase' ]

What sort of spectrum to use. Default is 'psd', which takes the power spectral
density. 'complex' returns the complex-valued frequency spectrum. 'mag-
nitude' returns the magnitude spectrum. 'angle' returns the phase spectrum
without unwrapping. 'phase' returns the phase spectrum with unwrapping.

noverlap : integer

The number of points of overlap between blocks. The default value is 128.

scale : [ 'default' | 'linear' | 'dB' ]

The scaling of the values in the spec. 'linear' is no scaling. 'dB' returns
the values in dB scale. When mode is 'psd', this is dB power (10 * log10).
Otherwise this is dB amplitude (20 * log10). 'default' is 'dB' if mode is 'psd'
or 'magnitude' and 'linear' otherwise. This must be 'linear' if mode is 'angle'
or 'phase'.

Fc : integer

The center frequency of x (defaults to 0), which offsets the x extents of the
plot to reflect the frequency range used when a signal is acquired and then
filtered and downsampled to baseband.
**cmap** :

A `matplotlib.colors.Colormap` instance; if `None`, use default determined by `rc`

**xextent** : [None | (xmin, xmax)]

The image extent along the x-axis. The default sets `xmin` to the left border of the first bin (`spectrum` column) and `xmax` to the right border of the last bin. Note that for `noverlap>0` the width of the bins is smaller than those of the segments.

**kwargs** :

Additional kwargs are passed on to `imshow` which makes the specgram image

Returns **spectrum** : 2-D array

Columns are the periodograms of successive segments.

**freqs** : 1-D array

The frequencies corresponding to the rows in `spectrum`.

**t** : 1-D array

The times corresponding to midpoints of segments (i.e., the columns in `spectrum`).

**im** : instance of class `AxesImage`

The image created by `imshow` containing the spectrogram

See also:

`psd()` differs in the default overlap; in returning the mean of the segment periodograms; in not returning times; and in generating a line plot instead of colormap.

`magnitude_spectrum()` A single spectrum, similar to having a single segment when `mode` is ‘magnitude’. Plots a line instead of a colormap.

`angle_spectrum()` A single spectrum, similar to having a single segment when `mode` is ‘angle’. Plots a line instead of a colormap.

`phase_spectrum()` A single spectrum, similar to having a single segment when `mode` is ‘phase’. Plots a line instead of a colormap.

In addition to the above described arguments, this function can take a `data` keyword argument. If such a `data` argument is given, the following arguments are replaced by `data[<arg>]`: * All arguments with the following names: ‘x’.

Notes

`detrend` and `scale_by_freq` only apply when `mode` is set to ‘psd’
Examples

```python
import matplotlib.pyplot
import numpy

matplotlib.pyplot.spectral()
set the default colormap to spectral and apply to current image if any. See help(colormaps) for more information

matplotlib.pyplot.spring()
set the default colormap to spring and apply to current image if any. See help(colormaps) for more information

matplotlib.pyplot.spy(Z, precision=0, marker=None, markersize=None, aspect='equal', **kwargs)
Plot the sparsity pattern on a 2-D array.

spy(Z) plots the sparsity pattern of the 2-D array Z.

Parameters Z : sparse array (n, m)
The array to be plotted.

precision : float, optional, default: 0

If precision is 0, any non-zero value will be plotted; else, values of |Z| > precision will be plotted.

For scipy.sparse.spmatrix instances, there is a special case: if precision is ‘present’, any value present in the array will be plotted, even if it is
```
identically zero.

**origin**: ['"upper", "lower"], optional, default: "upper"

Place the [0,0] index of the array in the upper left or lower left corner of the axes.

**aspect**: [‘auto’ | ‘equal’ | scalar], optional, default: “equal”

If ‘equal’, and `extent` is None, changes the axes aspect ratio to match that of the image. If `extent` is not None, the axes aspect ratio is changed to match that of the extent.

If ‘auto’, changes the image aspect ratio to match that of the axes.

If None, default to rc `image.aspect` value.

Two plotting styles are available: image or marker. Both are available for full arrays, but only the marker style works for `:class:`scipy.sparse.spmatrix` instances.

If *marker* and *markersize* are *None*, an image will be returned and any remaining kwargs are passed to

:func:`matplotlib.pyplot.imshow`; else, a

:func:`matplotlib.pyplot.plot` object will be returned with

the value of marker determining the marker type, and any remaining kwargs passed to the


If *marker* and *markersize* are *None*, useful kwargs include:

*  *cmap*
*  *alpha*

See also:

`imshow` for image options.

`plot` for plotting options

`matplotlib.pyplot.stackplot`(*x*, *args*, **kwargs)

Draws a stacked area plot.

*x*: 1d array of dimension N

*y*: [2d array of dimension MxN, OR any number 1d arrays each of dimension] 1xN. The data is assumed to be unstacked. Each of the following calls is legal:

```
stackplot(x, y)  # where y is MxN
stackplot(x, y1, y2, y3, y4)  # where y1, y2, y3, y4, are all 1xNm
```
Keyword arguments:

**baseline** ["zero", 'sym', 'wiggle', 'weighted_wiggle]] Method used to calculate the baseline. ‘zero’ is just a simple stacked plot. ‘sym’ is symmetric around zero and is sometimes called ThemeRiver. ‘wiggle’ minimizes the sum of the squared slopes. ‘weighted_wiggle’ does the same but weights to account for size of each layer. It is also called Streamgraph-layout. More details can be found at http://leebyron.com/streamgraph/.

**labels** : A list or tuple of labels to assign to each data series.

**colors** [A list or tuple of colors. These will be cycled through and] used to colour the stacked areas. All other keyword arguments are passed to fill_between()

Returns r : A list of PolyCollection, one for each element in the stacked area plot.

```
matplotlib.pyplot.stem(*args, **kwargs)
Create a stem plot.

Call signatures:

stem(y, linefmt='b-', markerfmt='bo', basefmt='r-')
stem(x, y, linefmt='b-', markerfmt='bo', basefmt='r-')
```

A stem plot plots vertical lines (using linefmt) at each x location from the baseline to y, and places a marker there using markerfmt. A horizontal line at 0 is is plotted using basefmt.

If no x values are provided, the default is (0, 1, ..., len(y) - 1)

Return value is a tuple (markerline, stemlines, baseline).

See also:

This document for details.

Example:
Note: In addition to the above described arguments, this function can take a data keyword argument. If such a data argument is given, the following arguments are replaced by data[<arg>]:

- All positional and all keyword arguments.

matplotlib.pyplot.step(x, y, *args, **kwargs)

Make a step plot.

Parameters:

- x : array_like
  1-D sequence, and it is assumed, but not checked, that it is uniformly increasing.

- y : array_like
  1-D sequence, and it is assumed, but not checked, that it is uniformly increasing.

Returns:

- list
  List of lines that were added.

Other Parameters:

- where : [ ‘pre’ | ‘post’ | ‘mid’ ]
  If ‘pre’ (the default), the interval from x[i] to x[i+1] has level y[i+1].
If ‘post’, that interval has level $y[i]$.  
If ‘mid’, the jumps in $y$ occur half-way between the $x$-values.

**Notes**

Additional parameters are the same as those for `plot()`.

---

**Note:** In addition to the above described arguments, this function can take a `data` keyword argument. If such a `data` argument is given, the following arguments are replaced by `data[<arg>]`:

- All arguments with the following names: ‘x’, ‘y’.  

```python
import matplotlib.pyplot as plt

plt.streamplot(x, y, u, v, density=1, linewidth=None, color=None,
               cmap=None, norm=None, arrowstyle='->',
               minlength=0.1, transform=None, zorder=None,
               start_points=None, hold=None, data=None)
```

Draws streamlines of a vector flow.

- $x$, $y$ [1d arrays] an *evenly spaced* grid.

- $u$, $v$ [2d arrays] $x$ and $y$-velocities. Number of rows should match length of $y$, and the number of columns should match $x$.

- **density** [float or 2-tuple] Controls the closeness of streamlines. When density = 1, the domain is divided into a 30x30 grid—density linearly scales this grid. Each cell in the grid can have, at most, one traversing streamline. For different densities in each direction, use [density$_x$, density$_y$].

- **linewidth** [numeric or 2d array] vary linewidth when given a 2d array with the same shape as velocities.

- **color** [matplotlib color code, or 2d array] Streamline color. When given an array with the same shape as velocities, color values are converted to colors using `cmap`.

- **cmap** [Colormap] Colormap used to plot streamlines and arrows. Only necessary when using an array input for `color`.

- **norm** [Normalize] Normalize object used to scale luminance data to 0, 1. If None, stretch (min, max) to (0, 1). Only necessary when color is an array.

- **arrowsize** [float] Factor scale arrow size.


- **minlength** [float] Minimum length of streamline in axes coordinates.

- **start_points**: **Nx2 array** Coordinates of starting points for the streamlines. In data coordinates, the same as the $x$ and $y$ arrays.

- **zorder** [int] any number

Returns:

45.1. matplotlib.pyplot 1559
**stream_container** [StreamplotSet] Container object with attributes

- lines: `matplotlib.collections.LineCollection` of streamlines
- arrows: collection of `matplotlib.patches.FancyArrowPatch` objects representing arrows half-way along stream lines.

This container will probably change in the future to allow changes to the colormap, alpha, etc. for both lines and arrows, but these changes should be backward compatible.

**matplotlib.pyplot.subplot(*args, **kwargs)**

Return a subplot axes positioned by the given grid definition.

Typical call signature:

```python
subplot(nrows, ncols, plot_number)
```

Where `nrows` and `ncols` are used to notionally split the figure into `nrows * ncols` sub-axes, and `plot_number` is used to identify the particular subplot that this function is to create within the notional grid. `plot_number` starts at 1, increments across rows first and has a maximum of `nrows * ncols`.

In the case when `nrows`, `ncols` and `plot_number` are all less than 10, a convenience exists, such that the a 3 digit number can be given instead, where the hundreds represent `nrows`, the tens represent `ncols` and the units represent `plot_number`. For instance:

```python
subplot(211)
```

produces a subaxes in a figure which represents the top plot (i.e. the first) in a 2 row by 1 column notional grid (no grid actually exists, but conceptually this is how the returned subplot has been positioned).

**Note:** Creating a subplot will delete any pre-existing subplot that overlaps with it beyond sharing a boundary:

```python
import matplotlib.pyplot as plt
plt.plot([1,2,3])  # plot a line, implicitly creating a subplot(111)
plt.plot([1,2,3])  # now create a subplot which represents the top plot of a grid
# with 2 rows and 1 column. Since this subplot will overlap the
# first, the plot (and its axes) previously created, will be removed
plt.subplot(211)
plt.plot(range(12))
plt.subplot(212, facecolor='y')  # creates 2nd subplot with yellow background
```

If you do not want this behavior, use the `add_subplot()` method or the `axes()` function instead.

Keyword arguments:

- `facecolor`: The background color of the subplot, which can be any valid color specifier. See `matplotlib.colors` for more information.
**polar**: A boolean flag indicating whether the subplot plot should be a polar projection. Defaults to `False`.

**projection**: A string giving the name of a custom projection to be used for the subplot. This projection must have been previously registered. See `matplotlib.projections`.

See also:

- `axes()` For additional information on `axes()` and `subplot()` keyword arguments.

- `examples/pie_and_polar_charts/polar_scatter_demo.py` For an example

Example:

![A tale of 2 subplots](image)

matplotlib.pyplot.subplot2grid(shape, loc, rowspan=1, colspan=1)

Create a subplot in a grid. The grid is specified by `shape`, at location of `loc`, spanning `rowspan`, `colspan` cells in each direction. The index for `loc` is 0-based.

```
subplot2grid(shape, loc, rowspan=1, colspan=1)
```

is identical to

```
gridspec=GridSpec(shape[0], shape[1])
subplotspec=gridspec.new_subplotspec(loc, rowspan, colspan)
subplot(subplotspec)
```
**Pyplot**

```python
matplot.pyplot.subplot_tool(targetfig=None)
```

Launch a subplot tool window for a figure.

A `matplotlib.widgets.SubplotTool` instance is returned.

```python
matplot.pyplot.subplots(nrows=1, ncols=1, sharex=False, sharey=False, squeeze=True, subplot_kw=None, gridspec_kw=None, **fig_kw)
```

Create a figure and a set of subplots.

This utility wrapper makes it convenient to create common layouts of subplots, including the enclosing figure object, in a single call.

**Parameters**

- **nrows, ncols**: int, optional, default: 1
  - Number of rows/columns of the subplot grid.

- **sharex, sharey**: bool or {'none', 'all', 'row', 'col'}, default: False
  - Controls sharing of properties among x (sharex) or y (sharey) axes:
    - True or ‘all’: x- or y-axis will be shared among all subplots.
    - False or ‘none’: each subplot x- or y-axis will be independent.
    - ‘row’: each subplot row will share an x- or y-axis.
    - ‘col’: each subplot column will share an x- or y-axis.

When subplots have a shared x-axis along a column, only the x tick labels of the bottom subplot are visible. Similarly, when subplots have a shared y-axis along a row, only the y tick labels of the first column subplot are visible.

- **squeeze**: bool, optional, default: True
  - If True, extra dimensions are squeezed out from the returned Axes object:
    - if only one subplot is constructed (nrows=ncols=1), the resulting single Axes object is returned as a scalar.
    - for Nx1 or 1xN subplots, the returned object is a 1D numpy object array of Axes objects are returned as numpy 1D arrays.
    - for NxM, subplots with N>1 and M>1 are returned as a 2D arrays.
  - If False, no squeezing at all is done: the returned Axes object is always a 2D array containing Axes instances, even if it ends up being 1x1.

- **subplot_kw**: dict, optional
  - Dict with keywords passed to the `add_subplot()` call used to create each subplot.

- **gridspec_kw**: dict, optional
  - Dict with keywords passed to the `GridSpec` constructor used to create the grid the subplots are placed on.

- **fig_kw**: dict
  - All additional keyword arguments are passed to the `figure()` call.
Returns **fig**: `matplotlib.figure.Figure` object

**ax**: Axes object or array of Axes objects.

ax can be either a single `matplotlib.axes.Axes` object or an array of Axes objects if more than one subplot was created. The dimensions of the resulting array can be controlled with the `squeeze` keyword, see above.

See also:

*figure*, *subplot*

Examples

First create some toy data:

```python
>>> x = np.linspace(0, 2*np.pi, 400)
>>> y = np.sin(x**2)
```

Creates just a figure and only one subplot

```python
>>> fig, ax = plt.subplots()
>>> ax.plot(x, y)
>>> ax.set_title('Simple plot')
```

Creates two subplots and unpacks the output array immediately

```python
>>> f, (ax1, ax2) = plt.subplots(1, 2, sharey=True)
>>> ax1.plot(x, y)
>>> ax1.set_title('Sharing Y axis')
>>> ax2.scatter(x, y)
```

Creates four polar axes, and accesses them through the returned array

```python
>>> fig, axes = plt.subplots(2, 2, subplot_kw=dict(polar=True))
>>> axes[0, 0].plot(x, y)
>>> axes[1, 1].scatter(x, y)
```

Share a X axis with each column of subplots

```python
>>> plt.subplots(2, 2, sharex='col')
```

Share a Y axis with each row of subplots

```python
>>> plt.subplots(2, 2, sharey='row')
```

Share both X and Y axes with all subplots

```python
>>> plt.subplots(2, 2, sharex='all', sharey='all')
```

Note that this is the same as
Matplotlib, Release 2.0.2

```python
>>> plt.subplots(2, 2, sharex=True, sharey=True)
```

**matplotlib.pyplot.subplots_adjust(**args**, **kwargs**)  
Tune the subplot layout.

call signature:

```python
subplots_adjust(left=None, bottom=None, right=None, top=None, 
    wspace=None, hspace=None)
```

The parameter meanings (and suggested defaults) are:

- **left** = 0.125  
  # the left side of the subplots of the figure

- **right** = 0.9  
  # the right side of the subplots of the figure

- **bottom** = 0.1  
  # the bottom of the subplots of the figure

- **top** = 0.9  
  # the top of the subplots of the figure

- **wspace** = 0.2  
  # the amount of width reserved for blank space between subplots,  
  # expressed as a fraction of the average axis width

- **hspace** = 0.2  
  # the amount of height reserved for white space between subplots,  
  # expressed as a fraction of the average axis height

The actual defaults are controlled by the rc file

**matplotlib.pyplot.summer()**  
set the default colormap to summer and apply to current image if any. See help(colormaps) for more information

**matplotlib.pyplot.suptitle(**args**, **kwargs**)**  
Add a centered title to the figure.

kwargs are **matplotlib.text.Text** properties. Using figure coordinates, the defaults are:

- **x** [0.5] The x location of the text in figure coords
- **y** [0.98] The y location of the text in figure coords

- **horizontalalignment** ['center'] The horizontal alignment of the text

- **verticalalignment** ['top'] The vertical alignment of the text

If the **fontproperties** keyword argument is given then the rcParams defaults for **fontsize** (figure.titlesize) and **fontweight** (figure.titleweight) will be ignored in favour of the **FontProperties** defaults.

A **matplotlib.text.Text** instance is returned.

Example:

```python
fig.suptitle('this is the figure title', fontsize=12)
```

**matplotlib.pyplot.switch_backend(newbackend)**  
Switch the default backend. This feature is **experimental**, and is only expected to work switching to an image backend. e.g., if you have a bunch of PostScript scripts that you want to run from an interactive ipython session, you may want to switch to the PS backend before running them to avoid
having a bunch of GUI windows popup. If you try to interactively switch from one GUI backend to
another, you will explode.

Calling this command will close all open windows.

```
matplotlib.pyplot.table(**kwargs)
```

Add a table to the current axes.

Call signature:

```
table(cellText=None, cellColours=None, cellLoc='right', colWidths=None, rowLabels=None, rowColours=None, rowLoc='left', colLabels=None, colColours=None, colLoc='center', loc='bottom', bbox=None):
```

Returns a matplotlib.table.Table instance. Either cellText or cellColours must be pro-
vided. For finer grained control over tables, use the Table class and add it to the axes with
`add_table()`.

Thanks to John Gill for providing the class and table.

`**kwargs` control the Table properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>agg_filter</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>alpha</code></td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td><code>animated</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>axes</code></td>
<td>an Axes instance</td>
</tr>
<tr>
<td><code>clip_box</code></td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td><code>clip_on</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>clip_path</code></td>
<td>[(Path, Transform)]</td>
</tr>
<tr>
<td><code>contains</code></td>
<td>a callable function</td>
</tr>
<tr>
<td><code>figure</code></td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td><code>fontsize</code></td>
<td>a float in points</td>
</tr>
<tr>
<td><code>gid</code></td>
<td>an id string</td>
</tr>
<tr>
<td><code>label</code></td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td><code>path_effects</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>picker</code></td>
<td>[None</td>
</tr>
<tr>
<td><code>rasterized</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>sketch_params</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>snap</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>transform</code></td>
<td>Transform instance</td>
</tr>
<tr>
<td><code>url</code></td>
<td>a url string</td>
</tr>
<tr>
<td><code>visible</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>zorder</code></td>
<td>any number</td>
</tr>
</tbody>
</table>

```
matplotlib.pyplot.text(x, y, s, fontdict=None, withdash=False, **kwargs)
```

Add text to the axes.

Add text in string `s` to axis at location `x`, `y`, data coordinates.
**Parameters** `x, y` : scalars
  data coordinates

`s` : string
  text

`fontdict` : dictionary, optional, default: None
  A dictionary to override the default text properties. If fontdict is None, the
defaults are determined by your rc parameters.

`withdash` : boolean, optional, default: False
  Creates a `TextWithDash` instance instead of a `Text` instance.

**Other Parameters** `kwargs` : `Text` properties.
  Other miscellaneous text parameters.

**Examples**

Individual keyword arguments can be used to override any given parameter:

```python
>>> text(x, y, s, fontsize=12)
```

The default transform specifies that text is in data coords, alternatively, you can specify text in axis
coords (0,0 is lower-left and 1,1 is upper-right). The example below places text in the center of the
axes:

```python
>>> text(0.5, 0.5, 'matplotlib', horizontalalignment='center',
        verticalalignment='center',
        transform=ax.transAxes)
```

You can put a rectangular box around the text instance (e.g., to set a background color) by using the
keyword `bbox`. `bbox` is a dictionary of `Rectangle` properties. For example:

```python
>>> text(x, y, s, bbox=dict(facecolor='red', alpha=0.5))
```

`matplotlib.pyplot.thetagrids(*args, **kwargs)`
  Get or set the theta locations of the gridlines in a polar plot.

  If no arguments are passed, return a tuple `(lines, labels)` where `lines` is an array of radial gridlines
  (`Line2D` instances) and `labels` is an array of tick labels (`Text` instances):

  ```python
  lines, labels = theagrids()
  ```

  Otherwise the syntax is:

  ```python
  lines, labels = theagrids(angles, labels=None, fmt='%d', frac = 1.1)
  ```

  set the angles at which to place the theta grids (these gridlines are equal along the theta dimension).
angles is in degrees.

labels, if not None, is a len(angles) list of strings of the labels to use at each angle.

If labels is None, the labels will be fmt%angle.

frac is the fraction of the polar axes radius at which to place the label (1 is the edge). e.g., 1.05 is outside the axes and 0.95 is inside the axes.

Return value is a list of tuples (lines, labels):

- lines are Line2D instances
- labels are Text instances.

Note that on input, the labels argument is a list of strings, and on output it is a list of Text instances.

Examples:

```python
# set the locations of the radial gridlines and labels
lines, labels = thetagrids( range(45,360,90) )

# set the locations and labels of the radial gridlines and labels
lines, labels = thetagrids( range(45,360,90), ('NE', 'NW', 'SW', 'SE') )
```

matplotlib.pyplot.tick_params(axis='both', **kwargs)

Change the appearance of ticks and tick labels.

**Parameters**

- axis : {'x', 'y', 'both'}, optional
  - Which axis to apply the parameters to.

**Other Parameters**

- axis : {'x', 'y', 'both'}
  - Axis on which to operate; default is ‘both’.
- reset : bool
  - If True, set all parameters to defaults before processing other keyword arguments. Default is False.
- which : {'major', 'minor', 'both'}
  - Default is ‘major’; apply arguments to which ticks.
- direction : {'in', 'out', 'inout'}
  - Puts ticks inside the axes, outside the axes, or both.
- length : float
  - Tick length in points.
- width : float
  - Tick width in points.
- color : color
  - Tick color; accepts any mpl color spec.
pad : float
    Distance in points between tick and label.

labelsize : float or str
    Tick label font size in points or as a string (e.g., ‘large’).

labelcolor : color
    Tick label color; mpl color spec.

colors : color
    Changes the tick color and the label color to the same value: mpl color spec.

zorder : float
    Tick and label zorder.

bottom, top, left, right : bool or {'on', 'off'}
    controls whether to draw the respective ticks.

labelbottom, labeltop, labelleft, labelright : bool or {'on', 'off'}
    controls whether to draw the respective tick labels.

Examples

Usage

```python
ax.tick_params(direction='out', length=6, width=2, colors='r')
```

This will make all major ticks be red, pointing out of the box, and with dimensions 6 points by 2 points. Tick labels will also be red.

```python
matplotlib.pyplot.ticklabel_format(**kwargs)
```
Change the `ScalarFormatter` used by default for linear axes.

Optional keyword arguments:
**style**

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>style</code></td>
<td>[ <code>sci</code> (or <code>scientific</code>)</td>
</tr>
</tbody>
</table>

**scilimits**

| (m, n), pair of integers; if `style` is `sci`, scientific notation will be used for numbers outside the range $10^m$ to $10^n$. Use (0,0) to include all numbers. |

**useOffset**

| [ `True` | `False` | offset]; if True, the offset will be calculated as needed; if False, no offset will be used; if a numeric offset is specified, it will be used. |

**axis**

| [ `x` | `y` | `both` ] |

Only the major ticks are affected. If the method is called when the `ScalarFormatter` is not the `Formatter` being used, an `AttributeError` will be raised.

**matplotlib.pyplot.tight_layout**(pad=1.08, h_pad=None, w_pad=None, rect=None)

Automatically adjust subplot parameters to give specified padding.

Parameters:

**pad** [float] padding between the figure edge and the edges of subplots, as a fraction of the font-size.

**h_pad, w_pad** [float] padding (height/width) between edges of adjacent subplots. Defaults to `pad_inches`.

**rect** [if rect is given, it is interpreted as a rectangle] (left, bottom, right, top) in the normalized figure coordinate that the whole subplots area (including labels) will fit into. Default is (0, 0, 1, 1).

**matplotlib.pyplot.title**(s, *args, **kwargs)

Set a title of the current axes.

Set one of the three available axes titles. The available titles are positioned above the axes in the center, flush with the left edge, and flush with the right edge.

See also:

See `text()` for adding text to the current axes

**Parameters**

**label** : str

Text to use for the title

**fontdict** : dict

A dictionary controlling the appearance of the title text, the default `fontdict` is:

```python
dict({'fontsize': rcParams['axes.titlesize'], 'fontweight': rcParams['axes.titleweight'], 'verticalalignment': 'baseline', 'horizontalalignment': loc})
```

**loc** : ```{'center', 'left', 'right'}``` str, optional
Which title to set, defaults to ‘center’

Returns text : Text

The matplotlib text instance representing the title

Other Parameters kwargs : text properties

Other keyword arguments are text properties, see Text for a list of valid text properties.

matplotlib.pyplot.tricontour(*args, **kwargs)

Draw contours on an unstructured triangular grid. tricontour() and tricontourf() draw contour lines and filled contours, respectively. Except as noted, function signatures and return values are the same for both versions.

The triangulation can be specified in one of two ways; either:

tricontour(triangulation, ...)

where triangulation is a matplotlib.tri.Triangulation object, or

tricontour(x, y, ...)
tricontour(x, y, triangles, ...)
tricontour(x, y, triangles=triangles, ...)
tricontour(x, y, mask=mask, ...)
tricontour(x, y, triangles, mask=mask, ...)

in which case a Triangulation object will be created. See Triangulation for a explanation of these possibilities.

The remaining arguments may be:

tricontour(..., Z)

where Z is the array of values to contour, one per point in the triangulation. The level values are chosen automatically.

tricontour(..., Z, N)

contour $N$ automatically-chosen levels.

tricontour(..., Z, V)

draw contour lines at the values specified in sequence $V$, which must be in increasing order.

tricontourf(..., Z, V)

fill the $(\text{len}(V)-1)$ regions between the values in $V$, which must be in increasing order.

tricontour(Z, **kwargs)

Use keyword args to control colors, linewidth, origin, cmap ... see below for more details.
C = tricontour(...) returns a TriContourSet object.

Optional keyword arguments:

**colors**: [ *None* | string | (mpl_colors) ] If *None*, the colormap specified by cmap will be used.

If a string, like ‘r’ or ‘red’, all levels will be plotted in this color.

If a tuple of matplotlib color args (string, float, rgb, etc), different levels will be plotted in different colors in the order specified.

**alpha**: float The alpha blending value

**cmap**: [ *None* | Colormap ] A cm *Colormap* instance or *None*. If *cmap* is *None* and *colors* is *None*, a default Colormap is used.

**norm**: [ *None* | Normalize ] A *matplotlib.colors.Normalize* instance for scaling data values to colors. If *norm* is *None* and *colors* is *None*, the default linear scaling is used.

**levels** [level0, level1, ..., leveln] A list of floating point numbers indicating the level curves to draw, in increasing order; e.g., to draw just the zero contour pass levels=[0]

**origin**: [ *None* | ‘upper’ | ‘lower’ | ‘image’ ] If *None*, the first value of Z will correspond to the lower left corner, location (0,0). If ‘image’, the rc value for image.origin will be used.

This keyword is not active if X and Y are specified in the call to contour.

**extent**: [ *None* | (x0,x1,y0,y1) ]

If *origin* is not *None*, then *extent* is interpreted as in *matplotlib.pyplot.imshow()*: it gives the outer pixel boundaries. In this case, the position of Z[0,0] is the center of the pixel, not a corner. If *origin* is *None*, then (x0, y0) is the position of Z[0,0], and (x1, y1) is the position of Z[-1,-1].

This keyword is not active if X and Y are specified in the call to contour.

**locator**: [ *None* | ticker.Locator subclass ] If *locator* is *None*, the default *MaxNLocator* is used. The locator is used to determine the contour levels if they are not given explicitly via the *V* argument.

**extend**: [ ‘neither’ | ‘both’ | ‘min’ | ‘max’ ] Unless this is ‘neither’, contour levels are automatically added to one or both ends of the range so that all data are included. These added ranges are then mapped to the special colormap values which default to the ends of the colormap range, but can be set via *matplotlib.colors.Colormap.set_under()* and *matplotlib.colors.Colormap.set_over()* methods.

**xunits, yunits**: [ *None* | registered units ] Override axis units by specifying an instance of a *matplotlib.units.ConversionInterface*.

tricontour-only keyword arguments:

**linewidths**: [ *None* | number | tuple of numbers ] If *linewidths* is *None*, the default width in *lines.linewidth* in *matplotlibrc* is used.
If a number, all levels will be plotted with this linewidth.
If a tuple, different levels will be plotted with different linewidths in the order specified

**linestyles:** [None | ‘solid’ | ‘dashed’ | ‘dashdot’ | ‘dotted’] If linestyles is None, the ‘solid’ is used.

linestyles can also be an iterable of the above strings specifying a set of linestyles to be used. If this iterable is shorter than the number of contour levels it will be repeated as necessary.

If contour is using a monochrome colormap and the contour level is less than 0, then the linestyle specified in contour.negative_linestyle in matplotlibrc will be used.

tricontourf-only keyword arguments:

**antialiased:** [True | False] enable antialiasing

Note: tricontourf fills intervals that are closed at the top; that is, for boundaries z₁ and z₂, the filled region is:

| z₁ < z <= z₂ |

There is one exception: if the lowest boundary coincides with the minimum value of the z array, then that minimum value will be included in the lowest interval.

**Examples:**
Matplotlib.pyplot.tricontourf(*args, **kwargs)

Draw contours on an unstructured triangular grid. tricontour() and tricontourf() draw contour lines and filled contours, respectively. Except as noted, function signatures and return values are the same for both versions.

The triangulation can be specified in one of two ways; either:

tricontour(triangulation, ...)

where triangulation is a matplotlib.tri.Triangulation object, or

tricontour(x, y, ...)  
tricontour(x, y, triangles, ...)  
tricontour(x, y, triangles=triangles, ...)  
tricontour(x, y, mask=mask, ...)  
tricontour(x, y, triangles, mask=mask, ...)

in which case a Triangulation object will be created. See Triangulation for a explanation of these possibilities.

The remaining arguments may be:

tricontour(..., Z)

where Z is the array of values to contour, one per point in the triangulation. The level values are chosen automatically.
tricontour(..., Z, N)

draw contour lines at the values specified in sequence V, which must be in increasing order.

tricontourf(..., Z, V)

fill the (len(V)-1) regions between the values in V, which must be in increasing order.

tricontour(Z, **kwargs)

Use keyword args to control colors, linewidth, origin, cmap ... see below for more details.

C = tricontour(...) returns a TriContourSet object.

Optional keyword arguments:

colors: [ None | string | (mpl_colors) ] If None, the colormap specified by cmap will be used.

If a string, like ‘r’ or ‘red’, all levels will be plotted in this color.

If a tuple of matplotlib color args (string, float, rgb, etc), different levels will be plotted in different colors in the order specified.

alpha: float The alpha blending value

cmap: [ None | Colormap ] A cm Colormap instance or None. If cmap is None and colors is None, a default Colormap is used.

norm: [ None | Normalize ] A matplotlib.colors.Normalize instance for scaling data values to colors. If norm is None and colors is None, the default linear scaling is used.

levels [level0, level1, ..., leveln] A list of floating point numbers indicating the level curves to draw, in increasing order; e.g., to draw just the zero contour pass levels=[0]

origin: [ None | ‘upper’ | ‘lower’ | ‘image’ ] If None, the first value of Z will correspond to the lower left corner, location (0,0). If ‘image’, the rc value for image.origin will be used.

This keyword is not active if X and Y are specified in the call to contour.

extent: [ None | (x0,x1,y0,y1) ]

If origin is not None, then extent is interpreted as in matplotlib.pyplot.imshow(): it gives the outer pixel boundaries. In this case, the position of Z[0,0] is the center of the pixel, not a corner. If origin is None, then (x0, y0) is the position of Z[0,0], and (x1, y1) is the position of Z[-1,-1].

This keyword is not active if X and Y are specified in the call to contour.
**locator**: [None | ticker.Locator subclass] If locator is None, the default MaxNLocator is used. The locator is used to determine the contour levels if they are not given explicitly via the V argument.

**extend**: [‘neither’ | ‘both’ | ‘min’ | ‘max’] Unless this is ‘neither’, contour levels are automatically added to one or both ends of the range so that all data are included. These added ranges are then mapped to the special colormap values which default to the ends of the colormap range, but can be set via matplotlib.colors.Colormap.set_under() and matplotlib.colors.Colormap.set_over() methods.

**xunits, yunits**: [None | registered units] Override axis units by specifying an instance of a matplotlib.units.ConversionInterface.

tricontour-only keyword arguments:

**linewidths**: [None | number | tuple of numbers] If linewidths is None, the default width in lines.linewidth in matplotlibrc is used.

If a number, all levels will be plotted with this linewidth.

If a tuple, different levels will be plotted with different linewidths in the order specified

**linestyles**: [None | ‘solid’ | ‘dashed’ | ‘dashdot’ | ‘dotted’] If linestyles is None, the ‘solid’ is used.

linestyles can also be an iterable of the above strings specifying a set of linestyles to be used. If this iterable is shorter than the number of contour levels it will be repeated as necessary.

If contour is using a monochrome colormap and the contour level is less than 0, then the linestyle specified in contour.negative_linestyle in matplotlibrc will be used.

tricontourf-only keyword arguments:

**antialiased**: [True | False] enable antialiasing

Note: tricontourf fills intervals that are closed at the top; that is, for boundaries z1 and z2, the filled region is:

\[ z1 < z \leq z2 \]

There is one exception: if the lowest boundary coincides with the minimum value of the z array, then that minimum value will be included in the lowest interval.

**Examples:**
Contour plot of Delaunay triangulation
matplotlib.pyplot.tripcolor(*args, **kwargs)

Create a pseudocolor plot of an unstructured triangular grid.

The triangulation can be specified in one of two ways; either:

```python
tripcolor(triangulation, ...)
```

where triangulation is a `matplotlib.tri.Triangulation` object, or

```python
tripcolor(x, y, ...)  
tripcolor(x, y, triangles, ...)  
tripcolor(x, y, triangles=triangles, ...)  
tripcolor(x, y, mask=mask, ...)  
tripcolor(x, y, triangles, mask=mask, ...)  
```

in which case a Triangulation object will be created. See `Triangulation` for a explanation of these possibilities.

The next argument must be C, the array of color values, either one per point in the triangulation if color values are defined at points, or one per triangle in the triangulation if color values are defined at triangles. If there are the same number of points and triangles in the triangulation it is assumed that color values are defined at points; to force the use of color values at triangles use the kwarg `facecolors*={C instead of just *C`.

`shading` may be ‘flat’ (the default) or ‘gouraud’. If `shading` is ‘flat’ and C values are defined at points, the color values used for each triangle are from the mean C of the triangle’s three points. If `shading`
is ‘gouraud’ then color values must be defined at points.

The remaining kwargs are the same as for `pcolor()`.

Example:
tripcolor of Delaunay triangulation, gouraud shading
```
matplotlib.pyplot.triplot(*args, **kwargs)
```

Draw a unstructured triangular grid as lines and/or markers.

The triangulation to plot can be specified in one of two ways; either:

```
triplot(triangulation, ...)  
```

where triangulation is a `matplotlib.tri.Triangulation` object, or

```
triplot(x, y, ...)  
triplot(x, y, triangles, ...)  
triplot(x, y, triangles=triangles, ...)  
triplot(x, y, mask=mask, ...)  
triplot(x, y, triangles, mask=mask, ...)  
```

in which case a Triangulation object will be created. See `Triangulation` for an explanation of these possibilities.

The remaining args and kwargs are the same as for `plot()`.

Return a list of 2 `Line2D` containing respectively:

- the lines plotted for triangles edges
- the markers plotted for triangles nodes

**Example:**
triplot of Delaunay triangulation
Matplotlib.pyplot.twinx(ax=None)

Make a second axes that shares the x-axis. The new axes will overlay ax (or the current axes if ax is None). The ticks for ax2 will be placed on the right, and the ax2 instance is returned.

See also:

two_scales.py For an example

Matplotlib.pyplot.twiny(ax=None)

Make a second axes that shares the y-axis. The new axis will overlay ax (or the current axes if ax is None). The ticks for ax2 will be placed on the top, and the ax2 instance is returned.

Matplotlib.pyplot.uninstall_repl_displayhook()

Uninstalls the matplotlib display hook.

Matplotlib.pyplot.violinplot(dataset, positions=None, vert=True, widths=0.5, showmeans=False, showextrema=True, showmedians=False, points=100, bw_method=None, hold=None, data=None)

Make a violin plot.

Make a violin plot for each column of dataset or each vector in sequence dataset. Each filled area extends to represent the entire data range, with optional lines at the mean, the median, the minimum, and the maximum.

**Parameters**

dataset : Array or a sequence of vectors.
The input data.

**positions**: array-like, default = [1, 2, ..., n]

Sets the positions of the violins. The ticks and limits are automatically set to match the positions.

**vert**: bool, default = True.

If true, creates a vertical violin plot. Otherwise, creates a horizontal violin plot.

**widths**: array-like, default = 0.5

Either a scalar or a vector that sets the maximal width of each violin. The default is 0.5, which uses about half of the available horizontal space.

**showmeans**: bool, default = False

If True, will toggle rendering of the means.

**showextrema**: bool, default = True

If True, will toggle rendering of the extrema.

**showmedians**: bool, default = False

If True, will toggle rendering of the medians.

**points**: scalar, default = 100

Defines the number of points to evaluate each of the gaussian kernel density estimations at.

**bw_method**: str, scalar or callable, optional

The method used to calculate the estimator bandwidth. This can be ‘scott’, ‘silverman’, a scalar constant or a callable. If a scalar, this will be used directly as kde.factor. If a callable, it should take a GaussianKDE instance as its only parameter and return a scalar. If None (default), ‘scott’ is used.

**Returns result**: dict

A dictionary mapping each component of the violinplot to a list of the corresponding collection instances created. The dictionary has the following keys:

- **bodies**: A list of the `matplotlib.collections.PolyCollection` instances containing the filled area of each violin.

- **cmeans**: A `matplotlib.collections.LineCollection` instance created to identify the mean values of each of the violin’s distribution.

- **cmins**: A `matplotlib.collections.LineCollection` instance created to identify the bottom of each violin’s distribution.

- **cmaxes**: A `matplotlib.collections.LineCollection` instance created to identify the top of each violin’s distribution.
• cbars: A `matplotlib.collections.LineCollection` instance created to identify the centers of each violin’s distribution.

• cmedians: A `matplotlib.collections.LineCollection` instance created to identify the median values of each of the violin’s distribution.

Note: In addition to the above described arguments, this function can take a `data` keyword argument. If such a `data` argument is given, the following arguments are replaced by `data[<arg>]:`

• All arguments with the following names: ‘dataset’.

```python
matplotlib.pyplot.viridis()
```
set the default colormap to viridis and apply to current image if any. See help(colormaps) for more information

```python
matplotlib.pyplot.vlines(x, ymin, ymax, colors='k', linestyles='solid', label='', hold=None, data=None, **kwargs)
```
Plot vertical lines.

Plot vertical lines at each `x` from `ymin` to `ymax`.

**Parameters**

- **x**: scalar or 1D array_like
  - x-indexes where to plot the lines.
- **ymin, ymax**: scalar or 1D array_like
  - Respective beginning and end of each line. If scalars are provided, all lines will have same length.
- **colors**: array_like of colors, optional, default: ‘k’
- **linestyles**: ['solid' | 'dashed' | 'dashdot' | 'dotted'], optional
- **label**: string, optional, default: ‘’

**Returns**

- **lines**: `LineCollection`

**Other Parameters**

- **kwargs**: `LineCollection` properties.

See also:

- `hlines` horizontal lines
Examples

![Vertical lines demo](image1.png) ![Horizontal lines demo](image2.png)

**Note:** In addition to the above described arguments, this function can take a `data` keyword argument. If such a `data` argument is given, the following arguments are replaced by `data[<arg>]`:

- All arguments with the following names: ‘colors’, ‘x’, ‘ymax’, ‘ymin’.

```python
matplotlib.pyplot.waitforbuttonpress(*args, **kwargs)
```

Blocking call to interact with the figure.

This will return True if a key was pressed, False if a mouse button was pressed and None if `timeout` was reached without either being pressed.

If `timeout` is negative, does not timeout.

```python
matplotlib.pyplot.winter()
```

Set the default colormap to winter and apply to current image if any. See `help(colormaps)` for more information.

```python
matplotlib.pyplot.xcorr(x, y, normed=True, detrend=<function detrend_none>, usevlines=True, maxlags=10, hold=None, data=None, **kwargs)
```

Plot the cross correlation between `x` and `y`.

The correlation with lag `k` is defined as \( \sum_n x[n+k] \ast \text{conj}(y[n]) \).

**Parameters**

- `x` : sequence of scalars of length `n`
- `y` : sequence of scalars of length `n`
- `hold` : boolean, optional, deprecated, default: True
- `detrend` : callable, optional, default: `mlab.detrend_none`
  
  `x` is detrended by the `detrend` callable. Default is no normalization.
normed : boolean, optional, default: True
    if True, input vectors are normalised to unit length.

usevlines : boolean, optional, default: True
    if True, Axes.vlines is used to plot the vertical lines from the origin to the
    acorr. Otherwise, Axes.plot is used.

maxlags : integer, optional, default: 10
    number of lags to show. If None, will return all 2 * len(x) - 1 lags.

Returns (lags, c, line, b) : where:
    • lags are a length 2*maxlags+1 lag vector.
    • c is the 2*maxlags+1 auto correlation vector
    • line is a Line2D instance returned by plot.
    • b is the x-axis (none, if plot is used).

Other Parameters linestyle : Line2D prop, optional, default: None
    Only used if usevlines is False.

marker : string, optional, default: ‘o’

Notes

The cross correlation is performed with numpy.correlate() with mode = 2.

Note: In addition to the above described arguments, this function can take a data keyword argument.
If such a data argument is given, the following arguments are replaced by data[arg]:
    • All arguments with the following names: ‘x’, ‘y’.

matplotlib.pyplot.xkcd(scale=1, length=100, randomness=2)
    Turns on xkcd sketch-style drawing mode. This will only have effect on things drawn after this
    function is called.

For best results, the “Humor Sans” font should be installed: it is not included with matplotlib.

Parameters scale : float, optional
    The amplitude of the wiggle perpendicular to the source line.

length : float, optional
    The length of the wiggle along the line.

randomness : float, optional
    The scale factor by which the length is shrunken or expanded.
Notes

This function works by a number of rcParams, so it will probably override others you have set before. If you want the effects of this function to be temporary, it can be used as a context manager, for example:

```python
with plt.xkcd():
    # This figure will be in XKCD-style
    fig1 = plt.figure()
    # ...

    # This figure will be in regular style
    fig2 = plt.figure()
```

**matplotlib.pyplot.xlabel**(s, *args, **kwargs)

Set the x axis label of the current axis.

Default override is:

```python
override = {
    'fontsize' : 'small',
    'verticalalignment' : 'top',
    'horizontalalignment' : 'center'
}
```

See also:

**text()** For information on how override and the optional args work

**matplotlib.pyplot.xlim**(args, **kwargs)

Get or set the x limits of the current axes.

```python
xmin, xmax = xlim()  # return the current xlim
xlim((xmin, xmax))   # set the xlim to xmin, xmax
xlim( xmin, xmax )   # set the xlim to xmin, xmax
```

If you do not specify args, you can pass the xmin and xmax as kwargs, e.g.:

```python
xlim(xmax=3)  # adjust the max leaving min unchanged
xlim(xmin=1)  # adjust the min leaving max unchanged
```

Setting limits turns autoscaling off for the x-axis.

The new axis limits are returned as a length 2 tuple.

**matplotlib.pyplot.xscale**(args, **kwargs)

Set the scaling of the x-axis.

call signature:

```python
xscale(scale, **kwargs)
```
The available scales are: ‘linear’ | ‘log’ | ‘logit’ | ‘symlog’

Different keywords may be accepted, depending on the scale:

‘linear’

‘log’

*baseX/baseY*: The base of the logarithm

*nonposX/nonposY*: 
*['mask' | ‘clip’]* 
on-negative values in x or y can be masked as invalid, or clipped to a very small positive number

*subSX/subSY*: Where to place the subticks between each major tick. Should be a sequence of integers. For example, in a log10 scale: [2, 3, 4, 5, 6, 7, 8, 9]

will place 8 logarithmically spaced minor ticks between each major tick.

‘logit’

*nonpos*: 
*['mask' | ‘clip’]* 
values beyond [0, 1] can be masked as invalid, or clipped to a number very close to 0 or 1

‘symlog’

*baseX/baseY*: The base of the logarithm

*linthreshX/linthreshY*: The range (-x, x) within which the plot is linear (to avoid having the plot go to infinity around zero).

*subSX/subSY*: Where to place the subticks between each major tick. Should be a sequence of integers. For example, in a log10 scale: [2, 3, 4, 5, 6, 7, 8, 9]

will place 8 logarithmically spaced minor ticks between each major tick.

*linScaleX/linScaleY*: This allows the linear range (-linthresh to linthresh) to be stretched relative to the logarithmic range. Its value is the number of decades to use for each half of the linear range. For example, when *linScale* == 1.0 (the default), the space used for the positive and negative halves of the linear range will be equal to one decade in the logarithmic range.

**matplotlib.pyplot.xticks(*args, **kwargs)**

Get or set the x-limits of the current tick locations and labels.

```
# return locs, labels where locs is an array of tick locations and
# labels is an array of tick labels.
locs, labels = xticks()

# set the locations of the xticks
xticks( arange(6) )

# set the locations and labels of the xticks
```

The keyword args, if any, are *Text* properties. For example, to rotate long labels:
```
xticks( arange(12), calendar.month_name[1:13], rotation=17 )
```

```matlab
matplotlib.pyplot.ylabel(s, *args, **kwargs)
```

Set the y axis label of the current axis.

Defaults override is:

```
override = {
    'fontsize' : 'small',
    'verticalalignment' : 'center',
    'horizontalalignment' : 'right',
    'rotation' : 'vertical'
}
```

See also:

`text()` For information on how override and the optional args work.

```matlab
matplotlib.pyplot ylim(*args, **kwargs)
```

Get or set the y-limits of the current axes.

```
ym, ymax = ylim()  # return the current ylim
ylim( (ym, ymax) ) # set the ylim to ymin, ymax
ylim( ym, ymax )  # set the ylim to ymin, ymax
```

If you do not specify args, you can pass the `ymin` and `ymax` as kwargs, e.g.:

```
ylim(ymax=3) # adjust the max leaving min unchanged
ylim(ymin=1) # adjust the min leaving max unchanged
```

Setting limits turns autoscaling off for the y-axis.

The new axis limits are returned as a length 2 tuple.

```matlab
matplotlib.pyplot.yscale(*args, **kwargs)
```

Set the scaling of the y-axis.

```
yscale(scale, **kwargs)
```

The available scales are: 'linear' | 'log' | 'logit' | 'symlog'

Different keywords may be accepted, depending on the scale:

- `linear`
- `log`
- `basex/basey`: The base of the logarithm
- `nonposx/nonposy`: ['mask' | 'clip'] non-positive values in x or y can be masked as invalid, or clipped to a very small positive number
subsx/subsy: Where to place the subticks between each major tick. Should be a sequence of integers. For example, in a log10 scale: [2, 3, 4, 5, 6, 7, 8, 9]

will place 8 logarithmically spaced minor ticks between each major tick.

‘logit’

nonpos: [‘mask’ | ‘clip’] values beyond ]0, 1[ can be masked as invalid, or clipped to a number very close to 0 or 1

’symlog’

basex/basey: The base of the logarithm

linthreshx/linthreshy: The range (-x, x) within which the plot is linear (to avoid having the plot go to infinity around zero).

subsx/subsy: Where to place the subticks between each major tick. Should be a sequence of integers. For example, in a log10 scale: [2, 3, 4, 5, 6, 7, 8, 9]

will place 8 logarithmically spaced minor ticks between each major tick.

linscalex/linscaley: This allows the linear range (-linthresh to linthresh) to be stretched relative to the logarithmic range. Its value is the number of decades to use for each half of the linear range. For example, when linscale == 1.0 (the default), the space used for the positive and negative halves of the linear range will be equal to one decade in the logarithmic range.

matplotlib.pyplot.yticks(*args, **kwargs)
Get or set the y-limits of the current tick locations and labels.

# return locs, labels where locs is an array of tick locations and # labels is an array of tick labels.
locs, labels = yticks()

# set the locations of the yticks
yticks( arange(6) )

# set the locations and labels of the yticks

The keyword args, if any, are Text properties. For example, to rotate long labels:

yticks( arange(12), calendar.month_name[1:13], rotation=45 )
46.1 matplotlib.rcsetup

The rcsetup module contains the default values and the validation code for customization using matplotlib’s rc settings.

Each rc setting is assigned a default value and a function used to validate any attempted changes to that setting. The default values and validation functions are defined in the rcsetup module, and are used to construct the rcParams global object which stores the settings and is referenced throughout matplotlib.

These default values should be consistent with the default matplotlibrc file that actually reflects the values given here. Any additions or deletions to the parameter set listed here should also be visited to the matplotlibrc.template in matplotlib’s root source directory.

```python
class matplotlib.rcsetup.ValidateInStrings(key, valid, ignorecase=False)
    Bases: object
    valid is a list of legal strings

class matplotlib.rcsetup.ValidateInterval(vmin, vmax, closedmin=True, closedmax=True)
    Bases: object
    Value must be in interval

matplotlib.rcsetup.cycler(*args, **kwargs)
    Creates a cycler.Cycler object much like cycler.cycler(), but includes input validation.
    cycler(arg) cycler(label, itr) cycler(label1=itr1[, label2=itr2[, ...]])
    Form 1 simply copies a given Cycler object.
    Form 2 creates a Cycler from a label and an iterable.
    Form 3 composes a Cycler as an inner product of the pairs of keyword arguments. In other words, all of the iterables are cycled simultaneously, as if through zip().

    Parameters arg : Cycler
        Copy constructor for Cycler.

    label : name
```
The property key. Must be a valid Artist property. For example, ‘color’ or ‘linestyle’. Aliases are allowed, such as ‘c’ for ‘color’ and ‘lw’ for ‘linewidth’.

itr : iterable

Finite-length iterable of the property values. These values are validated and will raise a ValueError if invalid.

Returns cycler : Cycler

New cycler.Cycler for the given properties

matplotlib.rcParams.deprecate_axes_colorcycle(value)

matplotlib.rcParams.deprecate_axes_hold(value)

matplotlib.rcParams.deprecate_savefig_extension(value)

matplotlib.rcParams.deprecate_svg_embed_char_paths(value)

matplotlib.rcParams.deprecate_svg_image_noscale(value)

matplotlib.rcParams.update_savefig_format(value)

matplotlib.rcParams.validate_animation_writer_path(path)

matplotlib.rcParams.validate_any(s)

matplotlib.rcParams.validate_anylist(s)

matplotlib.rcParams.validate_aspect(s)

matplotlib.rcParams.validate_axisbelow(s)

matplotlib.rcParams.validate_backend(s)

matplotlib.rcParams.validate_bbox(s)

matplotlib.rcParams.validate_bool(b)

Convert b to a boolean or raise

matplotlib.rcParams.validate_bool_maybe_none(b)

Convert b to a boolean or raise
matplotlib.rcsetup.validate_capstylelist(s)

matplotlib.rcsetup.validate_color(s)
    return a valid color arg
matplotlib.rcsetup.validate_color_for_prop_cycle(s)

matplotlib.rcsetup.validate_color_or_auto(s)

matplotlib.rcsetup.validate_color_or_inherit(s)
    return a valid color arg
matplotlib.rcsetup.validate_colorlist(s)
    return a list of colorspecs
matplotlib.rcsetup.validate_corner_mask(s)

matplotlib.rcsetup.validate_cycler(s)
    return a Cycler object from a string repr or the object itself
matplotlib.rcsetup.validate_dashlist(s)

matplotlib.rcsetup.validate_dpi(s)
    confirm s is string ‘figure’ or convert s to float or raise
matplotlib.rcsetup.validate_fillstylelist(s)

matplotlib.rcsetup.validate_float(s)
    convert s to float or raise
matplotlib.rcsetup.validate_float_or_None(s)
    convert s to float, None or raise
matplotlib.rcsetup.validate_floatlist(s)
    convert s to float or raise
matplotlib.rcsetup.validate_font_properties(s)

matplotlib.rcsetup.validate_fontsize(s)

matplotlib.rcsetup.validate_fontszelist(s)

matplotlib.rcsetup.validate_fonttype(s)
    confirm that this is a Postscript of PDF font type that we know how to convert to
matplotlib.rcsetup.validate_hatch(s)
    Validate a hatch pattern. A hatch pattern string can have any sequence of the following characters: \\
    / | - + . x o 0.
matplotlib.rcsetup.validate_hatchlist(s)
    Validate a hatch pattern. A hatch pattern string can have any sequence of the following characters: \\
    / | - + * . x o 0.
matplotlib.rcsetup.validate_hinting(s)

matplotlib.rcsetup.validate_hist_bins(s)

matplotlib.rcsetup.validate_int(s)
    convert s to int or raise
matplotlib.rcsetup.validate_int_or_None(s)
    if not None, tries to validate as an int
matplotlib.rcsetup.validate_joinstylelist(s)

matplotlib.rcsetup.validate_maskedarray(v)

matplotlib.rcsetup.validate_negative_linestyle_legacy(s)

class matplotlib.rcsetup.validate_nseq_float(n=None, allow_none=False)
    Bases: object

class matplotlib.rcsetup.validate_nseq_int(n=None)
    Bases: object

matplotlib.rcsetup.validate_path_exists(s)
    If s is a path, return s, else False
matplotlib.rcsetup.validate_ps_distiller(s)

matplotlib.rcsetup.validate_sketch(s)

matplotlib.rcsetup.validate_string_or_None(s)
    convert s to string or raise
matplotlib.rcsetup.validate_stringlist(s)
    return a list
matplotlib.rcsetup.validate_tkpythoninspect(s)

matplotlib.rcsetup.validate_toolbar(s)

matplotlib.rcsetup.validate_whiskers(s)
47.1 matplotlib.sankey

Module for creating Sankey diagrams using matplotlib

class matplotlib.sankey.Sankey(ax=None, scale=1.0, unit='', format='%.2f', gap=0.25, radius=0.1, shoulder=0.03, offset=0.15, head_angle=100, margin=0.4, tolerance=1e-06, **kwargs)

Bases: object

Sankey diagram in matplotlib

Sankey diagrams are a specific type of flow diagram, in which the width of the arrows is shown proportionally to the flow quantity. They are typically used to visualize energy or material or cost transfers between processes. Wikipedia (6/1/2011)

Create a new Sankey instance.

Optional keyword arguments:
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ax</td>
<td>axes onto which the data should be plotted If ax isn’t provided, new axes will be created.</td>
</tr>
<tr>
<td>scale</td>
<td>scaling factor for the flows scale sizes the width of the paths in order to maintain proper layout. The same scale is applied to all subdiagrams. The value should be chosen such that the product of the scale and the sum of the inputs is approximately 1.0 (and the product of the scale and the sum of the outputs is approximately -1.0).</td>
</tr>
<tr>
<td>unit</td>
<td>string representing the physical unit associated with the flow quantities If unit is None, then none of the quantities are labeled.</td>
</tr>
<tr>
<td>format</td>
<td>a Python number formatting string to be used in labeling the flow as a quantity (i.e., a number times a unit, where the unit is given)</td>
</tr>
<tr>
<td>gap</td>
<td>space between paths that break in/break away to/from the top or bottom</td>
</tr>
<tr>
<td>radius</td>
<td>inner radius of the vertical paths</td>
</tr>
<tr>
<td>shoulder</td>
<td>size of the shoulders of output arrowS</td>
</tr>
<tr>
<td>offset</td>
<td>text offset (from the dip or tip of the arrow)</td>
</tr>
<tr>
<td>head_angle</td>
<td>angle of the arrow heads (and negative of the angle of the tails) [deg]</td>
</tr>
<tr>
<td>margin</td>
<td>minimum space between Sankey outlines and the edge of the plot area</td>
</tr>
<tr>
<td>tolerance</td>
<td>acceptable maximum of the magnitude of the sum of flows The magnitude of the sum of connected flows cannot be greater than tolerance.</td>
</tr>
</tbody>
</table>

The optional arguments listed above are applied to all subdiagrams so that there is consistent alignment and formatting.

If `Sankey` is instantiated with any keyword arguments other than those explicitly listed above (**kwargs), they will be passed to `add()`, which will create the first subdiagram.

In order to draw a complex Sankey diagram, create an instance of `Sankey` by calling it without any kwargs:

```
sankey = Sankey()
```

Then add simple Sankey sub-diagrams:

```
sankey.add() # 1
sankey.add() # 2
#...
sankey.add() # n
```

Finally, create the full diagram:

```
sankey.finish()
```

Or, instead, simply daisy-chain those calls:

```
Sankey().add().add... .add().finish()
```

See also:
add() finish()

Examples:

The default settings produce a diagram like this.
Flow Diagram of a Widget

Widget

0% First
10% Second
20% Third
5%

25% Fifth
10%

40% Hurray!

60%
```python
add(patchlabel='', flows=None, orientations=None, labels='', trunklength=1.0, pathlengths=0.25, prior=None, connect=(0, 0), rotation=0, **kwargs)
```

Add a simple Sankey diagram with flows at the same hierarchical level.

Return value is the instance of `Sankey`.

Optional keyword arguments:
<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>patchlabel</td>
<td>label to be placed at the center of the diagram Note: label (not patchlabel) will be passed to the patch through **kwargs and can be used to create an entry in the legend.</td>
</tr>
<tr>
<td>flows</td>
<td>array of flow values By convention, inputs are positive and outputs are negative.</td>
</tr>
<tr>
<td>orientations</td>
<td>list of orientations of the paths Valid values are 1 (from/to the top), 0 (from/to the left or right), or -1 (from/to the bottom). If orientations == 0, inputs will break in from the left and outputs will break away to the right.</td>
</tr>
<tr>
<td>labels</td>
<td>list of specifications of the labels for the flows Each value may be None (no labels), ‘’ (just label the quantities), or a labeling string. If a single value is provided, it will be applied to all flows. If an entry is a non-empty string, then the quantity for the corresponding flow will be shown below the string. However, if the unit of the main diagram is None, then quantities are never shown, regardless of the value of this argument.</td>
</tr>
<tr>
<td>trunklength</td>
<td>length between the bases of the input and output groups</td>
</tr>
<tr>
<td>pathlengths</td>
<td>list of lengths of the arrows before break-in or after break-away If a single value is given, then it will be applied to the first (inside) paths on the top and bottom, and the length of all other arrows will be justified accordingly. The pathlengths are not applied to the horizontal inputs and outputs.</td>
</tr>
<tr>
<td>prior</td>
<td>index of the prior diagram to which this diagram should be connected</td>
</tr>
<tr>
<td>connect</td>
<td>a (prior, this) tuple indexing the flow of the prior diagram and the flow of this diagram which should be connected If this is the first diagram or prior is None, connect will be ignored.</td>
</tr>
<tr>
<td>rotation</td>
<td>angle of rotation of the diagram [deg] rotation is ignored if this diagram is connected to an existing one (using prior and connect). The interpretation of the orientations argument will be rotated accordingly (e.g., if rotation == 90, an orientations entry of 1 means to/from the left).</td>
</tr>
</tbody>
</table>

Valid kwargs are `matplotlib.patches.PathPatch()` arguments:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or ec</td>
<td>mpl color spec, None, ‘none’, or ‘auto’</td>
</tr>
</tbody>
</table>
### Table 47.1 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>facecolor</strong></td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td><strong>figure</strong></td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td><strong>fill</strong></td>
<td>[True</td>
</tr>
<tr>
<td><strong>gid</strong></td>
<td>an id string</td>
</tr>
<tr>
<td><strong>hatch</strong></td>
<td>[‘/’</td>
</tr>
<tr>
<td><strong>joinstyle</strong></td>
<td>[‘miter’</td>
</tr>
<tr>
<td><strong>label</strong></td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td><strong>linestyle</strong></td>
<td>[‘solid’</td>
</tr>
<tr>
<td><strong>linewidth</strong></td>
<td>float or None for default</td>
</tr>
<tr>
<td><strong>path_effects</strong></td>
<td>unknown</td>
</tr>
<tr>
<td><strong>picker</strong></td>
<td>[None</td>
</tr>
<tr>
<td><strong>rasterized</strong></td>
<td>[True</td>
</tr>
<tr>
<td><strong>sketch_params</strong></td>
<td>unknown</td>
</tr>
<tr>
<td><strong>snap</strong></td>
<td>unknown</td>
</tr>
<tr>
<td><strong>transform</strong></td>
<td><code>Transform</code> instance</td>
</tr>
<tr>
<td><strong>url</strong></td>
<td>a url string</td>
</tr>
<tr>
<td><strong>visible</strong></td>
<td>[True</td>
</tr>
<tr>
<td><strong>zorder</strong></td>
<td>any number</td>
</tr>
</tbody>
</table>

As examples, **fill=False** and **label='A legend entry'**. By default, **facecolor='#bfd1d4'** (light blue) and **linewidth=0.5**.

The indexing parameters (**prior** and **connect**) are zero-based.

The flows are placed along the top of the diagram from the inside out in order of their index within the **flows** list or array. They are placed along the sides of the diagram from the top down and along the bottom from the outside in.

If the sum of the inputs and outputs is nonzero, the discrepancy will appear as a cubic Bezier curve along the top and bottom edges of the trunk.

**See also:**

`finish()`

**finish()**

Adjust the axes and return a list of information about the Sankey subdiagram(s).

Return value is a list of subdiagrams represented with the following fields:
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>patch</td>
<td>Sankey outline (an instance of PathPatch)</td>
</tr>
<tr>
<td>flows</td>
<td>values of the flows (positive for input, negative for output)</td>
</tr>
<tr>
<td>angles</td>
<td>list of angles of the arrows [deg/90] For example, if the diagram has not been rotated, an input to the top side will have an angle of 3 (DOWN), and an output from the top side will have an angle of 1 (UP). If a flow has been skipped (because its magnitude is less than tolerance), then its angle will be None.</td>
</tr>
<tr>
<td>tips</td>
<td>array in which each row is an [x, y] pair indicating the positions of the tips (or “dips”) of the flow paths If the magnitude of a flow is less the tolerance for the instance of Sankey, the flow is skipped and its tip will be at the center of the diagram.</td>
</tr>
<tr>
<td>text</td>
<td>Text instance for the label of the diagram</td>
</tr>
<tr>
<td>texts</td>
<td>list of Text instances for the labels of flows</td>
</tr>
</tbody>
</table>

See also:

add()
48.1 matplotlib.scale

```python
class matplotlib.scale.InvertedLog10Transform(shorthand_name=None):
    Bases: matplotlib.scale.InvertedLogTransformBase
    Creates a new TransformNode.
    shorthand_name - a string representing the "name" of this transform. The name carries no significance other than to improve the readability of str(transform) when DEBUG=True.

    base = 10.0

    inverted()

class matplotlib.scale.InvertedLog2Transform(shorthand_name=None):
    Bases: matplotlib.scale.InvertedLogTransformBase
    Creates a new TransformNode.
    shorthand_name - a string representing the "name" of this transform. The name carries no significance other than to improve the readability of str(transform) when DEBUG=True.

    base = 2.0

    inverted()

class matplotlib.scale.InvertedLogTransform(base):
    Bases: matplotlib.scale.InvertedLogTransformBase

    inverted()

class matplotlib.scale.InvertedLogTransformBase(shorthand_name=None):
    Bases: matplotlib.transforms.Transform
    Creates a new TransformNode.
```

SCALE
shorthand_name - a string representing the “name” of this transform. The name carries no significance other than to improve the readability of \( \text{str(transform)} \) when DEBUG=True.

\[
\text{has_inverse} = \text{True}
\]

\[
\text{input_dims} = 1
\]

\[
\text{is_separable} = \text{True}
\]

\[
\text{output_dims} = 1
\]

\[
\text{transform_non_affine}(a)
\]

class matplotlib.scale.InvertedNaturalLogTransform(shorthand_name=None)
    Bases: matplotlib.scale.InvertedLogTransformBase
    Creates a new TransformNode.

shorthand_name - a string representing the “name” of this transform. The name carries no significance other than to improve the readability of \( \text{str(transform)} \) when DEBUG=True.

\[
\text{base} = 2.718281828459045
\]

inverted()

class matplotlib.scale.InvertedSymmetricalLogTransform(base, linthresh, linscale)
    Bases: matplotlib.transforms.Transform
    has_inverse = True

\[
\text{input_dims} = 1
\]

inverted()

\[
\text{is_separable} = \text{True}
\]

\[
\text{output_dims} = 1
\]

\[
\text{transform_non_affine}(a)
\]

class matplotlib.scale.LinearScale(axis, **kwargs)
    Bases: matplotlib.scale.ScaleBase
    The default linear scale.
get_transform()
The transform for linear scaling is just the IdentityTransform.

name = ‘linear’

set_default_locators_and_formatters(axis)
Set the locators and formatters to reasonable defaults for linear scaling.

class matplotlib.scale.Log10Transform(nonpos)
Bases: matplotlib.scale.LogTransformBase

base = 10.0

inverted()

class matplotlib.scale.Log2Transform(nonpos)
Bases: matplotlib.scale.LogTransformBase

base = 2.0

inverted()

class matplotlib.scale.LogScale(axis, **kwargs)
Bases: matplotlib.scale.ScaleBase

A standard logarithmic scale. Care is taken so non-positive values are not plotted.
For computational efficiency (to push as much as possible to Numpy C code in the common cases),
this scale provides different transforms depending on the base of the logarithm:

• base 10 (Log10Transform)
• base 2 (Log2Transform)
• base e (NaturalLogTransform)
• arbitrary base (LogTransform)

basex/basey: The base of the logarithm
nonposx/nonposy: [‘mask’ | ‘clip’] non-positive values in x or y can be masked as invalid, or clipped
to a very small positive number
subsx/subsy: Where to place the subticks between each major tick. Should be a sequence of integers.
For example, in a log10 scale: [2, 3, 4, 5, 6, 7, 8, 9]
will place 8 logarithmically spaced minor ticks between each major tick.

class InvertedLog10Transform(shorthand_name=None)
Bases: matplotlib.scale.InvertedLogTransformBase

Creates a new TransformNode.
shorthand_name - a string representing the “name” of this transform. The name carries no significance other than to improve the readability of str(transform) when DEBUG=True.

base = 10.0

inverted()

class InvertedLog2Transform(shorthand_name=None)
Bases: matplotlib.scale.InvertedLogTransformBase

Creates a new TransformNode.

shorthand_name - a string representing the “name” of this transform. The name carries no significance other than to improve the readability of str(transform) when DEBUG=True.

base = 2.0

inverted()

class InvertedLogTransform(base)
Bases: matplotlib.scale.InvertedLogTransformBase

inverted()

class InvertedNaturalLogTransform(shorthand_name=None)
Bases: matplotlib.scale.InvertedLogTransformBase

Creates a new TransformNode.

shorthand_name - a string representing the “name” of this transform. The name carries no significance other than to improve the readability of str(transform) when DEBUG=True.

base = 2.718281828459045

inverted()

class Log10Transform(nonpos)
Bases: matplotlib.scale.LogTransformBase

base = 10.0

inverted()

class Log2Transform(nonpos)
Bases: matplotlib.scale.LogTransformBase
base = 2.0

inverted()

class LogTransform(base, nonpos)
    Bases: matplotlib.scale.LogTransformBase
    inverted()

class LogTransformBase(nonpos)
    Bases: matplotlib.transforms.Transform
    has_inverse = True

    input_dims = 1

    is_separable = True

    output_dims = 1

    transform_non_affine(a)

class NaturalLogTransform(nonpos)
    Bases: matplotlib.scale.LogTransformBase

    base = 2.718281828459045

    inverted()

    get_transform()
        Return a Transform instance appropriate for the given logarithm base.

    limit_range_for_scale(vmin, vmax, minpos)
        Limit the domain to positive values.

    name = ‘log’

    set_default_locators_and_formatters(axis)
        Set the locators and formatters to specialized versions for log scaling.

class matplotlib.scale.LogTransform(base, nonpos)
    Bases: matplotlib.scale.LogTransformBase
    inverted()
class matplotlib.scale.LogTransformBase(nonpos)
    Bases: matplotlib.transforms.Transform
    
    has_inverse = True
    
    input_dims = 1
    
    is_separable = True
    
    output_dims = 1
    
    transform_non_affine(a)

class matplotlib.scale.LogisticTransform(nonpos='mask')
    Bases: matplotlib.transforms.Transform
    
    has_inverse = True
    
    input_dims = 1
    
    inverted()
    
    is_separable = True
    
    output_dims = 1
    
    transform_non_affine(a)
        logistic transform (base 10)

class matplotlib.scale.LogitScale(axis, nonpos='mask')
    Bases: matplotlib.scale.ScaleBase
    
    Logit scale for data between zero and one, both excluded.
    
    This scale is similar to a log scale close to zero and to one, and almost linear around 0.5. It maps the interval \(0, 1\) onto \(-\infty, +\infty\).
    
    nonpos: ['mask' | 'clip'] values beyond \(0, 1\) can be masked as invalid, or clipped to a number very close to 0 or 1
    
    get_transform()
        Return a LogitTransform instance.
    
    limit_range_for_scale(vmin, vmax, minpos)
        Limit the domain to values between 0 and 1 (excluded).
name = 'logit'

set_default_locators_and_formatters(axis)

class matplotlib.scale.LogitTransform(nonpos)
    Bases: matplotlib.transforms.Transform
    has_inverse = True

    input_dims = 1

    inverted()

    is_separable = True

    output_dims = 1

    transform_non_affine(a)
        logit transform (base 10), masked or clipped

class matplotlib.scale.NaturalLogTransform(nonpos)
    Bases: matplotlib.scale.LogTransformBase

    base = 2.718281828459045

    inverted()

class matplotlib.scale.ScaleBase
    Bases: object

    The base class for all scales.

    Scales are separable transformations, working on a single dimension.

    Any subclasses will want to override:
        • name
        • get_transform()
        • set_default_locators_and_formatters()

    And optionally:
        • limit_range_for_scale()

    get_transform()
        Return the Transform object associated with this scale.
**Matplotlib, Release 2.0.2**

`limit_range_for_scale(vmin, vmax, minpos)`

Returns the range `vmin, vmax`, possibly limited to the domain supported by this scale.

*minpos should be the minimum positive value in the data.* This is used by log scales to determine a minimum value.

`set_default_locators_and_formatters(axis)`

Set the `Locator` and `Formatter` objects on the given axis to match this scale.

```text
class matplotlib.scale.SymmetricalLogScale(axis, **kwargs)
Bases: matplotlib.scale.ScaleBase

The symmetrical logarithmic scale is logarithmic in both the positive and negative directions from the origin.

Since the values close to zero tend toward infinity, there is a need to have a range around zero that is linear. The parameter `linthresh` allows the user to specify the size of this range (-linthresh, linthresh).

*basex/basey:* The base of the logarithm

*linthreshx/linthreshy:* The range (-x, x) within which the plot is linear (to avoid having the plot go to infinity around zero).

*subsx/subsy:* Where to place the subticks between each major tick. Should be a sequence of integers.

For example, in a log10 scale: [2, 3, 4, 5, 6, 7, 8, 9]

will place 8 logarithmically spaced minor ticks between each major tick.

*linscalex/linscaley:* This allows the linear range (-linthresh to linthresh) to be stretched relative to the logarithmic range. Its value is the number of decades to use for each half of the linear range. For example, when `linscale == 1.0` (the default), the space used for the positive and negative halves of the linear range will be equal to one decade in the logarithmic range.

```text
class InvertedSymmetricalLogTransform(base, linthresh, linscale)
Bases: matplotlib.transforms.Transform

has_inverse = True

input_dims = 1

inverted()

is_separable = True

output_dims = 1

transform_non_affine(a)

class SymmetricalLogTransform(base, linthresh, linscale)
Bases: matplotlib.transforms.Transform
```
has_inverse = True

input_dims = 1

inverted()

is_separable = True

output_dims = 1

transform_non_affine(a)

get_transform()
    Return a SymmetricalLogTransform instance.

ame = 'symlog'

set_default_locators_and_formatters(axis)
    Set the locators and formatters to specialized versions for symmetrical log scaling.

class matplotlib.scale.SymmetricalLogTransform(base, linthresh, linscale)
    Bases: matplotlib.transforms.Transform

    has_inverse = True

    input_dims = 1

    inverted()

    is_separable = True

    output_dims = 1

    transform_non_affine(a)

matplotlib.scale.get_scale_docs()
    Helper function for generating docstrings related to scales.

matplotlib.scale.get_scale_names()

matplotlib.scale.register_scale(scale_class)
    Register a new kind of scale.
scale_class must be a subclass of ScaleBase.

```
matplotlib.scale.scale_factory(scale, axis, **kwargs)
```

Return a scale class by name.

ACCEPTS: [ linear | log | logit | symlog ]
49.1 matplotlib.spines

class matplotlib.spines.Spine(axes, spine_type, path, **kwargs)

Bases: matplotlib.patches.Patch

an axis spine – the line noting the data area boundaries

Spines are the lines connecting the axis tick marks and noting the boundaries of the data area. They can be placed at arbitrary positions. See function: set_position for more information.

The default position is ('outward', 0).

Spines are subclasses of class: Patch, and inherit much of their behavior.

Spines draw a line or a circle, depending if function: set_patch_line or function: set_patch_circle has been called. Line-like is the default.

- **axes**: the Axes instance containing the spine
- **spine_type**: a string specifying the spine type
- **path**: the path instance used to draw the spine

Valid kwargs are:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform), Patch, None]</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor</td>
<td>mpl color spec, None, 'none', or 'auto'</td>
</tr>
</tbody>
</table>

Continued on next page.
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>facecolor</code> or <code>fc</code></td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td><code>figure</code></td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td><code>fill</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>gid</code></td>
<td>an id string</td>
</tr>
<tr>
<td><code>hatch</code></td>
<td>[‘/</td>
</tr>
<tr>
<td><code>joinstyle</code></td>
<td>[‘miter’</td>
</tr>
<tr>
<td><code>label</code></td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td><code>linestyle</code> or <code>ls</code></td>
<td>[‘solid’</td>
</tr>
<tr>
<td><code>linewidth</code> or <code>lw</code></td>
<td>float or None for default</td>
</tr>
<tr>
<td><code>path_effects</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>picker</code></td>
<td>[None</td>
</tr>
<tr>
<td><code>rasterized</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>sketch_params</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>snap</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>transform</code></td>
<td><code>Transform</code> instance</td>
</tr>
<tr>
<td><code>url</code></td>
<td>a url string</td>
</tr>
<tr>
<td><code>visible</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>zorder</code></td>
<td>any number</td>
</tr>
</tbody>
</table>

### Classmethod `circular_spine`(*axes*, *center*, *radius*, **kwargs)

(staticmethod) Returns a circular `Spine`.

```python
classmethod circular_spine(axes, center, radius, **kwargs)
```

### `cla()`
Clear the current spine

```python
cla()
```

### `draw`(*artist*, *renderer*, *args*, **kwargs)

```python
draw(artist, renderer, *args, **kwargs)
```

### `get_bounds()`
Get the bounds of the spine.

```python
class get_bounds()
```

### `get_patch_transform()`

```python
class get_patch_transform()
```

### `get_path()`

```python
class get_path()
```

### `get_position()`
get the spine position

```python
class get_position()
```

### `get_smart_bounds()`
get whether the spine has smart bounds

```python
class get_smart_bounds()
```

### `get_spine_transform()`
get the spine transform

```python
class get_spine_transform()
```

### `is_frame_like()`
return True if directly on axes frame

```python
class is_frame_like()
```
This is useful for determining if a spine is the edge of an old style MPL plot. If so, this function will return True.

**classmethod linear_spine**(axes, spine_type, **kwargs)

(staticmethod) Returns a linear Spine.

**register_axis**(axis)

register an axis

An axis should be registered with its corresponding spine from the Axes instance. This allows the spine to clear any axis properties when needed.

**set_bounds**(low, high)

Set the bounds of the spine.

**set_color**(c)

Set the edgecolor.

ACCEPTS: matplotlib color arg or sequence of rgba tuples

See also:

**set_facecolor()**, **set_edgecolor()** For setting the edge or face color individually.

**set_patch_circle**(center, radius)

set the spine to be circular

**set_patch_line**()

set the spine to be linear

**set_position**(position)

set the position of the spine

Spine position is specified by a 2 tuple of (position type, amount). The position types are:

- ‘outward’ : place the spine out from the data area by the specified number of points. (Negative values specify placing the spine inward.)
- ‘axes’ : place the spine at the specified Axes coordinate (from 0.0-1.0).
- ‘data’ : place the spine at the specified data coordinate.

Additionally, shorthand notations define a special positions:

- ‘center’ -> (‘axes’, 0.5)
- ‘zero’ -> (‘data’, 0.0)

**set_smart_bounds**(value)

set the spine and associated axis to have smart bounds
50.1 matplotlib.style

matplotlib.style.context(style, after_reset=False)
Context manager for using style settings temporarily.

**Parameters**

- **style**: str, dict, or list
  A style specification. Valid options are:

<table>
<thead>
<tr>
<th>type</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>str</td>
<td>The name of a style or a path/URL to a style file. For a list of available style names, see style.available.</td>
</tr>
<tr>
<td>dict</td>
<td>Dictionary with valid key/value pairs for matplotlib.rcParams.</td>
</tr>
<tr>
<td>list</td>
<td>A list of style specifiers (str or dict) applied from first to last in the list.</td>
</tr>
</tbody>
</table>

- **after_reset**: bool
  If True, apply style after resetting settings to their defaults; otherwise, apply style on top of the current settings.

matplotlib.style.reload_library()
Reload style library.

matplotlib.style.use(style)
Use matplotlib style settings from a style specification.

The style name of ‘default’ is reserved for reverting back to the default style settings.

**Parameters**

- **style**: str, dict, or list
  A style specification. Valid options are:

<table>
<thead>
<tr>
<th>type</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>str</td>
<td>The name of a style or a path/URL to a style file. For a list of available style names, see style.available.</td>
</tr>
<tr>
<td>dict</td>
<td>Dictionary with valid key/value pairs for matplotlib.rcParams.</td>
</tr>
<tr>
<td>list</td>
<td>A list of style specifiers (str or dict) applied from first to last in the list.</td>
</tr>
</tbody>
</table>

matplotlib.style.library
Dictionary of available styles
matplotlib.style.available
List of available styles
51.1 `matplotlib.text`

Classes for including text in a figure.

```python
class matplotlib.text.Annotation(s, xy, xytext=None, xycoords='data', textcoords=None, arrowprops=None, annotation_clip=None, **kwargs)
```

Annotate the point `xy` with text `s`.

Additional kwargs are passed to `Text`.

**Parameters**

- `s` : str
  
The text of the annotation

- `xy` : iterable
  
  Length 2 sequence specifying the `(x,y)` point to annotate

- `xytext` : iterable, optional
  
  Length 2 sequence specifying the `(x,y)` to place the text at. If None, defaults to `xy`.

- `xycoords` : str, Artist, Transform, callable or tuple, optional
  
  The coordinate system that `xy` is given in.
  
  For a `str` the allowed values are:
### Properties and Their Descriptions

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘figure points’</td>
<td>points from the lower left of the figure</td>
</tr>
<tr>
<td>‘figure pixels’</td>
<td>pixels from the lower left of the figure</td>
</tr>
<tr>
<td>‘figure fraction’</td>
<td>fraction of figure from lower left</td>
</tr>
<tr>
<td>‘axes points’</td>
<td>points from lower left corner of axes</td>
</tr>
<tr>
<td>‘axes pixels’</td>
<td>pixels from lower left corner of axes</td>
</tr>
<tr>
<td>‘axes fraction’</td>
<td>fraction of axes from lower left</td>
</tr>
<tr>
<td>‘data’</td>
<td>use the coordinate system of the object being annotated</td>
</tr>
<tr>
<td></td>
<td>(default)</td>
</tr>
<tr>
<td>‘polar’</td>
<td>((\theta, r)) if not native ‘data’ coordinates</td>
</tr>
</tbody>
</table>

If a *Artist* object is passed in the units are fraction if it’s bounding box.

If a *Transform* object is passed in use that to transform \(xy\) to screen coordinates.

If a callable it must take a *RendererBase* object as input and return a *Transform* or *Bbox* object.

If a tuple must be length 2 tuple of str, *Artist*, *Transform* or callable objects. The first transform is used for the \(x\) coordinate and the second for \(y\).

See *Advanced Annotation* for more details.

Defaults to ‘data’

**textcoords**: str, *Artist*, *Transform*, callable or tuple, optional

The coordinate system that \(xytext\) is given, which may be different than the coordinate system used for \(xy\).

All \(xycoords\) values are valid as well as the following strings:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘offset points’</td>
<td>offset (in points) from the (xy) value</td>
</tr>
<tr>
<td>‘offset pixels’</td>
<td>offset (in pixels) from the (xy) value</td>
</tr>
</tbody>
</table>

defaults to the input of \(xycoords\)

**arrowprops**: dict, optional

If not None, properties used to draw a *FancyArrowPatch* arrow between \(xy\) and \(xytext\).

If *arrowprops* does not contain the key ‘arrowstyle’ the allowed keys are:
<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>width</td>
<td>the width of the arrow in points</td>
</tr>
<tr>
<td>headwidth</td>
<td>the width of the base of the arrow head in points</td>
</tr>
<tr>
<td>headlength</td>
<td>the length of the arrow head in points</td>
</tr>
<tr>
<td>shrink</td>
<td>fraction of total length to 'shrink' from both ends</td>
</tr>
<tr>
<td>?</td>
<td>any key to <code>matplotlib.patches.FancyArrowPatch</code></td>
</tr>
</tbody>
</table>

If the `arrowprops` contains the key 'arrowstyle' the above keys are forbidden. The allowed values of 'arrowstyle' are:

<table>
<thead>
<tr>
<th>Name</th>
<th>Attrs</th>
</tr>
</thead>
<tbody>
<tr>
<td>'-'</td>
<td>None</td>
</tr>
<tr>
<td>'-&gt;'</td>
<td>head_length=0.4,head_width=0.2</td>
</tr>
<tr>
<td>'-['</td>
<td>widthB=1.0,lengthB=0.2,angleB=None</td>
</tr>
<tr>
<td>'[-]'</td>
<td>widthA=1.0,widthB=1.0</td>
</tr>
<tr>
<td>'-</td>
<td>&gt;'</td>
</tr>
<tr>
<td>'&lt;-'</td>
<td>head_length=0.4,head_width=0.2</td>
</tr>
<tr>
<td>'&lt;-&gt;'</td>
<td>head_length=0.4,head_width=0.2</td>
</tr>
<tr>
<td>'&lt;</td>
<td>-&gt;'</td>
</tr>
<tr>
<td>'&lt;</td>
<td>-'</td>
</tr>
<tr>
<td>'&lt;</td>
<td>--&gt;'</td>
</tr>
<tr>
<td>'fancy'</td>
<td>head_length=0.4,head_width=0.4,tail_width=0.4</td>
</tr>
<tr>
<td>'simple'</td>
<td>head_length=0.5,head_width=0.5,tail_width=0.2</td>
</tr>
<tr>
<td>'wedge'</td>
<td>tail_width=0.3,shrink_factor=0.5</td>
</tr>
</tbody>
</table>

Valid keys for `FancyArrowPatch` are:

<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>arrowstyle</td>
<td>the arrow style</td>
</tr>
<tr>
<td>connectionstyle</td>
<td>the connection style</td>
</tr>
<tr>
<td>relpos</td>
<td>default is (0.5, 0.5)</td>
</tr>
<tr>
<td>patchA</td>
<td>default is bounding box of the text</td>
</tr>
<tr>
<td>patchB</td>
<td>default is None</td>
</tr>
<tr>
<td>shrinkA</td>
<td>default is 2 points</td>
</tr>
<tr>
<td>shrinkB</td>
<td>default is 2 points</td>
</tr>
<tr>
<td>mutation_scale</td>
<td>default is text size (in points)</td>
</tr>
<tr>
<td>mutation_aspect</td>
<td>default is 1.</td>
</tr>
<tr>
<td>?</td>
<td>any key for <code>matplotlib.patches.PathPatch</code></td>
</tr>
</tbody>
</table>

Defaults to None

**annotationClip** : bool, optional

Controls the visibility of the annotation when it goes outside the axes area.

If True, the annotation will only be drawn when the xy is inside the axes. If False, the annotation will always be drawn regardless of its position.

The default is None, which behave as True only if `xycoords` is “data”.

**Returns** Annotation

**anncoords**
contains(event)

draw(artist, renderer, *args, **kwargs)
    Draw the Annotation object to the given renderer.

get_window_extent(renderer=None)
    Return a Bbox object bounding the text and arrow annotation, in display units.

    renderer defaults to the _renderer attribute of the text object. This is not assigned until the first execution of draw(), so you must use this kwarg if you want to call get_window_extent() prior to the first draw(). For getting web page regions, it is simpler to call the method after saving the figure. The dpi used defaults to self.figure.dpi; the renderer dpi is irrelevant.

set_figure(fig)

update_positions(renderer)
    “Update the pixel positions of the annotated point and the text.

xyann

class matplotlib.text.OffsetFrom(artist, ref_coord, unit='points')
Bases: object
    Callable helper class for working with Annotation

    Parameters artist: Artist, BboxBase, or Transform
        The object to compute the offset from.

    ref_coord: length 2 sequence
        If artist is an Artist or BboxBase, this values is the location to of the offset origin in fractions of the artist bounding box.

        If artist is a transform, the offset origin is the transform applied to this value.

    unit: {'points', 'pixels'}
        The screen units to use (pixels or points) for the offset input.

get_unit()
    The unit for input to the transform used by __call__

set_unit(unit)
    The unit for input to the transform used by __call__

    Parameters unit: {'points', 'pixels'}

class matplotlib.text.Text(x=0, y=0, text='', color=None, verticalalignment='baseline', horizontalalignment='left', multialignment=None, fontproperties=None, rotation=None, linespacing=None, rotation_mode=None, usetex=None, wrap=False, **kwargs)
Bases: matplotlib.artist.Artist
Handle storing and drawing of text in window or data coordinates.

Create a `Text` instance at x, y with string `text`.

Valid kwargs are

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>agg_filter</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>alpha</code></td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td><code>animated</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>axes</code></td>
<td>an <code>Axes</code> instance</td>
</tr>
<tr>
<td><code>backgroundcolor</code></td>
<td>any matplotlib color</td>
</tr>
<tr>
<td><code>bbox</code></td>
<td>FancyBboxPatch prop dict</td>
</tr>
<tr>
<td><code>clip_box</code></td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td><code>clip_on</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>clip_path</code></td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td><code>color</code></td>
<td>any matplotlib color</td>
</tr>
<tr>
<td><code>contains</code></td>
<td>a callable function</td>
</tr>
<tr>
<td><code>family</code> or <code>fontfamily</code> or <code>fontname</code> or <code>name</code></td>
<td>[FONTNAME</td>
</tr>
<tr>
<td><code>figure</code></td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td><code>font_properties</code> or <code>font_properties</code></td>
<td>a <code>matplotlib.font_manager.FontProperties</code> instance</td>
</tr>
<tr>
<td><code>gid</code></td>
<td>an id string</td>
</tr>
<tr>
<td><code>horizontalalignment</code> or <code>ha</code></td>
<td>[‘center’</td>
</tr>
<tr>
<td><code>label</code></td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td><code>multialignment</code></td>
<td>[‘left’</td>
</tr>
<tr>
<td><code>path_effects</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>picker</code></td>
<td>[None</td>
</tr>
<tr>
<td><code>position</code></td>
<td>(x,y)</td>
</tr>
<tr>
<td><code>rasterized</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>rotation</code></td>
<td>[ angle in degrees</td>
</tr>
<tr>
<td><code>rotation_mode</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>size</code> or <code>fontsize</code></td>
<td>[size in points</td>
</tr>
<tr>
<td><code>sketch_params</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>snap</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>stretch</code> or <code>fontstretch</code></td>
<td>[a numeric value in range 0-1000</td>
</tr>
<tr>
<td><code>style</code> or <code>fontstyle</code></td>
<td>[‘normal’</td>
</tr>
<tr>
<td><code>text</code></td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td><code>transform</code></td>
<td><code>Transform</code> instance</td>
</tr>
<tr>
<td><code>url</code></td>
<td>a url string</td>
</tr>
<tr>
<td><code>usetex</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>variant</code> or <code>fontvariant</code></td>
<td>[‘normal’</td>
</tr>
<tr>
<td><code>verticalalignment</code> or <code>ma</code> or <code>va</code></td>
<td>[‘center’</td>
</tr>
<tr>
<td><code>visible</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>wrap</code></td>
<td>unknown</td>
</tr>
</tbody>
</table>
Table 51.1 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>float</td>
</tr>
<tr>
<td>y</td>
<td>float</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

**contains** *(mouseevent)*

Test whether the mouse event occurred in the patch.

In the case of text, a hit is true anywhere in the axis-aligned bounding-box containing the text.

Returns True or False.

**draw** *(artist, renderer, *args, **kwargs)*

Draws the *Text* object to the given *renderer*.

**get_bbox_patch** *

Return the bbox Patch object. Returns None if the FancyBboxPatch is not made.

**get_color** *

Return the color of the text

**get_family** *

Return the list of font families used for font lookup

**get_font_properties** *

alias for get_fontproperties

**get_fontfamily** *

alias for get_family

**get_fontname** *

alias for get_name

**get_fontproperties** *

Return the *FontProperties* object

**get_fontsize** *

alias for get_size

**get_fontstretch** *

alias for get_stretch

**get_fontstyle** *

alias for get_style

**get_fontvariant** *

alias for get_variant

**get_fontweight** *

alias for get_weight

**get_ha** *

alias for get_horizontalalignment
**get_horizontalalignment()**
Return the horizontal alignment as string. Will be one of ‘left’, ‘center’ or ‘right’.

**get_name()**
Return the font name as string

**get_position()**
Return the position of the text as a tuple (x, y)

**get_prop_tup**(renderer=None)
Return a hashable tuple of properties.
Not intended to be human readable, but useful for backends who want to cache derived information about text (e.g., layouts) and need to know if the text has changed.

**get_rotation()**
return the text angle as float in degrees

**get_rotation_mode()**
get text rotation mode

**get_size()**
Return the font size as integer

**get_stretch()**
Get the font stretch as a string or number

**get_style()**
Return the font style as string

**get_text()**
Get the text as string

**get_unitless_position()**
Return the unitless position of the text as a tuple (x, y)

**get_usetex()**
Return whether this Text object will render using TeX.
If the user has not manually set this value, it will default to the value of rcParams['text.usetex']

**get_va()**
alias for getverticalalignment()

**get_variant()**
Return the font variant as a string

**get_verticalalignment()**
Return the vertical alignment as string. Will be one of ‘top’, ‘center’, ‘bottom’ or ‘baseline’.

**get_weight()**
Get the font weight as string or number

**get_window_extent**(renderer=None, dpi=None)
Return a Bbox object bounding the text, in display units.
In addition to being used internally, this is useful for specifying clickable regions in a png file on a web page.

`renderer` defaults to the `_renderer` attribute of the text object. This is not assigned until the first execution of `draw()`, so you must use this kwarg if you want to call `get_window_extent()` prior to the first `draw()`. For getting web page regions, it is simpler to call the method after saving the figure.

`dpi` defaults to `self.figure.dpi`; the renderer `dpi` is irrelevant. For the web application, if `figure.dpi` is not the value used when saving the figure, then the value that was used must be specified as the `dpi` argument.

`get_wrap()`
Returns the wrapping state for the text.

```
static is_math_text(s, usetex=None)
```
Returns a cleaned string and a boolean flag. The flag indicates if the given string `s` contains any mathtext, determined by counting unescaped dollar signs. If no mathtext is present, the cleaned string has its dollar signs unescaped. If usetex is on, the flag always has the value “TeX”.

```
setBackgroundColor(color)
```
Set the background color of the text by updating the bbox.

See also:

```
set_bbox() To change the position of the bounding box.
```

ACCEPTS: any matplotlib color

```
set_bbox(rectprops)
```
Draw a bounding box around self. `rectprops` are any settable properties for a FancyBboxPatch, e.g., `facecolor='red', alpha=0.5`.

```
t.set_bbox(dict(facecolor='red', alpha=0.5))
```

The default boxstyle is ‘square’. The mutation scale of the FancyBboxPatch is set to the fontsize.

ACCEPTS: FancyBboxPatch prop dict

```
set_clip_box(clipbox)
```
Set the artist’s clip `Bbox`.

ACCEPTS: a `matplotlib.transforms.Bbox` instance

```
set_clip_on(b)
```
Set whether artist uses clipping.

When False artists will be visible out side of the axes which can lead to unexpected results.

ACCEPTS: [True | False]

```
set_clip_path(path, transform=None)
```
Set the artist’s clip path, which may be:

• a `Patch` (or subclass) instance
- a Path instance, in which case an optional Transform instance may be provided, which will be applied to the path before using it for clipping.

- None, to remove the clipping path

For efficiency, if the path happens to be an axis-aligned rectangle, this method will set the clipping box to the corresponding rectangle and set the clipping path to None.

ACCEPTS: [ (Path, Transform) | Patch | None ]

set_color(color)
Set the foreground color of the text

ACCEPTS: any matplotlib color

set_family(fontname)
Set the font family. May be either a single string, or a list of strings in decreasing priority. Each string may be either a real font name or a generic font class name. If the latter, the specific font names will be looked up in the matplotlibrc file.

ACCEPTS: [FONTNAME | ‘serif’ | ‘sans-serif’ | ‘cursive’ | ‘fantasy’ | ‘monospace’ ]

set_font_properties(fp)
alias for set_fontproperties

set_fontname(fontname)
alias for set_family

set_fontproperties(fp)
Set the font properties that control the text. fp must be a matplotlib.font_manager.FontProperties object.

ACCEPTS: a matplotlib.font_manager.FontProperties instance

set_fontsize(fontsize)
alias for set_size

set_fontstretch(stretch)
alias for set_stretch

set_fontstyle(fontstyle)
alias for set_style

set_fontvariant(variant)
alias for set_variant

set_fontweight(weight)
alias for set_weight

set_ha(align)
alias for set_horizontalalignment

set_horizontalalignment(align)
Set the horizontal alignment to one of

ACCEPTS: [ ‘center’ | ‘right’ | ‘left’ ]
set_linespacing(spacing)
    Set the line spacing as a multiple of the font size. Default is 1.2.
    
    ACCEPTS: float (multiple of font size)

set_ma(align)
    alias for set_verticalalignment

set_multialignment(align)
    Set the alignment for multiple lines layout. The layout of the bounding box of all the lines
is determined by the horizontalalignment and verticalalignment properties, but the multiline text
within that box can be
    
    ACCEPTS: ['left' | ‘right’ | ‘center’ ]

set_name(fontname)
    alias for set_family

set_position(xy)
    Set the (x, y) position of the text
    
    ACCEPTS: (x,y)

set_rotation(s)
    Set the rotation of the text
    
    ACCEPTS: [ angle in degrees | ‘vertical’ | ‘horizontal’ ]

set_rotation_mode(m)
    set text rotation mode. If “anchor”, the un-rotated text will first aligned according to their ha and
va, and then will be rotated with the alignment reference point as a origin. If None (default),
the text will be rotated first then will be aligned.

set_size(fontsize)
    Set the font size. May be either a size string, relative to the default font size, or an absolute font
size in points.
    
    ACCEPTS: [size in points | ‘xx-small’ | ‘x-small’ | ‘small’ | ‘medium’ | ‘large’ | ‘x-large’ |’xx-large’ ]

set_stretch(stretch)
    Set the font stretch (horizontal condensation or expansion).
    

set_style(fontstyle)
    Set the font style.
    
    ACCEPTS: [ ‘normal’ | ‘italic’ | ‘oblique’ ]

set_text(s)
    Set the text string s
    
    It may contain newlines (\n) or math in LaTeX syntax.
 ACCEPTS: string or anything printable with ‘%s’ conversion.

set_usetex(usetex)
  Set this Text object to render using TeX (or not).
  If None is given, the option will be reset to use the value of rcParams['text.usetex']

set_va(align)
  alias for set_verticalalignment

set_variant(variant)
  Set the font variant, either ‘normal’ or ‘small-caps’.
  ACCEPTS: [‘normal’ | ‘small-caps’]

set_verticalalignment(align)
  Set the vertical alignment
  ACCEPTS: [‘center’ | ‘top’ | ‘bottom’ | ‘baseline’]

set_weight(weight)
  Set the font weight.

set_wrap(wrap)
  Sets the wrapping state for the text.

set_x(x)
  Set the x position of the text
  ACCEPTS: float

set_y(y)
  Set the y position of the text
  ACCEPTS: float

update(kwargs)
  Update properties from a dictionary.

update_bbox_position_size(renderer)
  Update the location and the size of the bbox. This method should be used when the position and size of the bbox needs to be updated before actually drawing the bbox.

update_from(other)
  Copy properties from other to self

zorder = 3
class matplotlib.text.TextWithDash(x=0, y=0, text='', color=None, verticalalignment='center', horizontalalignment='center', multialignment=None, fontproperties=None, rotation=None, linespacing=None, dashlength=0.0, dashdirection=0, dashrotation=None, dashpad=3, dashpush=0)

Bases: matplotlib.text.Text

This is basically a Text with a dash (drawn with a Line2D) before/after it. It is intended to be a drop-in replacement for Text, and should behave identically to it when dashlength = 0.0.

The dash always comes between the point specified by set_position() and the text. When a dash exists, the text alignment arguments (horizontalalignment, verticalalignment) are ignored.

dashlength is the length of the dash in canvas units. (default = 0.0).

dashdirection is one of 0 or 1, where 0 draws the dash after the text and 1 before. (default = 0).

dashrotation specifies the rotation of the dash, and should generally stay None. In this case get_dashrotation() returns get_rotation(). (i.e., the dash takes its rotation from the text's rotation). Because the text center is projected onto the dash, major deviations in the rotation cause what may be considered visually unappealing results. (default = None)

dashpad is a padding length to add (or subtract) space between the text and the dash, in canvas units. (default = 3)

dashpush “pushes” the dash and text away from the point specified by set_position() by the amount in canvas units. (default = 0)

Note: The alignment of the two objects is based on the bounding box of the Text, as obtained by get_window_extent(). This, in turn, appears to depend on the font metrics as given by the rendering backend. Hence the quality of the “centering” of the label text with respect to the dash varies depending on the backend used.

Note: I’m not sure that I got the get_window_extent() right, or whether that’s sufficient for providing the object bounding box.

draw(renderer)
   Draw the TextWithDash object to the given renderer.

get_dashdirection()
   Get the direction dash. 1 is before the text and 0 is after.

get_dashlength()
   Get the length of the dash.

get_dashpad()
   Get the extra spacing between the dash and the text, in canvas units.

get_dashpush()
   Get the extra spacing between the dash and the specified text position, in canvas units.
get_dashrotation()
Get the rotation of the dash in degrees.

get_figure()
return the figure instance the artist belongs to

get_position()
Return the position of the text as a tuple ($x$, $y$)

get_prop_tup(renderer=None)
Return a hashable tuple of properties.
Not intended to be human readable, but useful for backends who want to cache derived information about text (e.g., layouts) and need to know if the text has changed.

get_unitless_position()
Return the unitless position of the text as a tuple ($x$, $y$)

get_window_extent(renderer=None)
Return a Bbox object bounding the text, in display units.
In addition to being used internally, this is useful for specifying clickable regions in a png file on a web page.

renderer defaults to the _renderer attribute of the text object. This is not assigned until the first execution of draw(), so you must use this kwarg if you want to call get_window_extent() prior to the first draw(). For getting web page regions, it is simpler to call the method after saving the figure.

set_dashdirection(dd)
Set the direction of the dash following the text. 1 is before the text and 0 is after. The default is 0, which is what you’d want for the typical case of ticks below and on the left of the figure.

ACCEPTS: int (1 is before, 0 is after)

set_dashlength(dl)
Set the length of the dash.

ACCEPTS: float (canvas units)

set_dashpad(dp)
Set the “pad” of the TextWithDash, which is the extra spacing between the dash and the text, in canvas units.

ACCEPTS: float (canvas units)

set_dashpush(dp)
Set the “push” of the TextWithDash, which is the extra spacing between the beginning of the dash and the specified position.

ACCEPTS: float (canvas units)

set_dashrotation(dr)
Set the rotation of the dash, in degrees

ACCEPTS: float (degrees)
set_figure(fig)
    Set the figure instance the artist belong to.
    ACCEPTS: a matplotlib.figure.Figure instance

set_position(xy)
    Set the (x, y) position of the TextWithDash.
    ACCEPTS: (x, y)

set_transform(t)
    Set the matplotlib.transforms.Transform instance used by this artist.
    ACCEPTS: a matplotlib.transforms.Transform instance

set_x(x)
    Set the x position of the TextWithDash.
    ACCEPTS: float

set_y(y)
    Set the y position of the TextWithDash.
    ACCEPTS: float

update_coords(renderer)
    Computes the actual x, y coordinates for text based on the input x, y and the dashlength. Since the rotation is with respect to the actual canvas’s coordinates we need to map back and forth.

matplotlib.text.get_rotation(rotation)
    Return the text angle as float. The returned angle is between 0 and 360 deg.
    rotation may be ‘horizontal’, ‘vertical’, or a numeric value in degrees.
52.1 matplotlib.ticker

52.1.1 Tick locating and formatting

This module contains classes to support completely configurable tick locating and formatting. Although the locators know nothing about major or minor ticks, they are used by the Axis class to support major and minor tick locating and formatting. Generic tick locators and formatters are provided, as well as domain specific custom ones.

Default Formatter

The default formatter identifies when the x-data being plotted is a small range on top of a large offset. To reduce the chances that the ticklabels overlap the ticks are labeled as deltas from a fixed offset. For example:

```python
ax.plot(np.arange(2000, 2010), range(10))
```

will have tick of 0-9 with an offset of +2e3. If this is not desired turn off the use of the offset on the default formatter:

```python
ax.get_xaxis().get_major_formatter().set_useOffset(False)
```

set the rcParam `axes.formatter.useoffset=False` to turn it off globally, or set a different formatter.

Tick locating

The Locator class is the base class for all tick locators. The locators handle autoscaling of the view limits based on the data limits, and the choosing of tick locations. A useful semi-automatic tick locator is MultipleLocator. You initialize this with a base, e.g., 10, and it picks axis limits and ticks that are multiples of your base.

The Locator subclasses defined here are

- **NullLocator**  
  No ticks

- **FixedLocator**  
  Tick locations are fixed
IndexLocator locator for index plots (e.g., where \( x = \text{range}(\text{len}(y)) \))

LinearLocator evenly spaced ticks from min to max

LogLocator logarithmically ticks from min to max

SymmetricalLogLocator locator for use with with the symlog norm, works like the LogLocator for the part outside of the threshold and add 0 if inside the limits

MultipleLocator

ticks and range are a multiple of base; either integer or float

OldAutoLocator choose a MultipleLocator and dynamically reassign it for intelligent ticking during navigation

MaxNLocator finds up to a max number of ticks at nice locations

AutoLocator MaxNLocator with simple defaults. This is the default tick locator for most plotting.

AutoMinorLocator locator for minor ticks when the axis is linear and the major ticks are uniformly spaced. It subdivides the major tick interval into a specified number of minor intervals, defaulting to 4 or 5 depending on the major interval.

LogitLocator Locator for logit scaling.

There are a number of locators specialized for date locations - see the dates module

You can define your own locator by deriving from Locator. You must override the \_call\_ method, which returns a sequence of locations, and you will probably want to override the autoscale method to set the view limits from the data limits.

If you want to override the default locator, use one of the above or a custom locator and pass it to the x or y axis instance. The relevant methods are:

```python
ax.xaxis.set_major_locator( xmajorLocator )
ax.xaxis.set_minor_locator( xminorLocator )
ax.yaxis.set_major_locator( ymajorLocator )
ax.yaxis.set_minor_locator( yminorLocator )
```

The default minor locator is the NullLocator, e.g., no minor ticks on by default.

**Tick formatting**

Tick formatting is controlled by classes derived from Formatter. The formatter operates on a single tick value and returns a string to the axis.

NullFormatter No labels on the ticks

IndexFormatter Set the strings from a list of labels

FixedFormatter Set the strings manually for the labels

FuncFormatter User defined function sets the labels

StrMethodFormatter Use string format method
FormatStrFormatter  Use an old-style sprintf format string

ScalarFormatter  Default formatter for scalars: autopick the format string

LogFormatter  Formatter for log axes

LogFormatterExponent  Format values for log axis using exponent = log_base(value).

LogFormatterMathtext  Format values for log axis using exponent = log_base(value) using Math text.

LogFormatterSciNotation  Format values for log axis using scientific notation.

LogitFormatter  Probability formatter.

You can derive your own formatter from the Formatter base class by simply overriding the __call__ method. The formatter class has access to the axis view and data limits.

To control the major and minor tick label formats, use one of the following methods:

```python
tax.xaxis.set_major_formatter( xmajorFormatter )
tax.xaxis.set_minor_formatter( xminorFormatter )
tax.yaxis.set_major_formatter( ymajorFormatter )
tax.yaxis.set_minor_formatter( yminorFormatter )
```

See `pylab_examples example code: major_minor_demo1.py` for an example of setting major and minor ticks. See the `matplotlib.dates` module for more information and examples of using date locators and formatters.

class matplotlib.ticker.TickHelper
    Bases: object

    axis = None

    create_dummy_axis(**kwargs)

    set_axis(axis)

    set_bounds(vmin, vmax)

    set_data_interval(vmin, vmax)

    set_view_interval(vmin, vmax)

class matplotlib.ticker.Formatter
    Bases: matplotlib.ticker.TickHelper

    Create a string based on a tick value and location.
fix_minus(s)
Some classes may want to replace a hyphen for minus with the proper unicode symbol (U+2212) for typographical correctness. The default is to not replace it.

Note, if you use this method, e.g., in format_data() or call, you probably don’t want to use it for format_data_short() since the toolbar uses this for interactive coord reporting and I doubt we can expect GUIs across platforms will handle the unicode correctly. So for now the classes that override fix_minus() should have an explicit format_data_short() method

format_data(value)
Returns the full string representation of the value with the position unspecified.

format_data_short(value)
Return a short string version of the tick value.

Defaults to the position-independent long value.

get_offset()

locs = []

set_locs(locs)

class matplotlib.ticker.FixedFormatter(seq)
Bases: matplotlib.ticker.Formatter
Return fixed strings for tick labels based only on position, not value.
Set the sequence of strings that will be used for labels.

get_offset()

set_offset_string(ofs)

class matplotlib.ticker.NullFormatter
Bases: matplotlib.ticker.Formatter
Always return the empty string.

class matplotlib.ticker.FuncFormatter(func)
Bases: matplotlib.ticker.Formatter
Use a user-defined function for formatting.

The function should take in two inputs (a tick value x and a position pos), and return a string containing the corresponding tick label.

class matplotlib.ticker.FormatStrFormatter(fmt)
Bases: matplotlib.ticker.Formatter
Use an old-style (‘%’ operator) format string to format the tick.
The format string should have a single variable format (%) in it. It will be applied to the value (not the position) of the tick.

```python
class matplotlib.ticker.StrMethodFormatter(fmt):
    Bases: matplotlib.ticker.Formatter

    Use a new-style format string (as used by str.format()) to format the tick.

    The field used for the value must be labeled `x` and the field used for the position must be labeled `pos`.
```

```python
class matplotlib.ticker.ScalarFormatter(useOffset=None, useMathText=None, useLocale=None):
    Bases: matplotlib.ticker.Formatter

    Format tick values as a number.

    Tick value is interpreted as a plain old number. If useOffset==True and the data range is much smaller than the data average, then an offset will be determined such that the tick labels are meaningful. Scientific notation is used for data < 10^-n or data >= 10^m, where n and m are the power limits set using set_powerlimits((n,m)). The defaults for these are controlled by the axes. formatter.limits rc parameter.

    fix_minus(s)
        Replace hyphens with a unicode minus.

    format_data(value)
        Return a formatted string representation of a number.

    format_data_short(value)
        Return a short formatted string representation of a number.

    get_offset()
        Return scientific notation, plus offset.

    get_useLocale()

    get_useOffset()

    pprint_val(x)

    set_locs(locs)
        Set the locations of the ticks.

    set_powerlimits(lims)
        Sets size thresholds for scientific notation.

        lims is a two-element sequence containing the powers of 10 that determine the switchover threshold. Numbers below 10**lims[0] and above 10**lims[1] will be displayed in scientific notation.

        For example, formatter.set_powerlimits((-3, 4)) sets the pre-2007 default in which scientific notation is used for numbers less than 1e-3 or greater than 1e4.

    See also:
```
Method `set_scientific()`

`set_scientific(b)`

Turn scientific notation on or off.

See also:

Method `set_powerlimits()`

`set_useLocale(val)`

`set_useOffset(val)`

`useLocale`

`useOffset`

class `matplotlib.ticker.LogFormatter`:

```
class matplotlib.ticker.LogFormatter(base=10.0, labelOnlyBase=False, minor_thresholds=None, linthresh=None)

Bases: matplotlib.ticker.Formatter

Base class for formatting ticks on a log or symlog scale.

It may be instantiated directly, or subclassed.

Parameters

base : float, optional, default: 10.

Base of the logarithm used in all calculations.

labelOnlyBase : bool, optional, default: False

If True, label ticks only at integer powers of base. This is normally True for major ticks and False for minor ticks.

minor_thresholds : (subset, all), optional, default: (1, 0.4)

If labelOnlyBase is False, these two numbers control the labeling of ticks that are not at integer powers of base; normally these are the minor ticks. The controlling parameter is the log of the axis data range. In the typical case where base is 10 it is the number of decades spanned by the axis, so we can call it ‘numdec’. If numdec ≤ all, all minor ticks will be labeled. If all < numdec ≤ subset, then only a subset of minor ticks will be labeled, so as to avoid crowding. If numdec > subset then no minor ticks will be labeled.

linthresh : None or float, optional, default: None

If a symmetric log scale is in use, its linthresh parameter must be supplied here.
```
Notes

The `set_locs` method must be called to enable the subsetting logic controlled by the `minor_thresholds` parameter.

In some cases such as the colorbar, there is no distinction between major and minor ticks; the tick locations might be set manually, or by a locator that puts ticks at integer powers of base and at intermediate locations. For this situation, disable the `minor_thresholds` logic by using `minor_thresholds=(np.inf, np.inf)`, so that all ticks will be labeled.

To disable labeling of minor ticks when `labelOnlyBase` is False, use `minor_thresholds=(0, 0)`. This is the default for the “classic” style.

Examples

To label a subset of minor ticks when the view limits span up to 2 decades, and all of the ticks when zoomed in to 0.5 decades or less, use `minor_thresholds=(2, 0.5)`.

To label all minor ticks when the view limits span up to 1.5 decades, use `minor_thresholds=(1.5, 1.5)`.

`base(base)`

change the `base` for labeling.

```
Warning: Should always match the base used for LogLocator
```

`format_data(value)`

`format_data_short(value)`

Return a short formatted string representation of a number.

`label_minor(labelOnlyBase)`

Switch minor tick labeling on or off.

Parameters

`labelOnlyBase : bool`

If True, label ticks only at integer powers of base.

`pprint_val(x, d)`

`set_locs(locs=None)`

Use axis view limits to control which ticks are labeled.

The `locs` parameter is ignored in the present algorithm.

```
class matplotlib.ticker.LogFormatterExponent(base=10.0, labelOnlyBase=False, minor_thresholds=None, linthresh=None)
```

Bases: `matplotlib.ticker.LogFormatter`

Format values for log axis using `exponent = log_base(value)`. 
class `matplotlib.ticker.LogFormatterMathtext`(*base=10.0, *labelOnlyBase=False, *minor_thresholds=None, *linthresh=None*)

Bases: `matplotlib.ticker.LogFormatter`

Format values for log axis using `exponent = log_base(value)`.

class `matplotlib.ticker.Locator`

Bases: `matplotlib.ticker.TickHelper`

Determine the tick locations;

Note, you should not use the same locator between different `Axis` because the locator stores references to the Axis data and view limits

```
MAXTICKS = 1000
```

`autoscale()`

autoscale the view limits

`pan(numsteps)`

Pan numticks (can be positive or negative)

`raise_if_exceeds(locs)`

raise a RuntimeError if Locator attempts to create more than MAXTICKS locs

`refresh()`

refresh internal information based on current lim

`set_params(**kwargs)`

Do nothing, and raise a warning. Any locator class not supporting the set_params() function will call this.

`tick_values(vmin, vmax)`

Return the values of the located ticks given `vmin` and `vmax`.

***Note:*** To get tick locations with the vmin and vmax values defined automatically for the associated axis simply call the Locator instance:

```
>>> print((type(loc)))
<type 'Locator'>
>>> print((loc()))
[1, 2, 3, 4]
```

`view_limits(vmin, vmax)`

select a scale for the range from vmin to vmax

Normally this method is overridden by subclasses to change locator behaviour.

`zoom(direction)`

Zoom in/out on axis; if direction is >0 zoom in, else zoom out

class `matplotlib.ticker.IndexLocator(base, offset)`

Bases: `matplotlib.ticker.Locator`
Place a tick on every multiple of some base number of points plotted, e.g., on every 5th point. It is assumed that you are doing index plotting; i.e., the axis is 0, len(data). This is mainly useful for x ticks.

place ticks on the i-th data points where (i-offset)%base==0

set_params(base=None, offset=None)
    Set parameters within this locator

tick_values(vmin, vmax)

class matplotlib.ticker.FixedLocator(locs, nbins=None)
    Bases: matplotlib.ticker.Locator
    Tick locations are fixed. If nbins is not None, the array of possible positions will be subsampled to keep the number of ticks <= nbins +1. The subsampling will be done so as to include the smallest absolute value; for example, if zero is included in the array of possibilities, then it is guaranteed to be one of the chosen ticks.

set_params(nbins=None)
    Set parameters within this locator.

tick_values(vmin, vmax)
    ” Return the locations of the ticks.

    Note: Because the values are fixed, vmin and vmax are not used in this method.

class matplotlib.ticker.NullLocator
    Bases: matplotlib.ticker.Locator
    No ticks

tick_values(vmin, vmax)
    ” Return the locations of the ticks.

    Note: Because the values are Null, vmin and vmax are not used in this method.

class matplotlib.ticker.LinearLocator(numticks=None, presets=None)
    Bases: matplotlib.ticker.Locator
    Determine the tick locations

    The first time this function is called it will try to set the number of ticks to make a nice tick partitioning. Thereafter the number of ticks will be fixed so that interactive navigation will be nice

    Use presets to set locs based on lom. A dict mapping vmin, vmax->locs

set_params(numticks=None, presets=None)
    Set parameters within this locator.

tick_values(vmin, vmax)

view_limits(vmin, vmax)
Try to choose the view limits intelligently

class matplotlib.ticker.LogLocator(base=10.0, subs=(1.0,), numdecs=4, numticks=None)
Bases: matplotlib.ticker.Locator

Determine the tick locations for log axes

Place ticks on the locations: \[ \text{subs}[j] \times \text{base}^i \]

Parameters subs: None, string, or sequence of float, optional, default (1.0,)

Gives the multiples of integer powers of the base at which to place ticks. The default places ticks only at integer powers of the base. The permitted string values are 'auto' and 'all', both of which use an algorithm based on the axis view limits to determine whether and how to put ticks between integer powers of the base. With 'auto', ticks are placed only between integer powers; with 'all', the integer powers are included. A value of None is equivalent to 'auto'.

base(base)
set the base of the log scaling (major tick every \text{base}^i, i integer)

nonsingular(vmin, vmax)

set_params(base=None, subs=None, numdecs=None, numticks=None)
Set parameters within this locator.

subs(subs)
set the minor ticks for the log scaling every \text{base}^i \times \text{subs}[j]

tick_values(vmin, vmax)

view_limits(vmin, vmax)
Try to choose the view limits intelligently

class matplotlib.ticker.AutoLocator
Bases: matplotlib.ticker.MaxNLocator

class matplotlib.ticker.MultipleLocator(base=1.0)
Bases: matplotlib.ticker.Locator

Set a tick on every integer that is multiple of base in the view interval

set_params(base)
Set parameters within this locator.

tick_values(vmin, vmax)

view_limits(dmin, dmax)
Set the view limits to the nearest multiples of base that contain the data

class matplotlib.ticker.MaxNLocator(*args, **kwargs)
Bases: matplotlib.ticker.Locator
Select no more than N intervals at nice locations.

Keyword args:

nbins Maximum number of intervals; one less than max number of ticks. If the string 'auto', the number of bins will be automatically determined based on the length of the axis.

steps Sequence of nice numbers starting with 1 and ending with 10; e.g., [1, 2, 4, 5, 10]

integer If True, ticks will take only integer values, provided at least min_n_ticks integers are found within the view limits.

symmetric If True, autoscaling will result in a range symmetric about zero.

prune ['lower' | 'upper' | 'both' | None] Remove edge ticks – useful for stacked or ganged plots where the upper tick of one axes overlaps with the lower tick of the axes above it, primarily when rcParams['axes.autolimit_mode'] is 'round_numbers'. If prune='lower', the smallest tick will be removed. If prune='upper', the largest tick will be removed. If prune='both', the largest and smallest ticks will be removed. If prune=None, no ticks will be removed.

min_n_ticks Relax nbins and integer constraints if necessary to obtain this minimum number of ticks.

bin_boundaries(vmin, vmax)

Deprecated since version 2.0: The bin_boundaries function was deprecated in version 2.0.

default_params = {'nbins': 10, 'steps': None, 'integer': False, 'symmetric': False, 'prune': None, 'min_n_ticks': 2}

set_params(**kwargs)

Set parameters within this locator.

tick_values(vmin, vmax)

view_limits(dmin, dmax)

class matplotlib.ticker.AutoMinorLocator(n=None)

Bases: matplotlib.ticker.Locator

Dynamically find minor tick positions based on the positions of major ticks. Assumes the scale is linear and major ticks are evenly spaced.

n is the number of subdivisions of the interval between major ticks; e.g., n=2 will place a single minor tick midway between major ticks.

If n is omitted or None, it will be set to 5 or 4.

tick_values(vmin, vmax)

class matplotlib.ticker.SymmetricalLogLocator(transform=None, subs=None, linthresh=None, base=None)

Bases: matplotlib.ticker.Locator

52.1. matplotlib.ticker
Determine the tick locations for symmetric log axes
place ticks on the location = base**i*subs[j]

set_params(subs=None, numticks=None)
Set parameters within this locator.
tick_values(vmin, vmax)

view_limits(vmin, vmax)
Try to choose the view limits intelligently
53.1 matplotlib.tight_layout

This module provides routines to adjust subplot params so that subplots are nicely fit in the figure. In doing so, only axis labels, tick labels, axes titles and offsetboxes that are anchored to axes are currently considered.

Internally, it assumes that the margins (left_margin, etc.) which are differences between ax.get_tightbbox and ax.bbox are independent of axes position. This may fail if Axes.adjustable is datalim. Also, This will fail for some cases (for example, left or right margin is affected by xlabel).

matplotlib.tight_layout.auto_adjust_subplotpars(fig, renderer, nrows_ncols, num1num2_list, subplot_list, ax_bbox_list=None, pad=1.08, h_pad=None, w_pad=None, rect=None)

Return a dictionary of subplot parameters so that spacing between subplots are adjusted. Note that this function ignore geometry information of subplot itself, but uses what is given by nrows_ncols and num1num2_list parameteres. Also, the results could be incorrect if some subplots have adjustable=datalim.

Parameters:

nrows_ncols number of rows and number of columns of the grid.

num1num2_list list of numbers specifying the area occupied by the subplot

subplot_list list of subplots that will be used to calculate optimal subplot_params.

pad [float] padding between the figure edge and the edges of subplots, as a fraction of the font-size.

h_pad, w_pad [float]

padding (height/width) between edges of adjacent subplots. Defaults to pad_inches.

rect [left, bottom, right, top] in normalized (0, 1) figure coordinates.

matplotlib.tight_layout.get_renderer(fig)

matplotlib.tight_layout.get_subplotspec_list(axes_list, grid_spec=None)

Return a list of subplotspec from the given list of axes. For an instance of axes that does not support subplotspec, None is inserted in the list.
If grid_spec is given, None is inserted for those not from the given grid_spec.

```python
matplotlib.tight_layout.get_tight_layout_figure(fig, axes_list, subplotspec_list, renderer, pad=1.08, h_pad=None, w_pad=None, rect=None)
```

Return subplot parameters for tight-layouted-figure with specified padding.

Parameters:

- `fig` : figure instance
- `axes_list` : a list of axes
- `subplotspec_list` : a list of subplotspec associated with each axes in `axes_list`
- `renderer` : renderer instance
- `pad` : [float] padding between the figure edge and the edges of subplots, as a fraction of the font-size.
- `h_pad`, `w_pad` : [float] padding (height/width) between edges of adjacent subplots. Defaults to `pad_inches`.
- `rect` : [if rect is given, it is interpreted as a rectangle] (left, bottom, right, top) in the normalized figure coordinate that the whole subplots area (including labels) will fit into. Default is (0, 0, 1, 1).
54.1 matplotlib.transforms

matplotlib includes a framework for arbitrary geometric transformations that is used to determine the final position of all elements drawn on the canvas.

Transforms are composed into trees of TransformNode objects whose actual value depends on their children. When the contents of children change, their parents are automatically invalidated. The next time an invalidated transform is accessed, it is recomputed to reflect those changes. This invalidation/caching approach prevents unnecessary recomputations of transforms, and contributes to better interactive performance.

For example, here is a graph of the transform tree used to plot data to the graph:
The framework can be used for both affine and non-affine transformations. However, for speed, we want use the backend renderers to perform affine transformations whenever possible. Therefore, it is possible to perform just the affine or non-affine part of a transformation on a set of data. The affine is always assumed to occur after the non-affine. For any transform:

\[
\text{full transform} = \text{non-affine part} + \text{affine part}
\]

The backends are not expected to handle non-affine transformations themselves.
class matplotlib.transforms.TransformNode(shorthand_name=None)
    Bases: object

    TransformNode is the base class for anything that participates in the transform tree and needs to 
    invalidate its parents or be invalidated. This includes classes that are not really transforms, such as 
    bounding boxes, since some transforms depend on bounding boxes to compute their values.

    Creates a new TransformNode.

    shorthand_name - a string representing the “name” of this transform. The name carries no sig- 
    nificance other than to improve the readability of str(transform) when DEBUG=True.

    frozen()
    Returns a frozen copy of this transform node. The frozen copy will not update when its children 
    change. Useful for storing a previously known state of a transform where copy.deepcopy() 
    might normally be used.

    invalidate()
    Invalidate this TransformNode and triggers an invalidation of its ancestors. Should be called 
    any time the transform changes.

    pass_through = False
    If pass_through is True, all ancestors will always be invalidated, even if ‘self’ is already invalid.

    set_children(*children)
    Set the children of the transform, to let the invalidation system know which transforms can 
    invalidate this transform. Should be called from the constructor of any transforms that depend 
    on other transforms.

class matplotlib.transforms.BboxBase(shorthand_name=None)
    Bases: matplotlib.transforms.TransformNode

    This is the base class of all bounding boxes, and provides read-only access to its data. A mutable 
    bounding box is provided by the Bbox class.

    The canonical representation is as two points, with no restrictions on their ordering. Convenience 
    properties are provided to get the left, bottom, right and top edges and width and height, but these are 
    not stored explicitly.

    Creates a new TransformNode.

    shorthand_name - a string representing the “name” of this transform. The name carries no sig- 
    nificance other than to improve the readability of str(transform) when DEBUG=True.

    anchored(c, container=None)
    Return a copy of the Bbox, shifted to position c within a container.

    c: may be either:

    • a sequence (cx, cy) where cx and cy range from 0 to 1, where 0 is left or bottom and 1 is 
      right or top

    • a string: - ‘C’ for centered - ‘S’ for bottom-center - ‘SE’ for bottom-left - ‘E’ for left - etc.

    Optional argument container is the box within which the Bbox is positioned; it defaults to the 
    initial Bbox.
bounds
   (property) Returns \((x_0, y_0, \text{width}, \text{height})\).

contains\((x, y)\)
   Returns \(True\) if \((x, y)\) is a coordinate inside the bounding box or on its edge.

containsx\((x)\)
   Returns True if \(x\) is between or equal to \(x_0\) and \(x_1\).

containsy\((y)\)
   Returns True if \(y\) is between or equal to \(y_0\) and \(y_1\).

corners()
   Return an array of points which are the four corners of this rectangle. For example, if this \texttt{Bbox}
   is defined by the points \((a, b)\) and \((c, d)\), \texttt{corners()} returns \((a, b), (a, d), (c, b)\) and \((c, d)\).

count_contains\((\text{vertices})\)
   Count the number of vertices contained in the \texttt{Bbox}.
   \texttt{vertices} is a Nx2 Numpy array.

count_overlaps\((\text{bboxes})\)
   Count the number of bounding boxes that overlap this one.
   \texttt{bboxes} is a sequence of \texttt{BboxBase} objects

expanded\((sw, sh)\)
   Return a new \texttt{Bbox} which is this \texttt{Bbox} expanded around its center by the given factors \(sw\) and \(sh\).

extents
   (property) Returns \((x_0, y_0, x_1, y_1)\).

frozen()
   \texttt{TransformNode} is the base class for anything that participates in the transform tree and needs
   to invalidate its parents or be invalidated. This includes classes that are not really transforms,
   such as bounding boxes, since some transforms depend on bounding boxes to compute their
   values.

fully_contains\((x, y)\)
   Returns True if \((x, y)\) is a coordinate inside the bounding box, but not on its edge.

fully_containsx\((x)\)
   Returns True if \(x\) is between but not equal to \(x_0\) and \(x_1\).

fully_containsy\((y)\)
   Returns True if \(y\) is between but not equal to \(y_0\) and \(y_1\).

fully_overlaps\((\text{other})\)
   Returns True if this bounding box overlaps with the given bounding box \texttt{other}, but not on its
   edge alone.

height
   (property) The height of the bounding box. It may be negative if \(y_1 < y_0\).
static intersection(bbox1, bbox2)
Return the intersection of the two bboxes or None if they do not intersect.

Implements the algorithm described at:
http://www.tekpool.com/node/2687

intervalx
(property) intervalx is the pair of x coordinates that define the bounding box. It is not guaranteed to be sorted from left to right.

intervaly
(property) intervale is the pair of y coordinates that define the bounding box. It is not guaranteed to be sorted from bottom to top.

inverse_transformed(transform)
Return a new Bbox object, statically transformed by the inverse of the given transform.

is_unit()
Returns True if the Bbox is the unit bounding box from (0, 0) to (1, 1).

max
(property) max is the top-right corner of the bounding box.

min
(property) min is the bottom-left corner of the bounding box.

overlaps(other)
Returns True if this bounding box overlaps with the given bounding box other.

p0
(property) p0 is the first pair of (x, y) coordinates that define the bounding box. It is not guaranteed to be the bottom-left corner. For that, use min.

p1
(property) p1 is the second pair of (x, y) coordinates that define the bounding box. It is not guaranteed to be the top-right corner. For that, use max.

padded(p)
Return a new Bbox that is padded on all four sides by the given value.

rotated(radians)
Return a new bounding box that bounds a rotated version of this bounding box by the given radians. The new bounding box is still aligned with the axes, of course.

shrunk(mx, my)
Return a copy of the Bbox, shrunk by the factor mx in the x direction and the factor my in the y direction. The lower left corner of the box remains unchanged. Normally mx and my will be less than 1, but this is not enforced.

shrunk_to_aspect(box_aspect, container=None, fig_aspect=1.0)
Return a copy of the Bbox, shrunk so that it is as large as it can be while having the desired aspect ratio, box_aspect. If the box coordinates are relative—that is, fractions of a larger box such as a figure—then the physical aspect ratio of that figure is specified with fig_aspect, so that box_aspect can also be given as a ratio of the absolute dimensions, not the relative dimensions.
size
  (property) The width and height of the bounding box. May be negative, in the same way as
  width and height.

splitx(*args)
e.g., bbox.splitx(f1, f2, ...)
  Returns a list of new Bbox objects formed by splitting the original one with vertical lines at
  fractional positions f1, f2, ...

splity(*args)
e.g., bbox.splity(f1, f2, ...)
  Returns a list of new Bbox objects formed by splitting the original one with horizontal lines at
  fractional positions f1, f2, ...

transformed(transform)
  Return a new Bbox object, statically transformed by the given transform.

translated(tx, ty)
  Return a copy of the Bbox, statically translated by tx and ty.

static union(bboxes)
  Return a Bbox that contains all of the given bboxes.

width
  (property) The width of the bounding box. It may be negative if x1 < x0.

x0
  (property) x0 is the first of the pair of x coordinates that define the bounding box. x0 is not
  guaranteed to be less than x1. If you require that, use xmin.

x1
  (property) x1 is the second of the pair of x coordinates that define the bounding box. x1 is not
  guaranteed to be greater than x0. If you require that, use xmax.

xmax
  (property) xmax is the right edge of the bounding box.

xmin
  (property) xmin is the left edge of the bounding box.

y0
  (property) y0 is the first of the pair of y coordinates that define the bounding box. y0 is not
  guaranteed to be less than y1. If you require that, use ymin.

y1
  (property) y1 is the second of the pair of y coordinates that define the bounding box. y1 is not
  guaranteed to be greater than y0. If you require that, use ymax.

ymax
  (property) ymax is the top edge of the bounding box.

ymin
  (property) ymin is the bottom edge of the bounding box.
class matplotlib.transforms.Bbox(points, **kwargs)
    Bases: matplotlib.transforms.BboxBase

A mutable bounding box.

points: a 2x2 numpy array of the form [[x0, y0], [x1, y1]]

If you need to create a Bbox object from another form of data, consider the static methods unit(), from_bounds() and from_extents().

static from_bounds(x0, y0, width, height)
    (staticmethod) Create a new Bbox from x0, y0, width and height.
    width and height may be negative.

static from_extents(*args)
    (staticmethod) Create a new Bbox from left, bottom, right and top.
    The y-axis increases upwards.

get_points()
    Get the points of the bounding box directly as a numpy array of the form: [[x0, y0], [x1, y1]].

ignore(value)
    Set whether the existing bounds of the box should be ignored by subsequent calls to update_from_data_xy().

    value:
    • When True, subsequent calls to update_from_data_xy() will ignore the existing bounds of the Bbox.
    • When False, subsequent calls to update_from_data_xy() will include the existing bounds of the Bbox.

mutated()
    return whether the bbox has changed since init

mutatedx()
    return whether the x-limits have changed since init

mutatedy()
    return whether the y-limits have changed since init

static null()
    (staticmethod) Create a new null Bbox from (inf, inf) to (-inf, -inf).

set(other)
    Set this bounding box from the “frozen” bounds of another Bbox.

set_points(points)
    Set the points of the bounding box directly from a numpy array of the form: [[x0, y0], [x1, y1]].
    No error checking is performed, as this method is mainly for internal use.

static unit()
    (staticmethod) Create a new unit Bbox from (0, 0) to (1, 1).
update_from_data(x, y, ignore=None)
Deprecated since version 2.0: The update_from_data function was deprecated in version 2.0. Use update_from_data_xy instead.

Update the bounds of the Bbox based on the passed in data. After updating, the bounds will have positive width and height; x0 and y0 will be the minimal values.

x: a numpy array of x-values
y: a numpy array of y-values

ignore:
  • when True, ignore the existing bounds of the Bbox.
  • when False, include the existing bounds of the Bbox.
  • when None, use the last value passed to ignore().

update_from_data_xy(xy, ignore=None, updatex=True, updatey=True)
Update the bounds of the Bbox based on the passed in data. After updating, the bounds will have positive width and height; x0 and y0 will be the minimal values.

xy: a numpy array of 2D points

ignore:
  • when True, ignore the existing bounds of the Bbox.
  • when False, include the existing bounds of the Bbox.
  • when None, use the last value passed to ignore().

updatex: when True, update the x values
updatey: when True, update the y values

update_from_path(path, ignore=None, updatex=True, updatey=True)
Update the bounds of the Bbox based on the passed in data. After updating, the bounds will have positive width and height; x0 and y0 will be the minimal values.

path: a Path instance

ignore:
  • when True, ignore the existing bounds of the Bbox.
  • when False, include the existing bounds of the Bbox.
  • when None, use the last value passed to ignore().

updatex: when True, update the x values
updatey: when True, update the y values

class matplotlib.transforms.TransformedBbox(bbox, transform, **kwargs)
Bases: matplotlib.transforms.BboxBase

A Bbox that is automatically transformed by a given transform. When either the child bounding box or transform changes, the bounds of this bbox will update accordingly.
bbox: a child Bbox
transform: a 2D Transform

def get_points()
    Get the points of the bounding box directly as a numpy array of the form: [[x0, y0], [x1, y1]].

class matplotlib.transforms.Transform(shorthand_name=None)
    Bases: matplotlib.transforms.TransformNode

    The base class of all TransformNode instances that actually perform a transformation.
    All non-affine transformations should be subclasses of this class. New affine transformations should be subclasses of Affine2D.
    Subclasses of this class should override the following members (at minimum):

        • input_dims
        • output_dims
        • transform()
        • is_separable
        • has_inverse
        • inverted() (if has_inverse is True)

    If the transform needs to do something non-standard with matplotlib.path.Path objects, such as adding curves where there were once line segments, it should override:

        • transform_path()

    Creates a new TransformNode.

    shorthand_name - a string representing the “name” of this transform. The name carries no significance other than to improve the readability of str(transform) when DEBUG=True.

    contains_branch(other)
    Return whether the given transform is a sub-tree of this transform.
    This routine uses transform equality to identify sub-trees, therefore in many situations it is object id which will be used.
    For the case where the given transform represents the whole of this transform, returns True.

    contains_branch_seperately(other_transform)
    Returns whether the given branch is a sub-tree of this transform on each separate dimension.
    A common use for this method is to identify if a transform is a blended transform containing an axes’ data transform. e.g.:

        x_isdata, y_isdata = trans.contains_branch_seperately(ax.transData)

    depth
    Returns the number of transforms which have been chained together to form this Transform instance.
Note: For the special case of a Composite transform, the maximum depth of the two is returned.

get_affine()
Get the affine part of this transform.

get_matrix()
Get the affine transformation array for the affine part of this transform.

has_inverse = False
True if this transform has a corresponding inverse transform.

input_dims = None
The number of input dimensions of this transform. Must be overridden (with integers) in the subclass.

inverted()
Return the corresponding inverse transformation.

The return value of this method should be treated as temporary. An update to self does not cause a corresponding update to its inverted copy.

x === self.inverted().transform(self.transform(x))

is_separable = False
True if this transform is separable in the x- and y- dimensions.

output_dims = None
The number of output dimensions of this transform. Must be overridden (with integers) in the subclass.

transform(values)
Performs the transformation on the given array of values.

Accepts a numpy array of shape (N x input_dims) and returns a numpy array of shape (N x output_dims).

Alternatively, accepts a numpy array of length input_dims and returns a numpy array of length output_dims.

transform_affine(values)
Performs only the affine part of this transformation on the given array of values.

transform(values) is always equivalent to transform_affine(transform_non_affine(values)).

In non-affine transformations, this is generally a no-op. In affine transformations, this is equivalent to transform(values).

Accepts a numpy array of shape (N x input_dims) and returns a numpy array of shape (N x output_dims).

Alternatively, accepts a numpy array of length input_dims and returns a numpy array of length output_dims.

transform_angles(angles, pts, radians=False, pushoff=1e-05)
Performs transformation on a set of angles anchored at specific locations.
The \textit{angles} must be a column vector (i.e., numpy array).

The \textit{pts} must be a two-column numpy array of x,y positions (angle transforms currently only work in 2D). This array must have the same number of rows as \textit{angles}.

\textit{radians} indicates whether or not input angles are given in radians (True) or degrees (False; the default).

\textit{pushoff} is the distance to move away from \textit{pts} for determining transformed angles (see discussion of method below).

The transformed angles are returned in an array with the same size as \textit{angles}.

The generic version of this method uses a very generic algorithm that transforms \textit{pts}, as well as locations very close to \textit{pts}, to find the angle in the transformed system.

\textbf{transform_bbox}(bbox)

Transform the given bounding box.

Note, for smarter transforms including caching (a common requirement for matplotlib figures), see \textit{TransformedBbox}.

\textbf{transform_non_affine}(values)

Performs only the non-affine part of the transformation.

\textit{transform(values)} is always equivalent to \textit{transform_affine(transform_non_affine(values))}.

In non-affine transformations, this is generally equivalent to \textit{transform(values)}. In affine transformations, this is always a no-op.

Accepts a numpy array of shape (N x \textit{input_dims}) and returns a numpy array of shape (N x \textit{output_dims}).

Alternatively, accepts a numpy array of length \textit{input_dims} and returns a numpy array of length \textit{output_dims}.

\textbf{transform_path}(path)

Returns a transformed path.

\textit{path}: a \textit{Path} instance.

In some cases, this transform may insert curves into the path that began as line segments.

\textbf{transform_path_affine}(path)

Returns a path, transformed only by the affine part of this transform.

\textit{path}: a \textit{Path} instance.

\textit{transform_path(path)} is equivalent to \textit{transform_path_affine(transform_path_non_affine(values))}.

\textbf{transform_path_non_affine}(path)

Returns a path, transformed only by the non-affine part of this transform.

\textit{path}: a \textit{Path} instance.

\textit{transform_path(path)} is equivalent to \textit{transform_path_affine(transform_path_non_affine(values))}. 

\textbf{54.1. matplotlib.transforms}
transform_point(point)

A convenience function that returns the transformed copy of a single point.

The point is given as a sequence of length `input_dims`. The transformed point is returned as a
sequence of length `output_dims`.

class matplotlib.transforms.TransformWrapper(child)

Bases: matplotlib.transforms.Transform

A helper class that holds a single child transform and acts equivalently to it.

This is useful if a node of the transform tree must be replaced at run time with a transform of a different
type. This class allows that replacement to correctly trigger invalidation.

Note that `TransformWrapper` instances must have the same input and output dimensions during their
entire lifetime, so the child transform may only be replaced with another child transform of the same
dimensions.

child: A class: `Transform` instance. This child may later be replaced with `set()`.

frozen()

Returns a frozen copy of this transform node. The frozen copy will not update when its children
change. Useful for storing a previously known state of a transform where `copy.deepcopy()`
might normally be used.

set(child)

Replace the current child of this transform with another one.

The new child must have the same number of input and output dimensions as the current child.

class matplotlib.transforms.AffineBase(*args, **kwargs)

Bases: matplotlib.transforms.Transform

The base class of all affine transformations of any number of dimensions.

get_affine()

Get the affine part of this transform.

transform(values)

Performs the transformation on the given array of values.

Accepts a numpy array of shape (N x `input_dims`) and returns a numpy array of shape (N x `output_dims`).

Alternatively, accepts a numpy array of length `input_dims` and returns a numpy array of length
`output_dims`.

transform_affine(values)

Performs only the affine part of this transformation on the given array of values.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally a no-op. In affine transformations, this is equiva-
 lent to `transform(values)`.

Accepts a numpy array of shape (N x `input_dims`) and returns a numpy array of shape (N x `output_dims`).
Alternatively, accepts a numpy array of length `input_dims` and returns a numpy array of length `output_dims`.

**transform_non_affine(points)**
Performs only the non-affine part of the transformation.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

Accepts a numpy array of shape `(N x input_dims)` and returns a numpy array of shape `(N x output_dims)`.
Alternatively, accepts a numpy array of length `input_dims` and returns a numpy array of length `output_dims`.

**transform_path(path)**
Returns a transformed path.

`path`: a `Path` instance.

In some cases, this transform may insert curves into the path that began as line segments.

**transform_path_affine(path)**
Returns a path, transformed only by the affine part of this transform.

`path`: a `Path` instance.

`transform_path(path)` is equivalent to `transform_path_affine(transform_path_non_affine(values))`.

**transform_path_non_affine(path)**
Returns a path, transformed only by the non-affine part of this transform.

`path`: a `Path` instance.

`transform_path(path)` is equivalent to `transform_path_affine(transform_path_non_affine(values))`.

---

**class** `matplotlib.transforms.Affine2DBase(*args, **kwargs)`

**Bases:** `matplotlib.transforms.AffineBase`

The base class of all 2D affine transformations.

2D affine transformations are performed using a 3x3 numpy array:

```
[ a  c  e ]
[ b  d  f ]
[ 0  0  1 ]
```

This class provides the read-only interface. For a mutable 2D affine transformation, use `Affine2D`.

Subclasses of this class will generally only need to override a constructor and `get_matrix()` that generates a custom 3x3 matrix.

**frozen()**
Returns a frozen copy of this transform node. The frozen copy will not update when its children change. Useful for storing a previously known state of a transform where `copy.deepcopy()` might normally be used.
inverted()
Return the corresponding inverse transformation.

The return value of this method should be treated as temporary. An update to self does not cause
a corresponding update to its inverted copy.

\[ x \implies self.inverted().transform(self.transform(x)) \]

static matrix_from_values(a, b, c, d, e, f)
    (staticmethod) Create a new transformation matrix as a 3x3 numpy array of the form:
    
    \[
    \begin{bmatrix}
    a & c & e \\
    b & d & f \\
    0 & 0 & 1
    \end{bmatrix}
    \]

to_values()
Return the values of the matrix as a sequence (a,b,c,d,e,f)

transform_affine(points)
Performs only the affine part of this transformation on the given array of values.

transform(values) is always equivalent to transform_affine(transform_non_affine(values)).

In non-affine transformations, this is generally a no-op. In affine transformations, this is equivalent to transform(values).

Accepts a numpy array of shape (N x input_dims) and returns a numpy array of shape (N x output_dims).

Alternatively, accepts a numpy array of length input_dims and returns a numpy array of length output_dims.

transform_point(point)
A convenience function that returns the transformed copy of a single point.

The point is given as a sequence of length input_dims. The transformed point is returned as a
sequence of length output_dims.

class matplotlib.transforms.Affine2D(matrix=None, **kwargs)
Bases: matplotlib.transforms.Affine2DBase

A mutable 2D affine transformation.

Initialize an Affine transform from a 3x3 numpy float array:

\[
\begin{bmatrix}
    a & c & e \\
    b & d & f \\
    0 & 0 & 1
\end{bmatrix}
\]

If matrix is None, initialize with the identity transform.

clear()
Reset the underlying matrix to the identity transform.

static from_values(a, b, c, d, e, f)
    (staticmethod) Create a new Affine2D instance from the given values:
get_matrix()

Get the underlying transformation matrix as a 3x3 numpy array:

\[
\begin{bmatrix}
  a & c & e \\
  b & d & f \\
  0 & 0 & 1
\end{bmatrix}
\]

static identity()

(staticmethod) Return a new Affine2D object that is the identity transform.

Unless this transform will be mutated later on, consider using the faster IdentityTransform class instead.

rotate(theta)

Add a rotation (in radians) to this transform in place.

Returns self, so this method can easily be chained with more calls to rotate(), rotate_deg(), translate() and scale().

rotate_around(x, y, theta)

Add a rotation (in radians) around the point (x, y) in place.

Returns self, so this method can easily be chained with more calls to rotate(), rotate_deg(), translate() and scale().

rotate_deg(degrees)

Add a rotation (in degrees) to this transform in place.

Returns self, so this method can easily be chained with more calls to rotate(), rotate_deg(), translate() and scale().

rotate_deg_around(x, y, degrees)

Add a rotation (in degrees) around the point (x, y) in place.

Returns self, so this method can easily be chained with more calls to rotate(), rotate_deg(), translate() and scale().

scale(sx, sy=None)

Adds a scale in place.

If sy is None, the same scale is applied in both the x- and y-directions.

Returns self, so this method can easily be chained with more calls to rotate(), rotate_deg(), translate() and scale().

set(other)

Set this transformation from the frozen copy of another Affine2DBase object.
set_matrix(mtx)
    Set the underlying transformation matrix from a 3x3 numpy array:

    \[
    \begin{bmatrix}
    a & c & e \\
    b & d & f \\
    0 & 0 & 1
    \end{bmatrix}
    \]

skew(xShear, yShear)
    Adds a skew in place.
    xShear and yShear are the shear angles along the x- and y-axes, respectively, in radians.
    Returns self, so this method can easily be chained with more calls to rotate(), rotate_deg(),
    translate() and scale().

skew_deg(xShear, yShear)
    Adds a skew in place.
    xShear and yShear are the shear angles along the x- and y-axes, respectively, in degrees.
    Returns self, so this method can easily be chained with more calls to rotate(), rotate_deg(),
    translate() and scale().

translate(tx, ty)
    Adds a translation in place.
    Returns self, so this method can easily be chained with more calls to rotate(), rotate_deg(),
    translate() and scale().

class matplotlib.transforms.IdentityTransform(*args, **kwargs)
    Bases: matplotlib.transforms.Affine2DBase
    A special class that does one thing, the identity transform, in a fast way.
    
    frozen()
    Returns a frozen copy of this transform node. The frozen copy will not update when its children
    change. Useful for storing a previously known state of a transform where copy.deepcopy() might
    normally be used.

get_affine()
    Return the corresponding inverse transformation.
    The return value of this method should be treated as temporary. An update to self does not cause
    a corresponding update to its inverted copy.
    x == self.inverted().transform(self.transform(x))

get_matrix()
    Get the Affine transformation array for the affine part of this transform.

inverted()
    Return the corresponding inverse transformation.
    The return value of this method should be treated as temporary. An update to self does not cause
    a corresponding update to its inverted copy.
x == self.inverted().transform(self.transform(x))

**transform(points)**
Performs only the non-affine part of the transformation.

**transform(values)** is always equivalent to **transform_affine(transform_non_affine(values)).**

In non-affine transformations, this is generally equivalent to **transform(values).** In affine transformations, this is always a no-op.

Accepts a numpy array of shape \((N \times \text{input\_dims})\) and returns a numpy array of shape \((N \times \text{output\_dims})\).

Alternatively, accepts a numpy array of length \(\text{input\_dims}\) and returns a numpy array of length \(\text{output\_dims}\).

**transform_affine(points)**
Performs only the non-affine part of the transformation.

**transform(values)** is always equivalent to **transform_affine(transform_non_affine(values)).**

In non-affine transformations, this is generally equivalent to **transform(values).** In affine transformations, this is always a no-op.

Accepts a numpy array of shape \((N \times \text{input\_dims})\) and returns a numpy array of shape \((N \times \text{output\_dims})\).

Alternatively, accepts a numpy array of length \(\text{input\_dims}\) and returns a numpy array of length \(\text{output\_dims}\).

**transform_non_affine(points)**
Performs only the non-affine part of the transformation.

**transform(values)** is always equivalent to **transform_affine(transform_non_affine(values)).**

In non-affine transformations, this is generally equivalent to **transform(values).** In affine transformations, this is always a no-op.

Accepts a numpy array of shape \((N \times \text{input\_dims})\) and returns a numpy array of shape \((N \times \text{output\_dims})\).

Alternatively, accepts a numpy array of length \(\text{input\_dims}\) and returns a numpy array of length \(\text{output\_dims}\).

**transform_path(path)**
Returns a path, transformed only by the non-affine part of this transform.

\(path: a \text{ Path} \) instance.

**transform_path(path)** is equivalent to **transform_path_affine(transform_path_non_affine(values)).**

**transform_path_affine(path)**
Returns a path, transformed only by the non-affine part of this transform.

\(path: a \text{ Path} \) instance.

**transform_path(path)** is equivalent to **transform_path_affine(transform_path_non_affine(values)).**
**transform_path_non_affine**(*path*)

Returns a path, transformed only by the non-affine part of this transform.

*path*: a *Path* instance.

transform_path(path) is equivalent to transform_path_affine(transform_path_non_affine(values)).

class matplotlib.transforms.BlendedGenericTransform(*x_transform, y_transform, **kwargs*)

Bases: matplotlib.transforms.Transform

A “blended” transform uses one transform for the *x*-direction, and another transform for the *y*-direction.

This “generic” version can handle any given child transform in the *x* - and *y*-directions.

Create a new “blended” transform using *x_transform* to transform the *x*-axis and *y_transform* to transform the *y*-axis.

You will generally not call this constructor directly but use the blended_transform_factory() function instead, which can determine automatically which kind of blended transform to create.

frozen()

Returns a frozen copy of this transform node. The frozen copy will not update when its children change. Useful for storing a previously known state of a transform where copy.deepcopy() might normally be used.

get_affine()

Get the affine part of this transform.

inverted()

Return the corresponding inverse transformation.

The return value of this method should be treated as temporary. An update to self does not cause a corresponding update to its inverted copy.

\[ x \equiv self.\text{inverted()}.\text{transform}(self.\text{transform}(x)) \]

**transform_non_affine**(*points*)

Performs only the non-affine part of the transformation.

transform(values) is always equivalent to transform_affine(transform_non_affine(values)).

In non-affine transformations, this is generally equivalent to transform(values). In affine transformations, this is always a no-op.

Accepts a numpy array of shape (N x input_dims) and returns a numpy array of shape (N x output_dims).

Alternatively, accepts a numpy array of length input_dims and returns a numpy array of length output_dims.

class matplotlib.transforms.BlendedAffine2D(*x_transform, y_transform, **kwargs*)

Bases: matplotlib.transforms.Affine2DBase

A “blended” transform uses one transform for the *x*-direction, and another transform for the *y*-direction.
This version is an optimization for the case where both child transforms are of type `Affine2DBase`. Create a new “blended” transform using `x_transform` to transform the x-axis and `y_transform` to transform the y-axis.

Both `x_transform` and `y_transform` must be 2D affine transforms.

You will generally not call this constructor directly but use the `blended_transform_factory()` function instead, which can determine automatically which kind of blended transform to create.

```python
get_matrix()
```

Get the Affine transformation array for the affine part of this transform.

```python
matplotlib.transforms.blended_transform_factory(x_transform, y_transform)
```

Create a new “blended” transform using `x_transform` to transform the x-axis and `y_transform` to transform the y-axis.

A faster version of the blended transform is returned for the case where both child transforms are affine.

```python
class matplotlib.transforms.CompositeGenericTransform(a, b, **kwargs)
```

A composite transform formed by applying transform `a` then transform `b`.

This “generic” version can handle any two arbitrary transformations.

Create a new composite transform that is the result of applying transform `a` then transform `b`.

You will generally not call this constructor directly but use the `composite_transform_factory()` function instead, which can automatically choose the best kind of composite transform instance to create.

```python
frozen()
```

Returns a frozen copy of this transform node. The frozen copy will not update when its children change. Useful for storing a previously known state of a transform where `copy.deepcopy()` might normally be used.

```python
get_affine()
```

Get the affine part of this transform.

```python
inverted()
```

Return the corresponding inverse transformation.

The return value of this method should be treated as temporary. An update to `self` does not cause a corresponding update to its inverted copy.

```python
x === self.inverted().transform(self.transform(x))
```

```python
transform_affine(points)
```

Performs only the affine part of this transformation on the given array of values.

```python
transform(values) is always equivalent to transform_affine(transform_non_affine(values)).
```

In non-affine transformations, this is generally a no-op. In affine transformations, this is equivalent to `transform(values)`.
Accepts a numpy array of shape (N x input_dims) and returns a numpy array of shape (N x output_dims).

Alternatively, accepts a numpy array of length input_dims and returns a numpy array of length output_dims.

**transform_non_affine(points)**

Performs only the non-affine part of the transformation.

transform(values) is always equivalent to transform_affine(transform_non_affine(values)).

In non-affine transformations, this is generally equivalent to transform(values). In affine transformations, this is always a no-op.

Accepts a numpy array of shape (N x input_dims) and returns a numpy array of shape (N x output_dims).

Alternatively, accepts a numpy array of length input_dims and returns a numpy array of length output_dims.

**transform_path_non_affine(path)**

Returns a path, transformed only by the non-affine part of this transform.

path: a Path instance.

transform_path(path) is equivalent to transform_path_affine(transform_path_non_affine(values)).

---

**class matplotlib.transforms.CompositeAffine2D(a, b, **kwargs)**

Bases: matplotlib.transforms.Affine2DBase

A composite transform formed by applying transform a then transform b.

This version is an optimization that handles the case where both a and b are 2D aффines.

Create a new composite transform that is the result of applying transform a then transform b.

Both a and b must be instances of Affine2DBase.

You will generally not call this constructor directly but use the composite_transform_factory() function instead, which can automatically choose the best kind of composite transform instance to create.

**get_matrix()**

Get the Aффine transformation array for the affine part of this transform.

**class matplotlib.transforms.BboxTransform(boxin, boxout, **kwargs)**

Bases: matplotlib.transforms.Affine2DBase

Create a new composite transform that is the result of applying transform a then transform b.

Shortcut versions of the blended transform are provided for the case where both child transforms are affine, or one or the other is the identity transform.

Composite transforms may also be created using the ‘+’ operator, e.g.:

```
c = a + b
```
BboxTransform linearly transforms points from one Bbox to another Bbox.

Create a new BboxTransform that linearly transforms points from boxin to boxout.

get_matrix()
Get the Affine transformation array for the affine part of this transform.

class matplotlib.transforms.BboxTransformTo(boxout, **kwargs)
Bases: matplotlib.transforms.Affine2DBase

BboxTransformTo is a transformation that linearly transforms points from the unit bounding box to a given Bbox.

Create a new BboxTransformTo that linearly transforms points from the unit bounding box to boxout.

class matplotlib.transforms.BboxTransformFrom(boxin, **kwargs)
Bases: matplotlib.transforms.Affine2DBase

BboxTransformFrom linearly transforms points from a given Bbox to the unit bounding box.

class matplotlib.transforms.ScaledTranslation(xt, yt, scale_trans, **kwargs)
Bases: matplotlib.transforms.Affine2DBase

A transformation that translates by xt and yt, after xt and yt have been transformad by the given transform scale_trans.

class matplotlib.transforms.TransformedPath(path, transform)
Bases: matplotlib.transforms.TransformNode

A TransformedPath caches a non-affine transformed copy of the Path. This cached copy is automatically updated when the non-affine part of the transform changes.

Note: Paths are considered immutable by this class. Any update to the path’s vertices/codes will not trigger a transform recomputation.

Create a new TransformedPath from the given Path and Transform.

class get_fully_transformed_path()
Return a fully-transformed copy of the child path.

class get_transformed_path_and_affine()
Return a copy of the child path, with the non-affine part of the transform already applied, along with the affine part of the path necessary to complete the transformation.
**get_transformed_points_and_affine()**

Return a copy of the child path, with the non-affine part of the transform already applied, along with the affine part of the path necessary to complete the transformation. Unlike *get_transformed_path_and_affine()*!, no interpolation will be performed.

*matplotlib.transforms.nonsingular(vmin, vmax, expander=0.001, tiny=1e-15, increasing=True)*

Modify the endpoints of a range as needed to avoid singularities.

*vmin, vmax* the initial endpoints.

*tiny* threshold for the ratio of the interval to the maximum absolute value of its endpoints. If the interval is smaller than this, it will be expanded. This value should be around 1e-15 or larger; otherwise the interval will be approaching the double precision resolution limit.

*expander* fractional amount by which *vmin* and *vmax* are expanded if the original interval is too small, based on *tiny*.

*increasing*: [True | False] If True (default), swap *vmin, vmax* if *vmin > vmax*

Returns *vmin, vmax*, expanded and/or swapped if necessary.

If either input is inf or NaN, or if both inputs are 0 or very close to zero, it returns -expander, expander.
55.1 matplotlib.tri

Unstructured triangular grid functions.

```python
class matplotlib.tri.Triangulation(x, y, triangles=None, mask=None)
```

An unstructured triangular grid consisting of npoints points and ntri triangles. The triangles can either be specified by the user or automatically generated using a Delaunay triangulation.

**Parameters**

- `x, y` : array_like of shape (npoints)
  
  Coordinates of grid points.

- `triangles` : integer array_like of shape (ntri, 3), optional
  
  For each triangle, the indices of the three points that make up the triangle, ordered in an anticlockwise manner. If not specified, the Delaunay triangulation is calculated.

- `mask` : boolean array_like of shape (ntri), optional
  
  Which triangles are masked out.

**Notes**

For a Triangulation to be valid it must not have duplicate points, triangles formed from colinear points, or overlapping triangles.

**Attributes**

- `edges`
- `neighbors`

- `is_delaunay` (bool) Whether the Triangulation is a calculated Delaunay triangulation (where triangles was not specified) or not.

- `calculate_plane_coefficients(z)`
  
  Calculate plane equation coefficients for all unmasked triangles from the point (x,y) coordinates and specified z-array of shape (npoints). Returned array has shape (npoints,3) and allows
z-value at (x,y) position in triangle tri to be calculated using \( z = \text{array}[\text{tri},0]x + \text{array}[\text{tri},1]y + \text{array}[\text{tri},2] \).

**edges**

Return integer array of shape (nedges,2) containing all edges of non-masked triangles.

Each edge is the start point index and end point index. Each edge (start,end and end,start) appears only once.

**static get_from_args_and_kwargs(**

Return a Triangulation object from the args and kwargs, and the remaining args and kwargs with the consumed values removed.

There are two alternatives: either the first argument is a Triangulation object, in which case it is returned, or the args and kwargs are sufficient to create a new Triangulation to return. In the latter case, see Triangulation.__init__ for the possible args and kwargs.

**get_masked_triangles()**

Return an array of triangles that are not masked.

**get_trifinder()**

Return the default matplotlib.tri.TriFinder of this triangulation, creating it if necessary. This allows the same TriFinder object to be easily shared.

**neighbors**

Return integer array of shape (ntri,3) containing neighbor triangles.

For each triangle, the indices of the three triangles that share the same edges, or -1 if there is no such neighboring triangle. neighbors[i,j] is the triangle that is the neighbor to the edge from point index triangles[i,j] to point index triangles[i,(j+1)%3].

**set_mask(mask)**

Set or clear the mask array. This is either None, or a boolean array of shape (ntri).

**class matplotlib.tri.TriFinder(triangulation)**

Abstract base class for classes used to find the triangles of a Triangulation in which (x,y) points lie.

Rather than instantiate an object of a class derived from TriFinder, it is usually better to use the function matplotlib.tri.Triangulation.get_trifinder().

Derived classes implement __call__(x,y) where x,y are array_like point coordinates of the same shape.

**class matplotlib.tri.TrapezoidMapTriFinder(triangulation)**

Bases: matplotlib.tri.trifinder.TriFinder


The triangulation must be valid, i.e. it must not have duplicate points, triangles formed from colinear points, or overlapping triangles. The algorithm has some tolerance to triangles formed from colinear points, but this should not be relied upon.

__call__(x, y)

Return an array containing the indices of the triangles in which the specified x,y points lie, or -1 for points that do not lie within a triangle.
$x$, $y$ are array_like x and y coordinates of the same shape and any number of dimensions. Returns integer array with the same shape and $x$ and $y$.

class matplotlib.tri.TriInterpolator(triangulation, z, trifinder=None)

Abstract base class for classes used to perform interpolation on triangular grids.

Derived classes implement the following methods:

• __call__(x, y), where $x$, $y$ are array_like point coordinates of the same shape, and that returns a masked array of the same shape containing the interpolated $z$-values.

• gradient(x, y), where $x$, $y$ are array_like point coordinates of the same shape, and that returns a list of 2 masked arrays of the same shape containing the 2 derivatives of the interpolator (derivatives of interpolated $z$ values with respect to $x$ and $y$).

class matplotlib.tri.LinearTriInterpolator(triangulation, z, trifinder=None)

Bases: matplotlib.tri.triinterpolate.TriInterpolator

A LinearTriInterpolator performs linear interpolation on a triangular grid.

Each triangle is represented by a plane so that an interpolated value at point $(x,y)$ lies on the plane of the triangle containing $(x,y)$. Interpolated values are therefore continuous across the triangulation, but their first derivatives are discontinuous at edges between triangles.

Parameters triangulation : Triangulation object

The triangulation to interpolate over.

$z$ : array_like of shape (npoints,)

Array of values, defined at grid points, to interpolate between.

trifinder : TriFinder object, optional

If this is not specified, the Triangulation’s default TriFinder will be used by calling matplotlib.tri.Triangulation.get_trifinder().

Methods

__call__(x, y)  (Returns interpolated values at x,y points)

gradient(x, y)  (Returns interpolated derivatives at x,y points)

__call__(x, y)

Returns a masked array containing interpolated values at the specified x,y points.

Parameters $x$, $y$ : array-like

$x$ and $y$ coordinates of the same shape and any number of dimensions.

Returns $z$ : np.ma.array

Masked array of the same shape as $x$ and $y$ : values corresponding to $(x, y)$ points outside of the triangulation are masked out.

gradient(x, y)

Returns a list of 2 masked arrays containing interpolated derivatives at the specified x,y points.
Parameters $x, y$ : array-like

$x$ and $y$ coordinates of the same shape and any number of dimensions.

Returns $dzdx, dzdy$ : np.ma.array

2 masked arrays of the same shape as $x$ and $y$; values corresponding to $(x,y)$ points outside of the triangulation are masked out. The first returned array contains the values of $\frac{\partial z}{\partial x}$ and the second those of $\frac{\partial z}{\partial y}$.

class matplotlib.tri.CubicTriInterpolator

Bases: matplotlib.tri.triinterpolate.TriInterpolator

A CubicTriInterpolator performs cubic interpolation on triangular grids.

In one-dimension - on a segment - a cubic interpolating function is defined by the values of the function and its derivative at both ends. This is almost the same in 2-d inside a triangle, except that the values of the function and its 2 derivatives have to be defined at each triangle node.

The CubicTriInterpolator takes the value of the function at each node - provided by the user - and internally computes the value of the derivatives, resulting in a smooth interpolation. (As a special feature, the user can also impose the value of the derivatives at each node, but this is not supposed to be the common usage.)

Parameters triangulation : Triangulation object

The triangulation to interpolate over.

$z$ : array_like of shape (npoints,)

Array of values, defined at grid points, to interpolate between.

kind : {'min_E', 'geom', 'user'}, optional

Choice of the smoothing algorithm, in order to compute the interpolant derivatives (defaults to ‘min_E’):

- if ‘min_E’: (default) The derivatives at each node is computed to minimize a bending energy.
- if ‘geom’: The derivatives at each node is computed as a weighted average of relevant triangle normals. To be used for speed optimization (large grids).
- if ‘user’: The user provides the argument $dz$, no computation is hence needed.

trifinder : TriFinder object, optional

If not specified, the Triangulation’s default TriFinder will be used by calling matplotlib.tri.Triangulation.get_trifinder().

dz : tuple of array_likes (dzdx, dzdy), optional

Used only if kind = ‘user’. In this case $dz$ must be provided as (dzdx, dzdy) where dzdx, dzdy are arrays of the same shape as $z$ and are the interpolant first derivatives at the triangulation points.
Notes

This note is a bit technical and details the way a `CubicTriInterpolator` computes a cubic interpolation.

The interpolation is based on a Clough-Tocher subdivision scheme of the triangulation mesh (to make it clearer, each triangle of the grid will be divided in 3 child-triangles, and on each child triangle the interpolated function is a cubic polynomial of the 2 coordinates). This technique originates from FEM (Finite Element Method) analysis; the element used is a reduced Hsieh-Clough-Tocher (HCT) element. Its shape functions are described in [R1]. The assembled function is guaranteed to be C1-smooth, i.e. it is continuous and its first derivatives are also continuous (this is easy to show inside the triangles but is also true when crossing the edges).

In the default case (`kind = 'min_E'`), the interpolant minimizes a curvature energy on the functional space generated by the HCT element shape functions - with imposed values but arbitrary derivatives at each node. The minimized functional is the integral of the so-called total curvature (implementation based on an algorithm from [R2] - PCG sparse solver):

\[
E(z) = \frac{1}{2} \int_{\Omega} \left( \frac{\partial^2 z}{\partial x^2} \right)^2 + \left( \frac{\partial^2 z}{\partial y^2} \right)^2 + 2 \left( \frac{\partial^2 z}{\partial y \partial x} \right)^2 \right) dx dy
\]  

(55.1)

If the case `kind = 'geom'` is chosen by the user, a simple geometric approximation is used (weighted average of the triangle normal vectors), which could improve speed on very large grids.

References

[R1], [R2]

Methods

```python
__call__(x, y)  # (Returns interpolated values at x,y points)
gradient(x, y)  # (Returns interpolated derivatives at x,y points)
```

__call__(x, y)

Returns a masked array containing interpolated values at the specified x,y points.

Parameters

- x, y : array-like
  - x and y coordinates of the same shape and any number of dimensions.

Returns

- z : np.ma.array
  - Masked array of the same shape as x and y; values corresponding to (x, y) points outside of the triangulation are masked out.

gradient(x, y)

Returns a list of 2 masked arrays containing interpolated derivatives at the specified x,y points.

Parameters

- x, y : array-like
x and y coordinates of the same shape and any number of dimensions.

**Returns** dzdx, dzdy : np.ma.array

2 masked arrays of the same shape as x and y ; values corresponding to (x,y) points outside of the triangulation are masked out. The first returned array contains the values of $\frac{\partial z}{\partial x}$ and the second those of $\frac{\partial z}{\partial y}$.

**Examples**

An example of effective application is shown below (plot of the direction of the vector field derivated from a known potential field):

![Gradient plot: an electrical dipole](image)

**class** matplotlib.tri.TriRefiner(triangulation)

Abstract base class for classes implementing mesh refinement.

A TriRefiner encapsulates a Triangulation object and provides tools for mesh refinement and interpolation.

Derived classes must implements:

- `refine_triangulation(return_tri_index=False, **kwargs)` , where the optional keyword arguments `kwargs` are defined in each TriRefiner concrete implementation, and which returns :
– a refined triangulation
– optionally (depending on return_tri_index), for each point of the refined triangulation: the
index of the initial triangulation triangle to which it belongs.

• refine_field(z, triinterpolator=None, **kwargs), where:
  – z array of field values (to refine) defined at the base triangulation nodes
  – triinterpolator is a TriInterpolator (optional)
  – the other optional keyword arguments kwargs are defined in each TriRefiner concrete im-
    plementation

and which returns (as a tuple) a refined triangular mesh and the interpolated values of the field
at the refined triangulation nodes.

class matplotlib.tri.UniformTriRefiner(triangulation)
Bases: matplotlib.tri.trirefine.TriRefiner
Uniform mesh refinement by recursive subdivisions.

Parameters triangulation: Triangulation

The encapsulated triangulation (to be refined)

refine_field(z, triinterpolator=None, subdiv=3)
Refines a field defined on the encapsulated triangulation.

Returns refi_tri (refined triangulation), refi_z (interpolated values of the field at the node of the
refined triangulation).

Parameters z : 1d-array-like of length n_points

Values of the field to refine, defined at the nodes of the encapsulated triangu-
lation. (n_points is the number of points in the initial triangulation)

triinterpolator: TriInterpolator, optional

Interpolator used for field interpolation. If not specified, a CubicTriInterpolator will be used.

subdiv: integer, optional

Recursion level for the subdivision. Defaults to 3. Each triangle will be
divided into 4**subdiv child triangles.

Returns refi_tri: Triangulation object

The returned refined triangulation

refi_z : 1d array of length: refi_tri node count.

The returned interpolated field (at refi_tri nodes)
Examples

The main application of this method is to plot high-quality iso-contours on a coarse triangular grid (e.g., triangulation built from relatively sparse test data):

refine_triangulation\texttt{(return\_tri\_index=False, subdiv=3)}

Computes an uniformly refined triangulation $\texttt{refi\_triangulation}$ of the encapsulated triangulation.

This function refines the encapsulated triangulation by splitting each father triangle into 4 child sub-triangles built on the edges midside nodes, recursively (level of recursion $\texttt{subdiv}$). In the end, each triangle is hence divided into $4^{\texttt{subdiv}}$ child triangles. The default value for $\texttt{subdiv}$ is 3 resulting in 64 refined subtriangles for each triangle of the initial triangulation.

**Parameters**

\texttt{return\_tri\_index} : boolean, optional

Boolean indicating whether an index table indicating the father triangle index of each point will be returned. Default value False.

\texttt{subdiv} : integer, optional

Recursion level for the subdivision. Defaults value 3. Each triangle will be divided into $4^{\texttt{subdiv}}$ child triangles.

**Returns**

\texttt{refi\_triangulation} : $\texttt{Triangulation}$

The returned refined triangulation
found_index : array-like of integers

Index of the initial triangulation containing triangle, for each point of refi_triangulation. Returned only if return_tri_index is set to True.

class matplotlib.tri.TriAnalyzer(triangulation)

Define basic tools for triangular mesh analysis and improvement.

A TriAnalyizer encapsulates a Triangulation object and provides basic tools for mesh analysis and mesh improvement.

Parameters triangulation : Triangulation object

The encapsulated triangulation to analyze.

Attributes

scale_factors

circle_ratios(rescale=True)

Returns a measure of the triangulation triangles flatness.

The ratio of the incircle radius over the circumcircle radius is a widely used indicator of a triangle flatness. It is always <= 0.5 and == 0.5 only for equilateral triangles. Circle ratios below 0.01 denote very flat triangles.

To avoid unduly low values due to a difference of scale between the 2 axis, the triangular mesh can first be rescaled to fit inside a unit square with scale_factors (Only if rescale is True, which is its default value).

Parameters rescale : boolean, optional

If True, a rescaling will be internally performed (based on scale_factors, so that the (unmasked) triangles fit exactly inside a unit square mesh. Default is True.

Returns circle_ratios : masked array

Ratio of the incircle radius over the circumcircle radius, for each ‘rescaled’ triangle of the encapsulated triangulation. Values corresponding to masked triangles are masked out.

get_flat_tri_mask(min_circle_ratio=0.01, rescale=True)

Eliminates excessively flat border triangles from the triangulation.

Returns a mask new_mask which allows to clean the encapsulated triangulation from its border-located flat triangles (according to their circle_ratios()). This mask is meant to be subsequently applied to the triangulation using matplotlib.tri.Triangulation.set_mask(). new_mask is an extension of the initial triangulation mask in the sense that an initially masked triangle will remain masked.

The new_mask array is computed recursively ; at each step flat triangles are removed only if they share a side with the current mesh border. Thus no new holes in the triangulated domain will be created.
**Parameters** `min_circle_ratio` : float, optional

Border triangles with incircle/circumcircle radii ratio \( r/R \) will be removed if \( r/R < min\_circle\_ratio \). Default value: 0.01

`rescale` : boolean, optional

If True, a rescaling will first be internally performed (based on `scale_factors`), so that the (unmasked) triangles fit exactly inside a unit square mesh. This rescaling accounts for the difference of scale which might exist between the 2 axis. Default (and recommended) value is True.

**Returns** `new_mask` : array-like of booleans

Mask to apply to encapsulated triangulation. All the initially masked triangles remain masked in the `new_mask`.

**Notes**

The rationale behind this function is that a Delaunay triangulation - of an unstructured set of points - sometimes contains almost flat triangles at its border, leading to artifacts in plots (especially for high-resolution contouring). Masked with computed `new_mask`, the encapsulated triangulation would contain no more unmasked border triangles with a circle ratio below `min_circle_ratio`, thus improving the mesh quality for subsequent plots or interpolation.

**Examples**

Please refer to the following illustrating example:
**scale_factors**

Factors to rescale the triangulation into a unit square.

Returns $k$, tuple of 2 scale factors.

**Returns** $k$ : tuple of 2 floats (kx, ky)

Tuple of floats that would rescale the triangulation: $[\text{triangulation.x} \times kx, \text{triangulation.y} \times ky]$ fits exactly inside a unit square.
56.1 matplotlib.type1font

This module contains a class representing a Type 1 font.

This version reads pfa and pfb files and splits them for embedding in pdf files. It also supports SlantFont and ExtendFont transformations, similarly to pdfTeX and friends. There is no support yet for subsetting.

Usage:

```python
>>> font = Type1Font(filename)
>>> clear_part, encrypted_part, finale = font.parts
>>> slanted_font = font.transform({'slant': 0.167})
>>> extended_font = font.transform({'extend': 1.2})
```

Sources:

- Adobe Technical Note #5040, Supporting Downloadable PostScript Language Fonts.

```python
class matplotlib.type1font.Type1Font(input)
Bases: object

A class representing a Type-1 font, for use by backends.

parts
    A 3-tuple of the cleartext part, the encrypted part, and the finale of zeros.

prop
    A dictionary of font properties.
```

Initialize a Type-1 font. input can be either the file name of a pfb file or a 3-tuple of already-decoded Type-1 font parts.
transform(effects)

Transform the font by slanting or extending. effects should be a dict where effects['slant'] is the tangent of the angle that the font is to be slanted to the right (so negative values slant to the left) and effects['extend'] is the multiplier by which the font is to be extended (so values less than 1.0 condense). Returns a new Type1Font object.

matplotlib.type1font.ord(x)
57.1 matplotlib.units

The classes here provide support for using custom classes with matplotlib, e.g., those that do not expose the array interface but know how to converter themselves to arrays. It also supports classes with units and units conversion. Use cases include converters for custom objects, e.g., a list of datetime objects, as well as for objects that are unit aware. We don’t assume any particular units implementation, rather a units implementation must provide a ConversionInterface, and the register with the Registry converter dictionary. For example, here is a complete implementation which supports plotting with native datetime objects:

```python
import matplotlib.units as units
import matplotlib.dates as dates
import matplotlib.ticker as ticker
import datetime

class DateConverter(units.ConversionInterface):
    @staticmethod
    def convert(value, unit, axis):
        'convert value to a scalar or array'
        return dates.date2num(value)

    @staticmethod
    def axisinfo(unit, axis):
        'return major and minor tick locators and formatters'
        if unit != 'date': return None
        majloc = dates.AutoDateLocator()
        majfmt = dates.AutoDateFormatter(majloc)
        return AxisInfo(majloc=majloc,
                        majfmt=majfmt,
                        label='date')

    @staticmethod
    def default_units(x, axis):
        'return the default unit for x or None'
        return 'date'

# finally we register our object type with a converter
units.registry[datetime.date] = DateConverter()
```
class matplotlib.units.AxisInfo(majloc=None, minloc=None, majfmt=None, minfmt=None, 
label=None, default_limits=None)

Bases: object

information to support default axis labeling and tick labeling, and default limits

majloc and minloc: TickLocators for the major and minor ticks 
majfmt and minfmt: TickFormatters for the major and minor ticks 
label: the default axis label 
default_limits: the default min, max of the axis if no data is present 
If any of the above are None, the axis will simply use the default

class matplotlib.units.ConversionInterface

Bases: object

The minimal interface for a converter to take custom instances (or sequences) and convert them to values mpl can use

static axisinfo(unit, axis)
return an units.AxisInfo instance for axis with the specified units

static convert(obj, unit, axis)
convert obj using unit for the specified axis. If obj is a sequence, return the converted sequence. The output must be a sequence of scalars that can be used by the numpy array layer

static default_units(x, axis)
return the default unit for x or None for the given axis

static is_numlike(x)
The matplotlib datalim, autoscaling, locators etc work with scalars which are the units converted to floats given the current unit. The converter may be passed these floats, or arrays of them, even when units are set. Derived conversion interfaces may opt to pass plain-ol unitless numbers through the conversion interface and this is a helper function for them.

class matplotlib.units.Registry
Bases: dict

register types with conversion interface

get_converter(x)
get the converter interface instance for x, or None
58.1 matplotlib.widgets

58.1.1 GUI neutral widgets

Widgets that are designed to work for any of the GUI backends. All of these widgets require you to predefine a matplotlib.axes.Axes instance and pass that as the first arg. matplotlib doesn’t try to be too smart with respect to layout – you will have to figure out how wide and tall you want your Axes to be to accommodate your widget.

```python
class matplotlib.widgets.AxesWidget(ax)
    Bases: matplotlib.widgets.Widget

    Widget that is connected to a single Axes.

    To guarantee that the widget remains responsive and not garbage-collected, a reference to the object should be maintained by the user.

    This is necessary because the callback registry maintains only weak-refs to the functions, which are member functions of the widget. If there are no references to the widget object it may be garbage collected which will disconnect the callbacks.

    Attributes:

    ax [Axes] The parent axes for the widget

    canvas [FigureCanvasBase subclass] The parent figure canvas for the widget.

    active [bool] If False, the widget does not respond to events.

    connect_event(event, callback)
        Connect callback with an event.

        This should be used in lieu of figure.canvas.mpl_connect since this function stores callback ids for later clean up.

    disconnect_events()
        Disconnect all events created by this widget.

class matplotlib.widgets.Button(ax, label, image=None, color='0.85', hovercolor='0.95')
    Bases: matplotlib.widgets.AxesWidget
```

1687
A GUI neutral button.
For the button to remain responsive you must keep a reference to it.

The following attributes are accessible

- **ax**: The `matplotlib.axes.Axes` the button renders into.
- **label**: A `matplotlib.text.Text` instance.
- **color**: The color of the button when not hovering.
- **hovercolor**: The color of the button when hovering.

Call `on_clicked()` to connect to the button

**Parameters**

- `ax` : `matplotlib.axes.Axes`
The `matplotlib.axes.Axes` instance the button will be placed into.

- `label` : str
  The button text. Accepts string.

- `image` : array, mpl image, Pillow Image
  The image to place in the button, if not `None`. Can be any legal arg to `imshow` (numpy array, matplotlib Image instance, or Pillow Image).

- `color` : color
  The color of the button when not activated

- `hovercolor` : color
  The color of the button when the mouse is over it

**disconnect**

- `disconnect(cid)`
  remove the observer with connection id `cid`

**on_clicked**

- `on_clicked(func)`
  When the button is clicked, call this `func` with event.
  A connection id is returned. It can be used to disconnect the button from its callback.

**class** `matplotlib.widgets.CheckButtons`( `ax`, `labels`, `actives` )

**Bases**: `matplotlib.widgets.AxesWidget`

A GUI neutral radio button.
For the check buttons to remain responsive you must keep a reference to this object.

The following attributes are exposed

- **ax** : The `matplotlib.axes.Axes` instance the buttons are located in

- **labels** : List of `matplotlib.text.Text` instances

- **lines** : List of (line1, line2) tuples for the x’s in the check boxes. These lines exist for each box, but have `set_visible(False)` when its box is not checked.

- **rectangles** : List of `matplotlib.patches.Rectangle` instances
Connect to the CheckButtons with the on_clicked() method

Add check buttons to matplotlib.axes.Axes instance ax

labels A len(buttons) list of labels as strings

actives

A len(buttons) list of booleans indicating whether the button is active

disconnect(cid)

remove the observer with connection id cid

on_clicked(func)

When the button is clicked, call func with button label

A connection id is returned which can be used to disconnect

set_active(index)

Directly (de)activate a check button by index.

index is an index into the original label list that this object was constructed with. Raises ValueError if index is invalid.

Callbacks will be triggered if eventson is True.

class matplotlib.widgets.Cursor(ax, horizOn=True, vertOn=True, useblit=False, **lineprops)

Bases: matplotlib.widgets.AxesWidget

A horizontal and vertical line that spans the axes and moves with the pointer. You can turn off the hline or vline respectively with the following attributes:

horizOn Controls the visibility of the horizontal line

vertOn Controls the visibility of the horizontal line

and the visibility of the cursor itself with the visible attribute.

For the cursor to remain responsive you must keep a reference to it.

Add a cursor to ax. If useblit=True, use the backend-dependent blitting features for faster updates (GTKAgg only for now). lineprops is a dictionary of line properties.
clear(event)
clear the cursor

onmove(event)
on mouse motion draw the cursor if visible

class matplotlib.widgets.EllipseSelector(ax, onselect, drawtype='box', minspanx=None, minspany=None, useblit=False, lineprops=None, rectprops=None, spancoords='data', button=None, maxdist=10, marker_props=None, interactive=False, state_modifier_keys=None)

Bases: matplotlib.widgets.RectangleSelector

Select an elliptical region of an axes.

For the cursor to remain responsive you must keep a reference to it.

Example usage:

```python
from matplotlib.widgets import EllipseSelector
from pylab import *

def onselect(eclick, erelease):
    'eclick and erelease are matplotlib events at press and release'
    print('startposition : (%f, %f)' % (eclick.xdata, eclick.ydata))
```
```python
print(' endposition : (%f, %f)' % (erelease.xdata, erelease.ydata))
print(' used button : ', eclick.button)

def toggle_selector(event):
    print(' Key pressed. ')
    if event.key in ['Q', 'q'] and toggle_selector.ES.active:
        print(' EllipseSelector deactivated. ')
        toggle_selector.RS.set_active(False)
    if event.key in ['A', 'a'] and not toggle_selector.ES.active:
        print(' EllipseSelector activated. ')
        toggle_selector.ES.set_active(True)

x = arange(100)/(99.0)
y = sin(x)
fig = figure
ax = subplot(111)
ax.plot(x,y)
toggle_selector.ES = EllipseSelector(ax, onselect, drawtype='line')
connect('key_press_event', toggle_selector)
show()
```

Create a selector in `ax`. When a selection is made, clear the span and call onselect with:

```python
onselect(pos_1, pos_2)
```

and clear the drawn box/line. The `pos_1` and `pos_2` are arrays of length 2 containing the x- and y-coordinate.

If `minspanx` is not `None` then events smaller than `minspanx` in x direction are ignored (it’s the same for y).

The rectangle is drawn with `rectprops`; default:

```python
rectprops = dict(facecolor='red', edgecolor = 'black',
                 alpha=0.2, fill=True)
```

The line is drawn with `lineprops`; default:

```python
lineprops = dict(color='black', linestyle='-',
                  linewidth = 2, alpha=0.5)
```

Use `drawtype` if you want the mouse to draw a line, a box or nothing between click and actual position by setting

```python
drawtype = 'line', drawtype='box' or drawtype = 'none'.
```

`spancoords` is one of 'data' or 'pixels'. If 'data', `minspanx` and `minspanx` will be interpreted in the same coordinates as the x and y axis. If 'pixels', they are in pixels.

`button` is a list of integers indicating which mouse buttons should be used for rectangle selection. You can also specify a single integer if only a single button is desired. Default is `None`, which does not limit which button can be used.

58.1. matplotlib.widgets
Note, typically: 1 = left mouse button 2 = center mouse button (scroll wheel) 3 = right mouse button

Interactive will draw a set of handles and allow you interact with the widget after it is drawn.

State modifier keys are keyboard modifiers that affect the behavior of the widget.

The defaults are: {'move': ' ', 'clear': 'escape', 'square': 'shift', 'center': 'ctrl'}

Keyboard modifiers, which: ‘move’: Move the existing shape. ‘clear’: Clear the current shape.
‘square’: Makes the shape square. ‘center’: Make the initial point the center of the shape. ‘square’
and ‘center’ can be combined.

draw_shape(extents)

class matplotlib.widgets.Lasso(ax, xy, callback=None, useblit=True)

Selection curve of an arbitrary shape.

The selected path can be used in conjunction with contains_point() to select data points from an
image.

Unlike LassoSelector, this must be initialized with a starting point xy, and the Lasso events are
destroyed upon release.

Parameters:

ax [Axes] The parent axes for the widget.
	xy [array] Coordinates of the start of the lasso.

callback [function] Whenever the lasso is released, the callback function is called and passed the
vertices of the selected path.

onmove(event)

onrelease(event)

class matplotlib.widgets.LassoSelector(ax, onselect=None, useblit=True, lineprops=None,
button=None)

Selection curve of an arbitrary shape.

For the selector to remain responsive you must keep a reference to it.

The selected path can be used in conjunction with contains_point() to select data points from an
image.

In contrast to Lasso, LassoSelector is written with an interface similar to RectangleSelector
and SpanSelector and will continue to interact with the axes until disconnected.

Parameters:

ax [Axes] The parent axes for the widget.
onselect [function] Whenever the lasso is released, the onselect function is called and passed the vertices of the selected path.

Example usage:

```python
ax = subplot(111)
ax.plot(x, y)

def onselect(verts):
    print(verts)
lasso = LassoSelector(ax, onselect)
```

*button* is a list of integers indicating which mouse buttons should be used for rectangle selection. You can also specify a single integer if only a single button is desired. Default is "None", which does not limit which button can be used.

Note, typically:
1 = left mouse button
2 = center mouse button (scroll wheel)
3 = right mouse button

onpress(event)

onrelease(event)

class matplotlib.widgets.LockDraw
Bases: object

Some widgets, like the cursor, draw onto the canvas, and this is not desirable under all circumstances, like when the toolbar is in zoom-to-rect mode and drawing a rectangle. The module level “lock” allows someone to grab the lock and prevent other widgets from drawing. Use `matplotlib.widgets.lock(someobj)` to prevent other widgets from drawing while you’re interacting with the canvas.

available(o)

drawing is available to o

isowner(o)

Return True if o owns this lock

locked()

Return True if the lock is currently held by an owner

release(o)

release the lock

class matplotlib.widgets.MultiCursor(canvas, axes, useblit=True, horizOn=False, vertOn=True, **lineprops)
Bases: matplotlib.widgets.Widget

Provide a vertical (default) and/or horizontal line cursor shared between multiple axes.

For the cursor to remain responsive you must keep a reference to it.
Example usage:

```python
from matplotlib.widgets import MultiCursor
from pylab import figure, show, np

t = np.arange(0.0, 2.0, 0.01)
s1 = np.sin(2*np.pi*t)
s2 = np.sin(4*np.pi*t)
fig = figure()
ax1 = fig.add_subplot(211)
ax1.plot(t, s1)
ax2 = fig.add_subplot(212, sharex=ax1)
ax2.plot(t, s2)
multi = MultiCursor(fig.canvas, (ax1, ax2), color='r', lw=1, horizOn=False, vertOn=True)
show()
```

```python
clear(event)
    clear the cursor
connect()
    connect events
disconnect()
    disconnect events
onmove(event)
```

class matplotlib.widgets.RadioButton(ax, labels, active=0, activecolor='blue')
Bases: matplotlib.widgets.AxesWidget

A GUI neutral radio button.

For the buttons to remain responsive you must keep a reference to this object.

The following attributes are exposed:

- **ax** The `matplotlib.axes.Axes` instance the buttons are in
- **activecolor** The color of the button when clicked
- **labels** A list of `matplotlib.text.Text` instances
- **circles** A list of `matplotlib.patches.Circle` instances
- **value_selected** A string listing the current value selected

Connect to the RadioButtons with the `on_clicked()` method

Add radio buttons to `matplotlib.axes.Axes` instance `ax`

- **labels** A `len(buttons)` list of labels as strings
- **active** The index into labels for the button that is active
**activecolor** The color of the button when clicked

**disconnect** *(cid)*
remove the observer with connection id *cid*

**on_clicked** *(func)*
When the button is clicked, call *func* with button label

A connection id is returned which can be used to disconnect

**set_active** *(index)*
Trigger which radio button to make active.

*index* is an index into the original label list that this object was constructed with. Raise ValueError if the index is invalid.

Callbacks will be triggered if *eventson* is True.

```python
class matplotlib.widgets.RectangleSelector(ax, onselect, drawtype='box',
                                           minspanx=None, minspany=None,
                                           useblit=False, lineprops=None, rectprops=None, spancoords='data',
                                           button=None, maxdist=10, marker_props=None, interactive=False, state_modifier_keys=None)
```

Bases: matplotlib.widgets._SelectorWidget

Select a rectangular region of an axes.

For the cursor to remain responsive you must keep a reference to it.

Example usage:

```python
from matplotlib.widgets import RectangleSelector
from pylab import *

def onselect(eclick, erelease):
    'eclick and erelease are matplotlib events at press and release'
    print('startposition : (%f, %f) ' % (eclick.xdata, eclick.ydata))
    print('endposition : (%f, %f) ' % (erelease.xdata, erelease.ydata))
    print('used button : %s, eclick.button' % eclick.button)

def toggle_selector(event):
    print('Key pressed.')
    if event.key in ['Q', 'q'] and toggle_selector.RS.active:
        print('RectangleSelector deactivated.')
        toggle_selector.RS.set_active(False)
    if event.key in ['A', 'a'] and not toggle_selector.RS.active:
        print('RectangleSelector activated.')
        toggle_selector.RS.set_active(True)

x = arange(100)/(99.0)
y = sin(x)
fig = figure
ax = subplot(111)
ax.plot(x, y)
```

58.1. matplotlib.widgets
Create a selector in `ax`. When a selection is made, clear the span and call `onselect` with:

```python
donselect(pos_1, pos_2)
```

and clear the drawn box/line. The `pos_1` and `pos_2` are arrays of length 2 containing the x- and y-coordinate.

If `minspanx` is not `None` then events smaller than `minspanx` in x direction are ignored (it’s the same for y).

The rectangle is drawn with `rectprops`; default:

```python
rectprops = dict(facecolor='red', edgecolor = 'black',
                 alpha=0.2, fill=True)
```

The line is drawn with `lineprops`; default:

```python
lineprops = dict(color='black', linestyle='-',
                 linewidth = 2, alpha=0.5)
```

Use `drawtype` if you want the mouse to draw a line, a box or nothing between click and actual position by setting

```python
drawtype = 'line', drawtype='box' or drawtype = 'none'.
```

`spancoords` is one of ‘data’ or ‘pixels’. If ‘data’, `minspanx` and `minspany` will be interpreted in the same coordinates as the x and y axis. If ‘pixels’, they are in pixels.

`button` is a list of integers indicating which mouse buttons should be used for rectangle selection. You can also specify a single integer if only a single button is desired. Default is `None`, which does not limit which button can be used.

**Note, typically:** 1 = left mouse button 2 = center mouse button (scroll wheel) 3 = right mouse button

`interactive` will draw a set of handles and allow you interact with the widget after it is drawn.

`state_modifier_keys` are keyboard modifiers that affect the behavior of the widget.

The defaults are: dict(move=’ ‘, clear='escape', square='shift', center='ctrl')

Keyboard modifiers, which: ‘move’: Move the existing shape. ‘clear’: Clear the current shape. ‘square’: Makes the shape square. ‘center’: Make the initial point the center of the shape. ‘square’ and ‘center’ can be combined.

`center`
   Center of rectangle

`corners`
   Corners of rectangle from lower left, moving clockwise.
draw_shape(*extents*)

edge_centers
    Midpoint of rectangle edges from left, moving clockwise.

extents
    Return (xmin, xmax, ymin, ymax).

gometry

class matplotlib.widgets.Slider(ax, label, valmin, valmax, valinit=0.5, valfmt='%1.2f',
    closedmin=True, closedmax=True, slidermin=None, slidermax=None, dragging=True, **kwargs)

Bases: matplotlib.widgets.AxesWidget

A slider representing a floating point range.

For the slider to remain responsive you must maintain a reference to it.

The following attributes are defined

- **ax**: the slider `matplotlib.axes.Axes` instance
- **val**: the current slider value
- **vline**: a `matplotlib.lines.Line2D` instance representing the initial value of the slider
- **poly**: a `matplotlib.patches.Polygon` instance which is the slider knob
- **valfmt**: the format string for formatting the slider text
- **label**: a `matplotlib.text.Text` instance for the slider label
- **closedmin**: whether the slider is closed on the minimum
- **closedmax**: whether the slider is closed on the maximum
- **slidermin**: another slider - if not `None`, this slider must be greater than `slidermin`
- **slidermax**: another slider - if not `None`, this slider must be less than `slidermax`
- **dragging**: allow for mouse dragging on slider

Call `on_changed()` to connect to the slider event

Create a slider from `valmin` to `valmax` in axes `ax`.

Additional kwargs are passed on to `self.poly` which is the `matplotlib.patches.Rectangle` that draws the slider knob. See the `matplotlib.patches.Rectangle` documentation for valid property names (e.g., `facecolor`, `edgecolor`, `alpha`, ...).

Parameters

- **ax**: Axes
  
The Axes to put the slider in
- **label**: str
  
  Slider label
- **valmin**: float
The minimum value of the slider

valmax : float
    The maximum value of the slider

valinit : float
    The slider initial position

label : str
    The slider label

valfmt : str
    Used to format the slider value, fprint format string

closedmin : bool
    Indicate whether the slider interval is closed on the bottom

closedmax : bool
    Indicate whether the slider interval is closed on the top

slidermin : Slider or None
    Do not allow the current slider to have a value less than slidermin

slidermax : Slider or None
    Do not allow the current slider to have a value greater than slidermax

dragging : bool
    if the slider can be dragged by the mouse

disconnect(cid)
    remove the observer with connection id cid

on_changed(func)
    When the slider value is changed, call func with the new slider position
    A connection id is returned which can be used to disconnect

reset()
    reset the slider to the initial value if needed

set_val(val)

class matplotlib.widgets.SpanSelector(ax, onselect, direction, minspan=None,
useblit=False, rectprops=None, onmove_callback=None, span_stays=False, button=None)
    Bases: matplotlib.widgets._SelectorWidget

Visually select a min/max range on a single axis and call a function with those values.
To guarantee that the selector remains responsive, keep a reference to it.
In order to turn off the SpanSelector, set `span_selector.active=False`. To turn it back on, set `span_selector.active=True`.

**Parameters**

- **ax**: `matplotlib.axes.Axes` object
- **onselect**: `func(min, max)`, min/max are floats
- **direction**: “horizontal” or “vertical”  
  The axis along which to draw the span selector
- **minspan**: float, default is None  
  If selection is less than `minspan`, do not call `onselect`
- **useblit**: bool, default is False  
  If True, use the backend-dependent blitting features for faster canvas updates. Only available for GTKAgg right now.
- **rectprops**: dict, default is None  
  Dictionary of `matplotlib.patches.Patch` properties
- **onmove_callback**: `func(min, max)`, min/max are floats, default is None  
  Called on mouse move while the span is being selected
- **span_stays**: bool, default is False  
  If True, the span stays visible after the mouse is released
- **button**: int or list of ints  
  Determines which mouse buttons activate the span selector
  
  - 1 = left mouse button  
  - 2 = center mouse button (scroll wheel)  
  - 3 = right mouse button

**Examples**

```python
>>> import matplotlib.pyplot as plt
>>> import matplotlib.widgets as mwidgets
>>> fig, ax = plt.subplots()
>>> ax.plot([1, 2, 3], [10, 50, 100])
>>> def onselect(vmin, vmax):
...     print(vmin, vmax)
... rectprops = dict(facecolor='blue', alpha=0.5)
>>> span = mwidgets.SpanSelector(ax, onselect, 'horizontal',
... rectprops=rectprops)
>>> fig.show()
```

See also: `widgets example code: span_selector.py`
ignore(event)
    return True if event should be ignored

def new_axes(ax)
    Set SpanSelector to operate on a new Axes

class matplotlib.widgets.SubplotTool(targetfig, toolfig)
    Bases: matplotlib.widgets.Widget
    A tool to adjust the subplot params of a matplotlib.figure.Figure.
    targetfig The figure instance to adjust.
    toolfig The figure instance to embed the subplot tool into. If None, a default figure will be created. If you are using this from the GUI

    def funcbottom(val)

    def funcwspace(val)

    def funclleft(val)

    def functop(val)

    def functime(val)

class matplotlib.widgets.ToolHandles(ax, x, y, marker='o', marker_props=None, useblit=True)
    Bases: object
    Control handles for canvas tools.

    Parameters
    ----------
    ax : matplotlib.axes.Axes
        Matplotlib axes where tool handles are displayed.
    x, y : 1D arrays
        Coordinates of control handles.
    marker : str
        Shape of marker used to display handle. See matplotlib.pyplot.plot.
    marker_props : dict
        Additional marker properties. See matplotlib.lines.Line2D.

    def closest(x, y)
        Return index and pixel distance to closest index.
set_animated(val)

set_data(pts, y=None)
    Set x and y positions of handles

set_visible(val)

x

y

class matplotlib.widgets.Widget
    Bases: object
    Abstract base class for GUI neutral widgets

active
    Is the widget active?

drawn = True

eventson = True

get_active()
    Get whether the widget is active.

ignore(event)
    Return True if event should be ignored.
    This method (or a version of it) should be called at the beginning of any event callback.

set_active(active)
    Set whether the widget is active.
Part III

The Matplotlib FAQ
59.1 Report a compilation problem

See *Getting help*. 

---

**Contents**

- *Installation*
  - Report a compilation problem
  - matplotlib compiled fine, but nothing shows up when I use it
  - How to completely remove matplotlib
    - *Source install*
  - How to Install
    - *Source install from git*
  - Linux Notes
  - OS-X Notes
    - *Which python for OS X?*
    - *Installing OSX binary wheels*
      - Python.org Python
      - Macports
      - Homebrew
      - Pip problems
    - *Installing via OSX mpkg installer package*
    - *Checking your installation*
  - Windows Notes
59.2 matplotlib compiled fine, but nothing shows up when I use it

The first thing to try is a `clean install` and see if that helps. If not, the best way to test your install is by running a script, rather than working interactively from a python shell or an integrated development environment such as IDLE which add additional complexities. Open up a UNIX shell or a DOS command prompt and cd into a directory containing a minimal example in a file. Something like simple_plot.py for example:

```python
import matplotlib.pyplot as plt
fig, ax = plt.subplots()
ax.plot([1,2,3])
plt.show()
```

and run it with:

```
python simple_plot.py --verbose-helpful
```

This will give you additional information about which backends matplotlib is loading, version information, and more. At this point you might want to make sure you understand matplotlib’s `configuration` process, governed by the `matplotlibrc` configuration file which contains instructions within and the concept of the matplotlib backend.

If you are still having trouble, see *Getting help*.

59.3 How to completely remove matplotlib

Occasionally, problems with matplotlib can be solved with a clean installation of the package.

The process for removing an installation of matplotlib depends on how matplotlib was originally installed on your system. Follow the steps below that goes with your original installation method to cleanly remove matplotlib from your system.

59.3.1 Source install

Unfortunately:

```
python setup.py clean
```

does not properly clean the build directory, and does nothing to the install directory. To cleanly rebuild:

1. Delete the caches from your `matplotlib configuration directory`.
2. Delete the build directory in the source tree.
3. Delete any matplotlib directories or eggs from your `installation directory`.
59.4 How to Install

59.4.1 Source install from git

Clone the main source using one of:

```
git clone git@github.com:matplotlib/matplotlib.git
```
or:

```
git clone git://github.com/matplotlib/matplotlib.git
```

and build and install as usual with:

```
> cd matplotlib
> python setup.py install
```

**Note:** If you are on debian/ubuntu, you can get all the dependencies required to build matplotlib with:

```
sudo apt-get build-dep python-matplotlib
```

If you are on Fedora/RedHat, you can get all the dependencies required to build matplotlib by first installing `yum-builddep` and then running:

```
su -c "yum-builddep python-matplotlib"
```

This does not build matplotlib, but it does get all of the build dependencies, which will make building from source easier.

If you want to be able to follow the development branch as it changes just replace the last step with (make sure you have `setuptools` installed):

```
> python setup.py develop
```

This creates links in the right places and installs the command line script to the appropriate places.

**Note:** Mac OS X users please see the *Building on OSX* guide.
Windows users please see the *Building on Windows* guide.

Then, if you want to update your matplotlib at any time, just do:

```
> git pull
```

When you run `git pull`, if the output shows that only Python files have been updated, you are all set. If C files have changed, you need to run the `python setup.py develop` command again to compile them.

There is more information on *using git* in the developer docs.
59.5 Linux Notes

Because most Linux distributions use some sort of package manager, we do not provide a pre-built binary for the Linux platform. Instead, we recommend that you use the “Add Software” method for your system to install matplotlib. This will guarantee that everything that is needed for matplotlib will be installed as well.

If, for some reason, you can not use the package manager, Linux usually comes with at least a basic build system. Follow the instructions found above for how to build and install matplotlib.

59.6 OS-X Notes

59.6.1 Which python for OS X?

Apple ships OS X with its own Python, in /usr/bin/python, and its own copy of matplotlib. Unfortunately, the way Apple currently installs its own copies of numpy, scipy and matplotlib means that these packages are difficult to upgrade (see system python packages). For that reason we strongly suggest that you install a fresh version of Python and use that as the basis for installing libraries such as numpy and matplotlib. One convenient way to install matplotlib with other useful Python software is to use one of the excellent Python scientific software collections that are now available:

- Anaconda from Continuum Analytics
- Canopy from Enthought

These collections include Python itself and a wide range of libraries; if you need a library that is not available from the collection, you can install it yourself using standard methods such as pip. Continuum and Enthought offer their own installation support for these collections; see the Anaconda and Canopy web pages for more information.

Other options for a fresh Python install are the standard installer from python.org, or installing Python using a general OSX package management system such as homebrew or macports. Power users on OSX will likely want one of homebrew or macports on their system to install open source software packages, but it is perfectly possible to use these systems with another source for your Python binary, such as Anaconda, Canopy or Python.org Python.

59.6.2 Installing OSX binary wheels

If you are using recent Python from https://www.python.org, Macports or Homebrew, then you can use the standard pip installer to install matplotlib binaries in the form of wheels.

Python.org Python

Install pip following the standard pip install instructions. For the impatient, open a new Terminal.app window and:

```
curl -O https://bootstrap.pypa.io/get-pip.py
```
Then (Python 2.7):

```
python get-pip.py
```

or (Python 3):

```
python3 get-pip.py
```

You can now install matplotlib and all its dependencies with:

```
pip install matplotlib
```

**Macports**

For Python 2.7:

```
sudo port install py27-pip
sudo pip-2.7 install matplotlib
```

For Python 3.4:

```
sudo port install py34-pip
sudo pip-3.4 install matplotlib
```

**Homebrew**

For Python 2.7:

```
pip2 install matplotlib
```

For Python 3.4:

```
pip3 install matplotlib
```

You might also want to install IPython; we recommend you install IPython with the IPython notebook option, like this:

- Python.org Python: `pip install ipython[notebook]`
- Macports: `sudo pip-2.7 install ipython[notebook]` or `sudo pip-3.4 install ipython[notebook]`
- Homebrew: `pip2 install ipython[notebook]` or `pip3 install ipython[notebook]`

**Pip problems**

If you get errors with pip trying to run a compiler like gcc or clang, then the first thing to try is to install `xcode` and retry the install. If that does not work, then check *Getting help.*
59.6.3 Installing via OSX mpkg installer package

matplotlib also has a disk image (.dmg) installer, which contains a typical Installer.app package to install matplotlib. You should use binary wheels instead of the disk image installer if you can, because:

- wheels work with Python.org Python, homebrew and macports, the disk image installer only works with Python.org Python.

- The disk image installer doesn’t check for recent versions of packages that matplotlib depends on, and unconditionally installs the versions of dependencies contained in the disk image installer. This can overwrite packages that you have already installed, which might cause problems for other packages, if you have a pre-existing Python.org setup on your computer.

If you still want to use the disk image installer, read on.

**Note:** Before installing via the disk image installer, be sure that all of the packages were compiled for the same version of python. Often, the download site for NumPy and matplotlib will display a supposed ‘current’ version of the package, but you may need to choose a different package from the full list that was built for your combination of python and OSX.

The disk image installer will have a .dmg extension, and will have a name like matplotlib-1.4.0-py2.7-macosx10.6.dmg. The name of the installer depends on the versions of python and matplotlib it was built for, and the version of OSX that the matching Python.org installer was built for. For example, if the matching Python.org Python installer was built for OSX 10.6 or greater, the dmg file will end in -macosx10.6.dmg.

You need to download this disk image file, open the disk image file by double clicking, and find the new matplotlib disk image icon on your desktop. Double click on that icon to show the contents of the image. Then double-click on the .mpkg icon, which will have a name like matplotlib-1.4.0-py2.7-macosx10.6.mpkg, it will run the Installer.app, prompt you for a password if you need system-wide installation privileges, and install to a directory like /Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/site-packages (exact path depends on your Python version).

59.6.4 Checking your installation

The new version of matplotlib should now be on your Python “path”. Check this with one of these commands at the Terminal.app command line:

```python
python2.7 -c 'import matplotlib; print(matplotlib.__version__, matplotlib.__file__)'
```

(Python 2.7) or:

```python
python3.4 -c 'import matplotlib; print(matplotlib.__version__, matplotlib.__file__)'
```

(Python 3.4). You should see something like this:

```
1.4.0 /Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/site-packages/
    matplotlib/__init__.pyc
```

where 1.4.0 is the matplotlib version you just installed, and the path following depends on whether you are using Python.org Python, Homebrew or Macports. If you see another version, or you get an error like this:
then check that the Python binary is the one you expected by doing one of these commands in Terminal.app:

```bash
which python2.7
```

or:

```bash
which python3.4
```

If you get the result `/usr/bin/python2.7`, then you are getting the Python installed with OSX, which is probably not what you want. Try closing and restarting Terminal.app before running the check again. If that doesn’t fix the problem, depending on which Python you wanted to use, consider reinstalling Python.org Python, or check your homebrew or macports setup. Remember that the disk image installer only works for Python.org Python, and will not get picked up by other Pythons. If all these fail, please let us know: see Getting help.

## 59.7 Windows Notes

See Windows.
CHAPTER SIXTY

USAGE

Contents

• Usage
  – General Concepts
  – Parts of a Figure
    * Figure
    * Axes
    * Axis
    * Artist
  – Types of inputs to plotting functions
  – Matplotlib, pyplot and pylab: how are they related?
  – Coding Styles
  – What is a backend?
  – WX backends
  – GTK and Cairo
  – How do I select PyQt4 or PySide?
  – What is interactive mode?
    * Interactive example
    * Non-interactive example
  * Summary
60.1 General Concepts

Matplotlib has an extensive codebase that can be daunting to many new users. However, most of matplotlib can be understood with a fairly simple conceptual framework and knowledge of a few important points.

Plotting requires action on a range of levels, from the most general (e.g., ‘contour this 2-D array’) to the most specific (e.g., ‘color this screen pixel red’). The purpose of a plotting package is to assist you in visualizing your data as easily as possible, with all the necessary control – that is, by using relatively high-level commands most of the time, and still have the ability to use the low-level commands when needed.

Therefore, everything in matplotlib is organized in a hierarchy. At the top of the hierarchy is the matplotlib “state-machine environment” which is provided by the matplotlib.pyplot module. At this level, simple functions are used to add plot elements (lines, images, text, etc.) to the current axes in the current figure.

Note: Pyplot’s state-machine environment behaves similarly to MATLAB and should be most familiar to users with MATLAB experience.

The next level down in the hierarchy is the first level of the object-oriented interface, in which pyplot is used only for a few functions such as figure creation, and the user explicitly creates and keeps track of the figure and axes objects. At this level, the user uses pyplot to create figures, and through those figures, one or more axes objects can be created. These axes objects are then used for most plotting actions.

For even more control – which is essential for things like embedding matplotlib plots in GUI applications – the pyplot level may be dropped completely, leaving a purely object-oriented approach.
60.2 Parts of a Figure

The whole figure. The figure keeps track of all the child Axes, a smattering of ‘special’ artists (titles, figure legends, etc), and the canvas. (Don’t worry too much about the canvas, it is crucial as it is the object that actually does the drawing to get you your plot, but as the user it is more-or-less invisible to you). A figure can have any number of Axes, but to be useful should have at least one.

The easiest way to create a new figure is with pyplot:
fig = plt.figure()  # an empty figure with no axes
fig, ax_lst = plt.subplots(2, 2)  # a figure with a 2x2 grid of Axes

60.2.2 Axes

This is what you think of as ‘a plot’, it is the region of the image with the data space (marked as the inner blue box). A given figure can contain many Axes, but a given Axes object can only be in one Figure. The Axes contains two (or three in the case of 3D) Axis objects (be aware of the difference between Axes and Axis) which take care of the data limits (the data limits can also be controlled via set via the set_xlim() and set_ylim() Axes methods). Each Axes has a title (set via set_title()), an x-label (set via set_xlabel()), and a y-label set via set_ylabel()).

The Axes class and it’s member functions are the primary entry point to working with the OO interface.

60.2.3 Axis

These are the number-line-like objects (circled in green). They take care of setting the graph limits and generating the ticks (the marks on the axis) and ticklabels (strings labeling the ticks). The location of the ticks is determined by a Locator object and the ticklabel strings are formatted by a Formatter. The combination of the correct Locator and Formatter gives very fine control over the tick locations and labels.

60.2.4 Artist

Basically everything you can see on the figure is an artist (even the Figure, Axes, and Axis objects). This includes Text objects, Line2D objects, collection objects, Patch objects ... (you get the idea). When the figure is rendered, all of the artists are drawn to the canvas. Most Artists are tied to an Axes; such an Artist cannot be shared by multiple Axes, or moved from one to another.

60.3 Types of inputs to plotting functions

All of plotting functions expect np.array or np.ma.masked_array as input. Classes that are ‘array-like’ such as pandas data objects and np.matrix may or may not work as intended. It is best to convert these to np.array objects prior to plotting.

For example, to covert a pandas.DataFrame

```python
a = pandas.DataFrame(np.random.rand(4,5), columns = list('abcde'))
a_asndarray = a.values
```

and to covert a np.matrix

```python
b = np.matrix([[1,2],[3,4]])
b_asarray = np.asarray(b)
```
60.4 Matplotlib, pyplot and pylab: how are they related?

Matplotlib is the whole package; `matplotlib.pyplot` is a module in matplotlib; and `pylab` is a module that gets installed alongside `matplotlib`.

Pyplot provides the state-machine interface to the underlying object-oriented plotting library. The state-machine implicitly and automatically creates figures and axes to achieve the desired plot. For example:

```python
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 2, 100)
plt.plot(x, x, label='linear')
plt.plot(x, x**2, label='quadratic')
plt.plot(x, x**3, label='cubic')
plt.xlabel('x label')
plt.ylabel('y label')
plt.title("Simple Plot")
plt.legend()
plt.show()
```

The first call to `plt.plot` will automatically create the necessary figure and axes to achieve the desired plot. Subsequent calls to `plt.plot` re-use the current axes and each add another line. Setting the title, legend, and axis labels also automatically use the current axes and set the title, create the legend, and label the axis respectively.

`pylab` is a convenience module that bulk imports `matplotlib.pyplot` (for plotting) and `numpy` (for mathematics and working with arrays) in a single name space. Although many examples use `pylab`, it is no longer recommended.

For non-interactive plotting it is suggested to use `pyplot` to create the figures and then the OO interface for plotting.

60.5 Coding Styles

When viewing this documentation and examples, you will find different coding styles and usage patterns. These styles are perfectly valid and have their pros and cons. Just about all of the examples can be converted into another style and achieve the same results. The only caveat is to avoid mixing the coding styles for your own code.

**Note:** Developers for matplotlib have to follow a specific style and guidelines. See *The Matplotlib Developers’ Guide.*
Of the different styles, there are two that are officially supported. Therefore, these are the preferred ways to use matplotlib.

For the pyplot style, the imports at the top of your scripts will typically be:

```python
import matplotlib.pyplot as plt
import numpy as np
```

Then one calls, for example, np.arange, np.zeros, np.pi, plt.figure, plt.plot, plt.show, etc. Use the pyplot interface for creating figures, and then use the object methods for the rest:

```python
import matplotlib.pyplot as plt
import numpy as np
x = np.arange(0, 10, 0.2)
y = np.sin(x)
fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(x, y)
plt.show()
```

So, why all the extra typing instead of the MATLAB-style (which relies on global state and a flat namespace)? For very simple things like this example, the only advantage is academic: the wordier styles are more explicit, more clear as to where things come from and what is going on. For more complicated applications, this explicitness and clarity becomes increasingly valuable, and the richer and more complete object-oriented interface will likely make the program easier to write and maintain.

Typically one finds oneself making the same plots over and over again, but with different data sets, which leads to needing to write specialized functions to do the plotting. The recommended function signature is something like:

```python
def my_plotter(ax, data1, data2, param_dict):
    """
    A helper function to make a graph
    
    Parameters
    ----------
    ax : Axes
        The axes to draw to
    data1 : array
        The x data
    data2 : array
        The y data
    param_dict : dict
        Dictionary of kwargs to pass to ax.plot
    
    Returns
    -------
    out : list
        list of artists added
    """
```
```python
out = ax.plot(data1, data2, **param_dict)
return out
```

which you would then use as:

```python
fig, ax = plt.subplots(1, 1)
my_plotter(ax, data1, data2, {'marker':'x'})
```

or if you wanted to have 2 sub-plots:

```python
fig, (ax1, ax2) = plt.subplots(1, 2)
my_plotter(ax1, data1, data2, {'marker':'x'})
my_plotter(ax2, data3, data4, {'marker':'o'})
```

Again, for these simple examples this style seems like overkill, however once the graphs get slightly more complex it pays off.

### 60.6 What is a backend?

A lot of documentation on the website and in the mailing lists refers to the “backend” and many new users are confused by this term. matplotlib targets many different use cases and output formats. Some people use matplotlib interactively from the python shell and have plotting windows pop up when they type commands. Some people embed matplotlib into graphical user interfaces like wxpython or pygtk to build rich applications. Others use matplotlib in batch scripts to generate postscript images from some numerical simulations, and still others in web application servers to dynamically serve up graphs.

To support all of these use cases, matplotlib can target different outputs, and each of these capabilities is called a backend; the “frontend” is the user facing code, i.e., the plotting code, whereas the “backend” does all the hard work behind-the-scenes to make the figure. There are two types of backends: user interface backends (for use in pygtk, wxpython, tkinter, qt4, or macosx; also referred to as “interactive backends”) and hardcopy backends to make image files (PNG, SVG, PDF, PS; also referred to as “non-interactive backends”).

There are four ways to configure your backend. If they conflict each other, the method mentioned last in the following list will be used, e.g. calling `use()` will override the setting in your `matplotlibrc`.

1. The backend parameter in your `matplotlibrc` file (see Customizing matplotlib):

   ```
   backend : WXAgg  # use wxpython with antigrain (agg) rendering
   ```

2. Setting the `MPLBACKEND` environment variable, either for your current shell or for a single script:

   ```
   $ export MPLBACKEND="module://my_backend"
   $ python simple_plot.py
   $ MPLBACKEND="module://my_backend" python simple_plot.py
   ```

   Setting this environment variable will override the `backend` parameter in any `matplotlibrc`, even if there is a `matplotlibrc` in your current working directory. Therefore setting `MPLBACKEND` globally, e.g. in your `.bashrc` or `.profile`, is discouraged as it might lead to counter-intuitive behavior.
3. To set the backend for a single script, you can alternatively use the `-d` command line argument:

```
> python script.py -d backend
```

This method is deprecated as the `-d` argument might conflict with scripts which parse command line arguments (see issue #1986). You should use `MPLBACKEND` instead.

4. If your script depends on a specific backend you can use the `use()` function:

```python
import matplotlib
matplotlib.use('PS')  # generate postscript output by default
```

If you use the `use()` function, this must be done before importing `matplotlib.pyplot`. Calling `use()` after pyplot has been imported will have no effect. Using `use()` will require changes in your code if users want to use a different backend. Therefore, you should avoid explicitly calling `use()` unless absolutely necessary.

---

**Note:** Backend name specifications are not case-sensitive; e.g., ‘GTKAgg’ and ‘gtkagg’ are equivalent.

With a typical installation of matplotlib, such as from a binary installer or a Linux distribution package, a good default backend will already be set, allowing both interactive work and plotting from scripts, with output to the screen and/or to a file, so at least initially you will not need to use any of the methods given above.

If, however, you want to write graphical user interfaces, or a web application server (Matplotlib in a web application server), or need a better understanding of what is going on, read on. To make things a little more customizable for graphical user interfaces, matplotlib separates the concept of the renderer (the thing that actually does the drawing) from the canvas (the place where the drawing goes). The canonical renderer for user interfaces is Agg which uses the Anti-Grain Geometry C++ library to make a raster (pixel) image of the figure. All of the user interfaces except macosx can be used with agg rendering, e.g., WXAgg, GTKAgg, QT4Agg, QT5Agg, TkAgg. In addition, some of the user interfaces support other rendering engines. For example, with GTK, you can also select GDK rendering (backend GTK deprecated in 2.0) or Cairo rendering (backend GTKcairo).

For the rendering engines, one can also distinguish between vector or raster renderers. Vector graphics languages issue drawing commands like “draw a line from this point to this point” and hence are scale free, and raster backends generate a pixel representation of the line whose accuracy depends on a DPI setting.

Here is a summary of the matplotlib renderers (there is an eponymous backed for each; these are non-interactive backends, capable of writing to a file):
### Filetypes Table

<table>
<thead>
<tr>
<th>Renderer</th>
<th>Filetypes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AGG</td>
<td>png</td>
<td><em>raster graphics</em> – high quality images using the Anti-Grain Geometry engine</td>
</tr>
<tr>
<td>PS</td>
<td>ps eps</td>
<td><em>vector graphics</em> – Postscript output</td>
</tr>
<tr>
<td>PDF</td>
<td>pdf</td>
<td><em>vector graphics</em> – Portable Document Format</td>
</tr>
<tr>
<td>SVG</td>
<td>svg</td>
<td><em>vector graphics</em> – Scalable Vector Graphics</td>
</tr>
<tr>
<td>Cairo</td>
<td>png ps pdf svg</td>
<td><em>vector graphics</em> – Cairo graphics</td>
</tr>
<tr>
<td>GDK</td>
<td>png jpg tiff</td>
<td><em>raster graphics</em> – the Gimp Drawing Kit Deprecated in 2.0</td>
</tr>
</tbody>
</table>

And here are the user interfaces and renderer combinations supported; these are interactive backends, capable of displaying to the screen and of using appropriate renderers from the table above to write to a file:

<table>
<thead>
<tr>
<th>Backend</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GTK-Agg</td>
<td>Agg rendering to a <em>GTK</em> 2.x canvas (requires PyGTK and pycairo or cairocffi; Python2 only)</td>
</tr>
<tr>
<td>GTK3Agg</td>
<td>Agg rendering to a <em>GTK</em> 3.x canvas (requires PyGObject and pycairo or cairocffi)</td>
</tr>
<tr>
<td>GTK-Cairo</td>
<td>Cairo rendering to a <em>GTK</em> 2.x canvas (requires PyGTK and pycairo or cairocffi; Python2 only)</td>
</tr>
<tr>
<td>GTK3Cairo</td>
<td>Cairo rendering to a <em>GTK</em> 3.x canvas (requires PyGObject and pycairo or cairocffi; Python2 only)</td>
</tr>
<tr>
<td>WX-Agg</td>
<td>Agg rendering to a <em>wxWidgets</em> canvas (requires wxPython)</td>
</tr>
<tr>
<td>WX</td>
<td>Native <em>wxWidgets</em> drawing to a <em>wxWidgets</em> Canvas (not recommended and deprecated in 2.0) (requires wxPython)</td>
</tr>
<tr>
<td>TkAgg</td>
<td>Agg rendering to a <em>Tk</em> canvas (requires TkInter)</td>
</tr>
<tr>
<td>Qt4Agg</td>
<td>Agg rendering to a <em>Qt4</em> canvas (requires PyQt4 or pyside)</td>
</tr>
<tr>
<td>Qt5Agg</td>
<td>Agg rendering to a <em>Qt5</em> canvas (requires PyQt5)</td>
</tr>
<tr>
<td>macosx</td>
<td>Cocoa rendering in OSX windows (presently lacks blocking show() behavior when matplotlib is in non-interactive mode)</td>
</tr>
</tbody>
</table>

### 60.7 WX backends

At present the release version of wxPython (also known as wxPython classic) does not support python3. A work in progress redesigned version known as wxPython-Phoenix does support python3. Matplotlib should work with both versions.

### 60.8 GTK and Cairo

Both GTK2 and GTK3 have implicit dependencies on PyCairo regardless of the specific Matplotlib backend used. Unfortunately the latest release of PyCairo for Python3 does not implement the Python wrappers needed for the GTK3Agg backend. Cairocffi can be used as a replacement which implements the correct wrapper.
60.9 How do I select PyQt4 or PySide?

You can choose either PyQt4 or PySide when using the qt4 backend by setting the appropriate value for backend.qt4 in your matplotlibrc file. The default value is PyQt4.

The setting in your matplotlibrc file can be overridden by setting the QT_API environment variable to either pyqt or pyside to use PyQt4 or PySide, respectively.

Since the default value for the bindings to be used is PyQt4, matplotlib first tries to import it, if the import fails, it tries to import PySide.

60.10 What is interactive mode?

Use of an interactive backend (see What is a backend?) permits—but does not by itself require or ensure—plotting to the screen. Whether and when plotting to the screen occurs, and whether a script or shell session continues after a plot is drawn on the screen, depends on the functions and methods that are called, and on a state variable that determines whether matplotlib is in “interactive mode”. The default Boolean value is set by the matplotlibrc file, and may be customized like any other configuration parameter (see Customizing matplotlib). It may also be set via matplotlib.interactive(), and its value may be queried via matplotlib.is_interactive(). Turning interactive mode on and off in the middle of a stream of plotting commands, whether in a script or in a shell, is rarely needed and potentially confusing, so in the following we will assume all plotting is done with interactive mode either on or off.

Note: Major changes related to interactivity, and in particular the role and behavior of show(), were made in the transition to matplotlib version 1.0, and bugs were fixed in 1.0.1. Here we describe the version 1.0.1 behavior for the primary interactive backends, with the partial exception of macosx.

Interactive mode may also be turned on via matplotlib.pyplot.ion(), and turned off via matplotlib.pyplot.ioff().

Note: Interactive mode works with suitable backends in ipython and in the ordinary python shell, but it does not work in the IDLE IDE. If the default backend does not support interactivity, an interactive backend can be explicitly activated using any of the methods discussed in What is a backend?.

60.10.1 Interactive example

From an ordinary python prompt, or after invoking ipython with no options, try this:

```python
import matplotlib.pyplot as plt
plt.ion()
plt.plot([1.6, 2.7])
```

Assuming you are running version 1.0.1 or higher, and you have an interactive backend installed and selected by default, you should see a plot, and your terminal prompt should also be active; you can type additional
commands such as:

```python
plt.title("interactive test")
plt.xlabel("index")
```

and you will see the plot being updated after each line. This is because you are in interactive mode and you are using pyplot functions. Now try an alternative method of modifying the plot. Get a reference to the Axes instance, and call a method of that instance:

```python
ax = plt.gca()
ax.plot([3.1, 2.2])
```

Nothing changed, because the Axes methods do not include an automatic call to `draw_if_interactive()`; that call is added by the pyplot functions. If you are using methods, then when you want to update the plot on the screen, you need to call `draw()`:

```python
plt.draw()
```

Now you should see the new line added to the plot.

### 60.10.2 Non-interactive example

Start a fresh session as in the previous example, but now turn interactive mode off:

```python
import matplotlib.pyplot as plt
plt.ioff()
plt.plot([1.6, 2.7])
```

Nothing happened—or at least nothing has shown up on the screen (unless you are using `macosx` backend, which is anomalous). To make the plot appear, you need to do this:

```python
plt.show()
```

Now you see the plot, but your terminal command line is unresponsive; the `show()` command blocks the input of additional commands until you manually kill the plot window.

What good is this—being forced to use a blocking function? Suppose you need a script that plots the contents of a file to the screen. You want to look at that plot, and then end the script. Without some blocking command such as `show()`, the script would flash up the plot and then end immediately, leaving nothing on the screen.

In addition, non-interactive mode delays all drawing until `show()` is called; this is more efficient than redrawing the plot each time a line in the script adds a new feature.

Prior to version 1.0, `show()` generally could not be called more than once in a single script (although sometimes one could get away with it); for version 1.0.1 and above, this restriction is lifted, so one can write a script like this:

```python
import numpy as np
import matplotlib.pyplot as plt
plt.ioff()
```
which makes three plots, one at a time.

### 60.10.3 Summary

In interactive mode, pyplot functions automatically draw to the screen.

When plotting interactively, if using object method calls in addition to pyplot functions, then call `draw()` whenever you want to refresh the plot.

Use non-interactive mode in scripts in which you want to generate one or more figures and display them before ending or generating a new set of figures. In that case, use `show()` to display the figure(s) and to block execution until you have manually destroyed them.
chapter sixtyone

how-to

contents

• how-to
  – plotting: howto
    • plot numpy.datetime64 values
    • find all objects in a figure of a certain type
    • how to prevent ticklabels from having an offset
    • save transparent figures
    • save multiple plots to one pdf file
    • move the edge of an axes to make room for tick labels
    • automatically make room for tick labels
    • configure the tick linewidths
    • align my ylabels across multiple subplots
    • skip dates where there is no data
    • control the depth of plot elements
    • make the aspect ratio for plots equal
    • multiple y-axis scales
    • generate images without having a window appear
    • use show()
    • interpreting box plots and violin plots
  – contributing: howto
    • request a new feature
    • reporting a bug or submitting a patch
    • contribute to matplotlib documentation
61.1 Plotting: howto

61.1.1 Plot numpy.datetime64 values

For Matplotlib to plot dates (or any scalar with units) a converter to float needs to be registered with the `matplotlib.units` module. The current best converters for `datetime64` values are in `pandas`. Simply importing `pandas`

```python
import pandas as pd
```

should be sufficient as `pandas` will try to install the converters on import. If that does not work, or you need to reset `munits.registry` you can explicitly install the `pandas` converters by

```python
from pandas.tseries import converter as pdtc
pdtc.register()
```

If you only want to use the `pandas` converter for `datetime64` values

```python
from pandas.tseries import converter as pdtc
import matplotlib.units as munits
import numpy as np
munits.registry[np.datetime64] = pdtc.DatetimeConverter()
```

61.1.2 Find all objects in a figure of a certain type

Every Matplotlib artist (see Artist tutorial) has a method called `findobj()` that can be used to recursively search the artist for any artists it may contain that meet some criteria (e.g., match all `Line2D` instances or match some arbitrary filter function). For example, the following snippet finds every object in the figure which has a `set_color` property and makes the object blue:

```python
def myfunc(x):
    return hasattr(x, 'set_color')
```
for o in fig.findobj(myfunc):
    o.set_color('blue')

You can also filter on class instances:

```python
import matplotlib.text as text
for o in fig.findobj(text.Text):
    o.set_fontstyle('italic')
```

### 61.1.3 How to prevent ticklabels from having an offset

The default formatter will use an offset to reduce the length of the ticklabels. To turn this feature off on a per-axis basis:

```
ax.get_xaxis().get_major_formatter().set_useOffset(False)
```

set the rcParam `axes.formatter.useoffset`, or use a different formatter. See `ticker` for details.

### 61.1.4 Save transparent figures

The `savefig()` command has a keyword argument `transparent` which, if ‘True’, will make the figure and axes backgrounds transparent when saving, but will not affect the displayed image on the screen.

If you need finer grained control, e.g., you do not want full transparency or you want to affect the screen displayed version as well, you can set the alpha properties directly. The figure has a `Rectangle` instance called `patch` and the axes has a `Rectangle` instance called `patch`. You can set any property on them directly (`facecolor`, `edgecolor`, `linewidth`, `linestyle`, `alpha`). e.g.:

```python
fig = plt.figure()
f = fig.patch.set_alpha(0.5)
a = fig.add_subplot(111)
a.set_alpha(0.5)
```

If you need all the figure elements to be transparent, there is currently no global alpha setting, but you can set the alpha channel on individual elements, e.g.:

```python
ax.plot(x, y, alpha=0.5)
a.set_xlabel('volts', alpha=0.5)
```

### 61.1.5 Save multiple plots to one pdf file

Many image file formats can only have one image per file, but some formats support multi-page files. Currently only the pdf backend has support for this. To make a multi-page pdf file, first initialize the file:

```python
from matplotlib.backends.backend_pdf import PdfPages
pp = PdfPages('multipage.pdf')
```
You can give the `PdfPages` object to `savefig()`, but you have to specify the format:

```python
plt.savefig(pp, format='pdf')
```

An easier way is to call `PdfPages.savefig`:

```python
pp.savefig()
```

Finally, the multipage pdf object has to be closed:

```python
pp.close()
```

### 61.1.6 Move the edge of an axes to make room for tick labels

For subplots, you can control the default spacing on the left, right, bottom, and top as well as the horizontal and vertical spacing between multiple rows and columns using the `matplotlib.figure.Figure.subplots_adjust()` method (in pyplot it is `subplots_adjust()`). For example, to move the bottom of the subplots up to make room for some rotated x tick labels:

```python
fig = plt.figure()
fig.subplots_adjust(bottom=0.2)
ax = fig.add_subplot(111)
```

You can control the defaults for these parameters in your `matplotlibrc` file; see *Customizing matplotlib*. For example, to make the above setting permanent, you would set:

```python
figure.subplot.bottom : 0.2  # the bottom of the subplots of the figure
```

The other parameters you can configure are, with their defaults

- `left = 0.125` the left side of the subplots of the figure
- `right = 0.9` the right side of the subplots of the figure
- `bottom = 0.1` the bottom of the subplots of the figure
- `top = 0.9` the top of the subplots of the figure
- `wspace = 0.2` the amount of width reserved for blank space between subplots, expressed as a fraction of the average axis width
- `hspace = 0.2` the amount of height reserved for white space between subplots, expressed as a fraction of the average axis height

If you want additional control, you can create an `Axes` using the `axes()` command (or equivalently the figure `add_axes()` method), which allows you to specify the location explicitly:

```python
ax = fig.add_axes([left, bottom, width, height])
```

where all values are in fractional (0 to 1) coordinates. See `pylab_examples example code: axes_demo.py` for an example of placing axes manually.
61.1.7 Automatically make room for tick labels

**Note:** This is now easier to handle than ever before. Calling `tight_layout()` can fix many common layout issues. See the Tight Layout guide.

The information below is kept here in case it is useful for other purposes.

In most use cases, it is enough to simply change the subplots adjust parameters as described in *Move the edge of an axes to make room for tick labels*. But in some cases, you don’t know ahead of time what your tick labels will be, or how large they will be (data and labels outside your control may be being fed into your graphing application), and you may need to automatically adjust your subplot parameters based on the size of the tick labels. Any *Text* instance can report its extent in window coordinates (a negative x coordinate is outside the window), but there is a rub.

The *RendererBase* instance, which is used to calculate the text size, is not known until the figure is drawn (*draw()*). After the window is drawn and the text instance knows its renderer, you can call `get_window_extent()`. One way to solve this chicken and egg problem is to wait until the figure is drawn by connecting (*mpl_connect()* to the “on_draw” signal (*DrawEvent*) and get the window extent there, and then do something with it, e.g., move the left of the canvas over; see Event handling and picking.

Here is an example that gets a bounding box in relative figure coordinates (0..1) of each of the labels and uses it to move the left of the subplots over so that the tick labels fit in the figure

```python
import matplotlib.pyplot as plt
import matplotlib.transforms as mtransforms
fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(range(10))
ax.set_yticks((2,5,7))
labels = ax.set_yticklabels(('really, really, really', 'long', 'labels'))
def on_draw(event):
    bboxes = []
    for label in labels:
        bbox = label.get_window_extent()
        # the figure transform goes from relative coords->pixels and we
        # want the inverse of that
        bboxi = bbox.inverse_transformed(fig.transFigure)
        bboxes.append(bboxi)

    # this is the bbox that bounds all the bboxes, again in relative
    # figure coords
    bbox = mtransforms.Bbox.union(bboxes)
    if fig.subplotpars.left < bbox.width:
        # we need to move it over
        fig.subplots_adjust(left=1.1*bbox.width)  # pad a little
        fig.canvas.draw()
    return False

fig.canvas.mpl_connect('draw_event', on_draw)
```

61.1. Plotting: howto
61.1.8 Configure the tick linethickness

In Matplotlib, the ticks are markers. All Line2D objects support a line (solid, dashed, etc) and a marker (circle, square, tick). The tick linethickness is controlled by the “markeredgewidth” property:

```python
import matplotlib.pyplot as plt
fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(range(10))

for line in ax.get_xticklines() + ax.get_yticklines():
    line.set_markersize(10)
plt.show()
```

The other properties that control the tick marker, and all markers, are markerfacecolor, markeredgecolor, markeredgewidth, markersize. For more information on configuring ticks, see Axis containers and Tick containers.
61.1.9 Align my ylabels across multiple subplots

If you have multiple subplots over one another, and the y data have different scales, you can often get ylabels that do not align vertically across the multiple subplots, which can be unattractive. By default, Matplotlib positions the x location of the ylabel so that it does not overlap any of the y ticks. You can override this default behavior by specifying the coordinates of the label. The example below shows the default behavior in the left subplots, and the manual setting in the right subplots.

```python
import numpy as np
import matplotlib.pyplot as plt

box = dict(facecolor='yellow', pad=5, alpha=0.2)

fig = plt.figure()
fig.subplots_adjust(left=0.2, wspace=0.6)

# Fixing random state for reproducibility
np.random.seed(19680801)

ax1 = fig.add_subplot(221)
ax1.plot(2000*np.random.rand(10))
ax1.set_title('ylabels not aligned')
ax1.set_ylabel('misaligned 1', bbox=box)
ax1.set_ylim(0, 2000)

ax3 = fig.add_subplot(223)
ax3.set_ylabel('misaligned 2', bbox=box)
ax3.plot(np.random.rand(10))

ax2 = fig.add_subplot(222)
ax2.set_title('ylabels aligned')
ax2.plot(2000*np.random.rand(10))
ax2.set_ylabel('aligned 1', bbox=box)
ax2.yaxis.set_label_coords(-0.3, 0.5)
ax2.set_ylim(0, 2000)

ax4 = fig.add_subplot(224)
ax4.plot(np.random.rand(10))
ax4.set_ylabel('aligned 2', bbox=box)
ax4.yaxis.set_label_coords(-0.3, 0.5)

plt.show()
```
61.1.10 Skip dates where there is no data

When plotting time series, e.g., financial time series, one often wants to leave out days on which there is no data, e.g., weekends. By passing in dates on the x-axis, you get large horizontal gaps on periods when there is not data. The solution is to pass in some proxy x-data, e.g., evenly sampled indices, and then use a custom formatter to format these as dates. The example below shows how to use an ‘index formatter’ to achieve the desired plot:

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.mlab as mlab
import matplotlib.ticker as ticker

r = mlab.csv2rec('..data/aapl.csv')
r.sort()
r = r[-30:]  # get the last 30 days
N = len(r)
ind = np.arange(N)  # the evenly spaced plot indices

def format_date(x, pos=None):
    thisind = np.clip(int(x+0.5), 0, N-1)
    return r.date[thisind].strftime('%Y-%m-%d')
```

Control the depth of plot elements

Within an axes, the order that the various lines, markers, text, collections, etc appear is determined by the set_zorder() property. The default order is patches, lines, text, with collections of lines and collections of patches appearing at the same level as regular lines and patches, respectively:

```python
line, = ax.plot(x, y, zorder=10)
```

You can also use the Axes property set_axisbelow() to control whether the grid lines are placed above or below your other plot elements.

Make the aspect ratio for plots equal

The Axes property set_aspect() controls the aspect ratio of the axes. You can set it to be ‘auto’, ‘equal’, or some ratio which controls the ratio:

```python
ax = fig.add_subplot(111, aspect='equal')
```

Multiple y-axis scales

A frequent request is to have two scales for the left and right y-axis, which is possible using twinx() (more than two scales are not currently supported, though it is on the wish list). This works pretty well, though there are some quirks when you are trying to interactively pan and zoom, because both scales do not get the signals.

The approach uses twinx() (and its sister twiny()) to use 2 different axes, turning the axes rectangular frame off on the 2nd axes to keep it from obscuring the first, and manually setting the tick locs and labels as desired. You can use separate matplotlib.ticker formatters and locators as desired because the two axes are independent.
61.1.14 Generate images without having a window appear

The easiest way to do this is use a non-interactive backend (see What is a backend?) such as Agg (for PNGs), PDF, SVG or PS. In your figure-generating script, just call the `matplotlib.use()` directive before importing pylab or pyplot:

```python
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
plt.plot([1,2,3])
plt.savefig('myfig')
```

See also:

Matplotlib in a web application server for information about running matplotlib inside of a web application.

61.1.15 Use `show()`

When you want to view your plots on your display, the user interface backend will need to start the GUI mainloop. This is what `show()` does. It tells Matplotlib to raise all of the figure windows created so far and start the mainloop. Because this mainloop is blocking by default (i.e., script execution is paused), you should only call this once per script, at the end. Script execution is resumed after the last window is closed.
Therefore, if you are using Matplotlib to generate only images and do not want a user interface window, you do not need to call `show` (see `Generate images without having a window appear` and `What is a backend`).

**Note:** Because closing a figure window invokes the destruction of its plotting elements, you should call `savefig()` before calling `show` if you wish to save the figure as well as view it.

New in version v1.0.0: `show` now starts the GUI mainloop only if it isn’t already running. Therefore, multiple calls to `show` are now allowed.

Having `show` block further execution of the script or the python interpreter depends on whether Matplotlib is set for interactive mode or not. In non-interactive mode (the default setting), execution is paused until the last figure window is closed. In interactive mode, the execution is not paused, which allows you to create additional figures (but the script won’t finish until the last figure window is closed).

**Note:** Support for interactive/non-interactive mode depends upon the backend. Until version 1.0.0 (and subsequent fixes for 1.0.1), the behavior of the interactive mode was not consistent across backends. As of v1.0.1, only the macosx backend differs from other backends because it does not support non-interactive mode.

Because it is expensive to draw, you typically will not want Matplotlib to redraw a figure many times in a script such as the following:

```python
plot([1,2,3])  # draw here ?
xlabel('time')  # and here ?
ylabel('volts')  # and here ?
title('a simple plot')  # and here ?
show()
```

However, it is possible to force Matplotlib to draw after every command, which might be what you want when working interactively at the python console (see `Using matplotlib in a python shell`), but in a script you want to defer all drawing until the call to `show`. This is especially important for complex figures that take some time to draw. `show()` is designed to tell Matplotlib that you’re all done issuing commands and you want to draw the figure now.

**Note:** `show()` should typically only be called at most once per script and it should be the last line of your script. At that point, the GUI takes control of the interpreter. If you want to force a figure draw, use `draw()` instead.

Many users are frustrated by `show` because they want it to be a blocking call that raises the figure, pauses the script until they close the figure, and then allow the script to continue running until the next figure is created and the next show is made. Something like this:

```python
# WARNING : illustrating how NOT to use show
for i in range(10):
    # make figure i
    show()
```
This is not what `show` does and unfortunately, because doing blocking calls across user interfaces can be tricky, is currently unsupported, though we have made significant progress towards supporting blocking events.

New in version v1.0.0: As noted earlier, this restriction has been relaxed to allow multiple calls to `show`. In most backends, you can now expect to be able to create new figures and raise them in a subsequent call to `show` after closing the figures from a previous call to `show`.

### 61.1.16 Interpreting box plots and violin plots

Tukey’s box plots (Robert McGill, John W. Tukey and Wayne A. Larsen: “The American Statistician” Vol. 32, No. 1, Feb., 1978, pp. 12-16) are statistical plots that provide useful information about the data distribution such as skewness. However, bar plots with error bars are still the common standard in most scientific literature, and thus, the interpretation of box plots can be challenging for the unfamiliar reader. The figure below illustrates the different visual features of a box plot.

![Box Plot Diagram](image)

Q1: Quartile 1, or median of the left data subset after dividing the original data set into 2 subsets via the median (25% of the data points fall below this threshold)

Q3: Quartile 3, median of the right data subset (75% of the data points fall below this threshold)

IQR: Interquartile-range, Q3 - Q1

Outliers: Data points are considered to be outliers if:

- value < Q1 - 1.5 x IQR or
- value > Q3 + 1.5 x IQR

Violin plots are closely related to box plots but add useful information such as the distribution of the sample data (density trace). Violin plots were added in Matplotlib 1.4.

### 61.2 Contributing: howto

#### 61.2.1 Request a new feature

Is there a feature you wish Matplotlib had? Then ask! The best way to get started is to email the developer mailing list for discussion. This is an open source project developed primarily in the contributors free time, so there is no guarantee that your feature will be added. The best way to get the feature you need added is to contribute it yourself.
61.2.2 Reporting a bug or submitting a patch

The development of Matplotlib is organized through github. If you would like to report a bug or submit a patch please use that interface.

To report a bug create an issue on github (this requires having a github account). Please include a Short, Self Contained, Correct (Compilable), Example demonstrating what the bug is. Including a clear, easy to test example makes it easy for the developers to evaluate the bug. Expect that the bug reports will be a conversation. If you do not want to register with github, please email bug reports to the mailing list.

The easiest way to submit patches to Matplotlib is through pull requests on github. Please see the The Matplotlib Developers’ Guide for the details.

61.2.3 Contribute to Matplotlib documentation

Matplotlib is a big library, which is used in many ways, and the documentation has only scratched the surface of everything it can do. So far, the place most people have learned all these features are through studying the examples (Search examples), which is a recommended and great way to learn, but it would be nice to have more official narrative documentation guiding people through all the dark corners. This is where you come in.

There is a good chance you know more about Matplotlib usage in some areas, the stuff you do every day, than many of the core developers who wrote most of the documentation. Just pulled your hair out compiling Matplotlib for windows? Write a FAQ or a section for the Installation page. Are you a digital signal processing wizard? Write a tutorial on the signal analysis plotting functions like xcorr(), psd() and specgram(). Do you use Matplotlib with django or other popular web application servers? Write a FAQ or tutorial and we’ll find a place for it in the User’s Guide. Bundle Matplotlib in a py2exe app? ... I think you get the idea.

Matplotlib is documented using the sphinx extensions to restructured text (ReST). sphinx is an extensible python framework for documentation projects which generates HTML and PDF, and is pretty easy to write; you can see the source for this document or any page on this site by clicking on the Show Source link at the end of the page in the sidebar.

The sphinx website is a good resource for learning sphinx, but we have put together a cheat-sheet at Developer’s tips for documenting matplotlib which shows you how to get started, and outlines the Matplotlib conventions and extensions, e.g., for including plots directly from external code in your documents.

Once your documentation contributions are working (and hopefully tested by actually building the docs) you can submit them as a patch against git. See Install git and Reporting a bug or submitting a patch. Looking for something to do? Search for TODO or look at the open issues on github.

61.3 Matplotlib in a web application server

Many users report initial problems trying to use matplotlib in web application servers, because by default Matplotlib ships configured to work with a graphical user interface which may require an X11 connection. Since many barebones application servers do not have X11 enabled, you may get errors if you don’t configure Matplotlib for use in these environments. Most importantly, you need to decide what kinds of images you want to generate (PNG, PDF, SVG) and configure the appropriate default backend. For 99% of users,
this will be the Agg backend, which uses the C++ antigrain rendering engine to make nice PNGs. The Agg backend is also configured to recognize requests to generate other output formats (PDF, PS, EPS, SVG). The easiest way to configure Matplotlib to use Agg is to call:

```python
# do this before importing pylab or pyplot
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
```

For more on configuring your backend, see *What is a backend?*.

Alternatively, you can avoid pylab/pyplot altogether, which will give you a little more control, by calling the API directly as shown in *api example code: agg_oo.py*.

You can either generate hardcopy on the filesystem by calling savefig:

```python
# do this before importing pylab or pyplot
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot([1,2,3])
fig.savefig('test.png')
```

or by saving to a file handle:

```python
import sys
fig.savefig(sys.stdout)
```

Here is an example using Pillow. First, the figure is saved to a BytesIO object which is then fed to Pillow for further processing:

```python
from io import BytesIO
from PIL import Image
imgdata = BytesIO()
fig.savefig(imgdata, format='png')
imgdata.seek(0)  # rewind the data
im = Image.open(imgdata)
```

### 61.3.1 Matplotlib with apache

TODO; see *Contribute to Matplotlib documentation*.

### 61.3.2 Matplotlib with django

TODO; see *Contribute to Matplotlib documentation*. 
61.3.3 Matplotlib with zope

TODO; see Contribute to Matplotlib documentation.

61.3.4 Clickable images for HTML

Andrew Dalke of Dalke Scientific has written a nice article on how to make html click maps with Matplotlib agg PNGs. We would also like to add this functionality to SVG. If you are interested in contributing to these efforts that would be great.

61.4 Search examples

The nearly 300 code Matplotlib Examples included with the Matplotlib source distribution are full-text searchable from the search page, but sometimes when you search, you get a lot of results from the The Matplotlib API or other documentation that you may not be interested in if you just want to find a complete, free-standing, working piece of example code. To facilitate example searches, we have tagged every code example page with the keyword codex for code example which shouldn’t appear anywhere else on this site except in the FAQ. So if you want to search for an example that uses an ellipse, search for codex ellipse.

61.5 Cite Matplotlib

If you want to refer to Matplotlib in a publication, you can use “Matplotlib: A 2D Graphics Environment” by J. D. Hunter In Computing in Science & Engineering, Vol. 9, No. 3. (2007), pp. 90-95 (see this reference page):

```bibtex
@article{Hunter:2007,
    Address = {10662 LOS VAQUEROS CIRCLE, PO BOX 3014, LOS ALAMITOS, CA 90720-1314 USA},
    Author = {Hunter, John D.},
    Date-Added = {2010-09-23 12:22:10 -0700},
    Date-Modified = {2010-09-23 12:22:10 -0700},
    Isi = {000245668100019},
    Isi-Recid = {155389429},
    Journal = {Computing In Science \& Engineering},
    Month = {May-Jun},
    Number = {3},
    Pages = {90--95},
    Publisher = {IEEE COMPUTER SOC},
    Times-Cited = {21},
    Title = {Matplotlib: A 2D graphics environment},
    Type = {Editorial Material},
    Volume = {9},
    Year = {2007},
    Abstract = {Matplotlib is a 2D graphics package used for Python for application
development, interactive scripting, and publication-quality image
generation across user interfaces and operating systems.},
    Bdsk-Url-1 = {http://gateway.isiknowledge.com/gateway/Gateway.cgi?GWVersion=2&
        SrcApp=Alerting&DestApp=WOS&DestLinkType=FullRecord;_KeyUT=000245668100019}}
```
62.1 Obtaining matplotlib version

To find out your matplotlib version number, import it and print the __version__ attribute:

```python
>>> import matplotlib
>>> matplotlib.__version__
'0.98.0'
```

62.2 matplotlib install location

You can find what directory matplotlib is installed in by importing it and printing the __file__ attribute:

```python
>>> import matplotlib
>>> matplotlib.__file__
'/home/jdhunter/dev/lib64/python2.5/site-packages/matplotlib/__init__.pyc'
```
62.3 matplotlib configuration and cache directory locations

Each user has a matplotlib configuration directory which may contain a matplotlibrc file. To locate your matplotlib/configuration directory, use matplotlib.get_configdir():

```python
>>> import matplotlib as mpl
>>> mpl.get_configdir()
'/home/darren/.config/matplotlib'
```

On unix-like systems, this directory is generally located in your HOME directory under the .config/ directory.

In addition, users have a cache directory. On unix-like systems, this is separate from the configuration directory by default. To locate your .cache/ directory, use matplotlib.get_cachedir():

```python
>>> import matplotlib as mpl
>>> mpl.get_cachedir()
'/home/darren/.cache/matplotlib'
```

On windows, both the config directory and the cache directory are the same and are in your Documents and Settings or Users directory by default:

```python
>>> import matplotlib as mpl
>>> mpl.get_configdir()
'C:\Documents and Settings\jdhunter\.matplotlib'
>>> mpl.get_cachedir()
'C:\Documents and Settings\jdhunter\.matplotlib'
```

If you would like to use a different configuration directory, you can do so by specifying the location in your MPLCONFIGDIR environment variable – see Setting environment variables in Linux and OS-X. Note that MPLCONFIGDIR sets the location of both the configuration directory and the cache directory.

62.4 Getting help

There are a number of good resources for getting help with matplotlib. There is a good chance your question has already been asked:

- The mailing list archive.
- Github issues.
- Stackoverflow questions tagged matplotlib.

If you are unable to find an answer to your question through search, please provide the following information in your e-mail to the mailing list:

- your operating system; (Linux/UNIX users: post the output of `uname -a`)
- matplotlib version:
Matplotlib, Release 2.0.2

```python
python -c 'import matplotlib; print matplotlib.__version__'
```

- where you obtained matplotlib (e.g., your Linux distribution’s packages, github, PyPi, or Anaconda or Enthought Canopy).
- any customizations to your matplotlibrc file (see Customizing matplotlib).
- if the problem is reproducible, please try to provide a minimal, standalone Python script that demonstrates the problem. This is the critical step. If you can’t post a piece of code that we can run and reproduce your error, the chances of getting help are significantly diminished. Very often, the mere act of trying to minimize your code to the smallest bit that produces the error will help you find a bug in your code that is causing the problem.
- you can get very helpful debugging output from matplotlib by running your script with a verbose-helpful or --verbose-debug flags and posting the verbose output the lists:

```bash
> python simple_plot.py --verbose-helpful > output.txt
```

If you compiled matplotlib yourself, please also provide
- any changes you have made to setup.py or setupext.py
- the output of:

```bash
rm -rf build
python setup.py build
```

The beginning of the build output contains lots of details about your platform that are useful for the matplotlib developers to diagnose your problem.

- your compiler version – e.g., gcc --version

Including this information in your first e-mail to the mailing list will save a lot of time.

You will likely get a faster response writing to the mailing list than filing a bug in the bug tracker. Most developers check the bug tracker only periodically. If your problem has been determined to be a bug and can not be quickly solved, you may be asked to file a bug in the tracker so the issue doesn’t get lost.

### 62.5 Problems with recent git versions

First make sure you have a clean build and install (see How to completely remove matplotlib), get the latest git update, install it and run a simple test script in debug mode:

```bash
rm -rf build
rm -rf /path/to/site-packages/matplotlib*
git pull
python setup.py install > build.out
python examples/pylab_examples/simple_plot.py --verbose-debug > run.out
```

and post build.out and run.out to the matplotlib-devel mailing list (please do not post git problems to the users list).
Of course, you will want to clearly describe your problem, what you are expecting and what you are getting, but often a clean build and install will help. See also *Getting help*. 
CHAPTER SIXTYTHREE

ENVIRONMENT VARIABLES

Contents

- Environment Variables
  - Setting environment variables in Linux and OS-X
    * BASH/KSH
    * CSH/TCSH
  - Setting environment variables in windows

HOME

The user’s home directory. On Linux, ~ is shorthand for HOME.

PATH

The list of directories searched to find executable programs

PYTHONPATH

The list of directories that is added to Python’s standard search list when importing packages and modules

MPLCONFIGDIR

This is the directory used to store user customizations to matplotlib, as well as some caches to improve performance. If MPLCONFIGDIR is not defined, HOME/.matplotlib is used if it is writable. Otherwise, the Python standard library tempfile.gettempdir() is used to find a base directory in which the matplotlib subdirectory is created.

MPLBACKEND

This optional variable can be set to choose the matplotlib backend. Using the -d command line parameter or the use() function will override this value. See What is a backend?.

63.1 Setting environment variables in Linux and OS-X

To list the current value of PYTHONPATH, which may be empty, try:
The procedure for setting environment variables in depends on what your default shell is. **BASH** seems to be the most common, but **CSH** is also common. You should be able to determine which by running at the command prompt:

```bash
echo $SHELL
```

### 63.1.1 BASH/KSH

To create a new environment variable:

```bash
export PYTHONPATH=~/Python
```

To prepend to an existing environment variable:

```bash
export PATH=~/bin:${PATH}
```

The search order may be important to you, do you want `~/bin` to be searched first or last? To append to an existing environment variable:

```bash
export PATH=${PATH}:~/bin
```

To make your changes available in the future, add the commands to your `~/.bashrc` file.

### 63.1.2 CSH/TCSH

To create a new environment variable:

```csh
setenv PYTHONPATH ~/Python
```

To prepend to an existing environment variable:

```csh
setenv PATH ~/bin:${PATH}
```

The search order may be important to you, do you want `~/bin` to be searched first or last? To append to an existing environment variable:

```csh
setenv PATH ${PATH}:~/bin
```

To make your changes available in the future, add the commands to your `~/.cshrc` file.

### 63.2 Setting environment variables in windows

Open the **Control Panel** (*Start* → *Control Panel*), start the **System** program. Click the **Advanced** tab and select the **Environment Variables** button. You can edit or add to the **User Variables**.
WORKING WITH MATPLOTLIB IN VIRTUAL ENVIRONMENTS

64.1 Introduction

When running matplotlib in a virtual environment you may discover a few issues. matplotlib itself has no issue with virtual environments. However, the GUI frameworks that matplotlib uses for interactive figures have some issues with virtual environments. Everything below assumes some familiarity with the Matplotlib backends as found in What is a backend?

If you only use the IPython/Jupyter Notebook's inline and notebook backends and non interactive backends you should not have any issues and can ignore everything below.

If you are using Matplotlib on OSX you may also want to consider the OSX framework FAQ.

64.2 GUI Frameworks

Interactive Matplotlib relies heavily on the interaction with external GUI frameworks.

Most GUI frameworks are not pip installable. This makes it tricky to install them within a virtual environment. This problem does not exist if you use Conda environments where you can install all Conda supported GUI frameworks directly into the environment. In regular virtualenv environment various workarounds exist. Some of these are given here:

- The TKAgg backend doesn’t require any external dependencies and is normally always available.
- The QT4 framework PySide is pip installable.
- PYQT5 is pip installable on Python 3.5.
Other frameworks are harder to install into a virtual environment. There are at least two possible ways to get access to these in a virtual environment.

One often suggested solution is to use the `--system-site-packages` option to `virtualenv` when creating an environment. This adds all system wide packages to the virtual environment. However, this breaks the isolation between the virtual environment and the system install. Among other issues it results in hard to debug problems with system packages shadowing the environment packages. If you use `virtualenvwrapper` this can be toggled with the `toggleglobalsitepackages` command.

Alternatively, you can manually symlink the GUI frameworks into the environment. I.e. to use PyQt5, you should symlink `PyQt5` and `sip` from your system site packages directory into the environment taking care that the environment and the systemwide install use the same python version.
65.1 Introduction

On OSX, two different types of Python Builds exist: a regular build and a framework build. In order to interact correctly with OSX through the native GUI frameworks you need a framework build of Python. At the time of writing the macosx and WXAgg backends require a framework build to function correctly. This can result in issues for a python installation not build as a framework and may also happen in virtual envs and when using (Ana)Conda. From Matplotlib 1.5 onwards the macosx backend checks that a framework build is available and fails if a non framework build is found. WX has a similar check build in.

Without this check a partially functional figure is created. Among the issues with it is that it is produced in the background and cannot be put in front of any other window. Several solutions and work arounds exist see below.
65.2 Short version

65.2.1 VirtualEnv

If you are on Python 3, use `venv` instead of `virtualenv`:

```python
python -m venv my-virtualenv
source my-virtualenv/bin/activate
```

Otherwise you will need one of the workarounds below.

65.2.2 Pyenv

If you are using pyenv and virtualenv you can enable your python version to be installed as a framework:

```bash
PYTHON_CONFIGURE_OPTS="--enable-framework" pyenv install x.x.x
```

65.2.3 Conda

The default python provided in (Ana)Conda is not a framework build. However, the Conda developers have made it easy to install a framework build in both the main environment and in Conda envs. To use this install `python.app` and use `pythonw` rather than `python`.

65.3 Long version

Unfortunately virtualenv creates a non framework build even if created from a framework build of Python. As documented above you can use venv as an alternative on Python 3.

The issue has been reported on the virtualenv bug tracker here and here.

Until this is fixed, one of the following workarounds can be used:

65.3.1 PYTHONHOME Function

The best known work around is to use the non virtualenv python along with the PYTHONHOME environment variable. This can be done by defining a function in your `.bashrc` using

```bash
function frameworkpython {
    if [[ ! -z "$VIRTUAL_ENV" ]]; then
        PYTHONHOME=$VIRTUAL_ENV /usr/local/bin/python "$@
    else
        /usr/local/bin/python "$@
    fi
}
```
This function can then be used in all of your virtualenvs without having to fix every single one of them.

With this in place you can run `frameworkpython` to get an interactive framework build within the virtualenv. To run a script you can do `frameworkpython test.py` where `test.py` is a script that requires a framework build. To run an interactive IPython session with the framework build within the virtual environment you can do `frameworkpython -m IPython

**PYTHONHOME and Jupyter**

This approach can be followed even if using Jupyter notebooks: you just need to setup a kernel with the suitable PYTHONHOME definition. The `jupyter-virtualenv-osx` script automates the creation of such a kernel.

**PYTHONHOME Script**

An alternative work around borrowed from the WX wiki, is to use the non virtualenv python along with the PYTHONHOME environment variable. This can be implemented in a script as below. To use this modify `PYVER` and `PATHTOPYTHON` and put the script in the virtualenv bin directory i.e. `PATHTOENV/bin/frameworkpython`

```bash
#!/bin/bash

# what real Python executable to use
PYVER=2.7
PATHTOPYTHON=/usr/local/bin/
PYTHON=${PATHTOPYTHON}python${PYVER}

# find the root of the virtualenv, it should be the parent of the dir this script is in
ENV=`$PYTHON -c "import os; print(os.path.abspath(os.path.join(os.path.dirname("$0"), '..')))"`

# now run Python with the virtualenv set as Python's HOME
export PYTHONHOME=$ENV
exec $PYTHON "$@
```

With this in place you can run `frameworkpython` as above but will need to add this script to every virtualenv

**PythonW Compiler**

In addition `virtualenv-pythonw-osx` provides an alternative workaround which may be used to solve the issue.
Part IV

Matplotlib AxesGrid Toolkit
The matplotlib AxesGrid toolkit is a collection of helper classes to ease displaying multiple images in matplotlib. While the aspect parameter in matplotlib adjust the position of the single axes, AxesGrid toolkit provides a framework to adjust the position of multiple axes according to their aspects.

Note: AxesGrid toolkit has been a part of matplotlib since v 0.99. Originally, the toolkit had a single namespace of axes_grid. In more recent version (since svn r8226), the toolkit has divided into two separate namespace (axes_grid1 and axisartist). While axes_grid namespace is maintained for the backward compatibility, use of axes_grid1 and axisartist is recommended.

Warning: axes_grid and axisartist (but not axes_grid1) uses a custom Axes class (derived from the mpl’s original Axes class). As a side effect, some commands (mostly tick-related) do not work. Use axes_grid1 to avoid this, or see how things are different in axes_grid and axisartist (LINK needed)
OVERVIEW OF AXESGRID TOOLKIT

66.1 What is AxesGrid toolkit?

The matplotlib AxesGrid toolkit is a collection of helper classes, mainly to ease displaying (multiple) images in matplotlib.

Note: AxesGrid toolkit has been a part of matplotlib since v 0.99. Originally, the toolkit had a single namespace of `axes_grid`. In more recent version (since svn r8226), the toolkit has divided into two separate namespace (`axes_grid1` and `axisartist`). While `axes_grid` namespace is maintained for the backward compatibility, use of `axes_grid1` and `axisartist` is recommended.

Warning: `axes_grid` and `axisartist` (but not `axes_grid1`) uses a custom Axes class (derived from the mpl’s original Axes class). As a side effect, some commands (mostly tick-related) do not work. Use `axes_grid1` to avoid this, or see how things are different in `axes_grid` and `axisartist` (LINK needed)

AxesGrid toolkit has two namespaces (`axes_grid1` and `axisartist`). `axisartist` contains custom Axes class that is meant to support for curvilinear grids (e.g., the world coordinate system in astronomy). Unlike mpl’s original Axes class which uses Axes.xaxis and Axes.yaxis to draw ticks, ticklines and etc., Axes in axisartist uses special artist (AxisArtist) which can handle tick, ticklines and etc. for curved coordinate systems.
Since it uses a special artists, some mpl commands that work on Axes.xaxis and Axes.yaxis may not work. See LINK for more detail.

`axes_grid1` is a collection of helper classes to ease displaying (multiple) images with matplotlib. In matplotlib, the axes location (and size) is specified in the normalized figure coordinates, which may not be ideal for displaying images that needs to have a given aspect ratio. For example, it helps you to have a colorbar whose height always matches that of the image. `ImageGrid`, `RGB Axes` and `AxesDivider` are helper classes that deals with adjusting the location of (multiple) Axes. They provides a framework to adjust the position of multiple axes at the drawing time. `ParasiteAxes` provides twinx(or twiny)-like features so that you can plot different data (e.g., different y-scale) in a same Axes. `AnchoredArtists` includes custom artists which are placed at some anchored position, like the legend.
66.2 AXES_GRID1

66.2.1 ImageGrid

A class that creates a grid of Axes. In matplotlib, the axes location (and size) is specified in the normalized figure coordinates. This may not be ideal for images that needs to be displayed with a given aspect ratio. For example, displaying images of a same size with some fixed padding between them cannot be easily done in matplotlib. ImageGrid is used in such case.

```python
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import ImageGrid
import numpy as np

im = np.arange(100)
im.shape = 10, 10

fig = plt.figure(1, (4., 4.))
grid = ImageGrid(fig, 111,
                 # similar to subplot(111)
                 nrows_ncols=(2, 2),        # creates 2x2 grid of axes
                 axes_pad=0.1,             # pad between axes in inch.
                 )

for i in range(4):
    grid[i].imshow(im)    # The AxesGrid object work as a list of axes.

plt.show()
```

![ImageGrid Example](image-url)
• The position of each axes is determined at the drawing time (see \texttt{AxesDivider}), so that the size of the entire grid fits in the given rectangle (like the aspect of axes). Note that in this example, the paddings between axes are fixed even if you changes the figure size.

• axes in the same column has a same axes width (in figure coordinate), and similarly, axes in the same row has a same height. The widths (height) of the axes in the same row (column) are scaled according to their view limits (xlim or ylim).

```python
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import ImageGrid

def get_demo_image():
    import numpy as np
    from matplotlib.cbook import get_sample_data
    f = get_sample_data("axes_grid/bivariate_normal.npy", asfileobj=False)
    z = np.load(f)
    # z is a numpy array of 15x15
    return z, (-3, 4, -4, 3)

F = plt.figure(1, (5.5, 3.5))
grid = ImageGrid(F, 111,  # similar to subplot(111)
nrows_ncols=(1, 3),
axes_pad=0.1,
add_all=True,
label_mode="L",
)

Z, extent = get_demo_image()  # demo image

im1 = Z
im2 = Z[:, :10]
im3 = Z[:, 10:]
vmin, vmax = Z.min(), Z.max()
for i, im in enumerate([im1, im2, im3]):
    ax = grid[i]
    ax.imshow(im, origin="lower", vmin=vmin,
              vmax=vmax, interpolation="nearest")

plt.draw()
plt.show()
```
- xaxis are shared among axes in a same column. Similarly, yaxis are shared among axes in a same row. Therefore, changing axis properties (view limits, tick location, etc. either by plot commands or using your mouse in interactive backends) of one axes will affect all other shared axes.

When initialized, ImageGrid creates given number (ngrids or ncols * nrows if ngrids is None) of Axes instances. A sequence-like interface is provided to access the individual Axes instances (e.g., grid[0] is the first Axes in the grid. See below for the order of axes).

AxesGrid takes following arguments,

<table>
<thead>
<tr>
<th>Name</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fig</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rect</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nrows_ncols</td>
<td></td>
<td>number of rows and cols. e.g., (2,2)</td>
</tr>
<tr>
<td>ngrids</td>
<td>None</td>
<td>number of grids. nrows x ncols if None</td>
</tr>
<tr>
<td>direction</td>
<td>“row”</td>
<td>increasing direction of axes number. [row</td>
</tr>
<tr>
<td>axes_pad</td>
<td>0.02</td>
<td>pad between axes in inches</td>
</tr>
<tr>
<td>add_all</td>
<td>True</td>
<td>Add axes to figures if True</td>
</tr>
<tr>
<td>share_all</td>
<td>False</td>
<td>xaxis &amp; yaxis of all axes are shared if True</td>
</tr>
<tr>
<td>aspect</td>
<td>True</td>
<td>aspect of axes</td>
</tr>
<tr>
<td>label_mode</td>
<td>“L”</td>
<td>location of tick labels thaw will be displayed. “1” (only the lower left axes), “L” (left most and bottom most axes), or “all”.</td>
</tr>
<tr>
<td>cbar_mode</td>
<td>None</td>
<td>[None</td>
</tr>
<tr>
<td>cbar_location</td>
<td>right</td>
<td>[right</td>
</tr>
<tr>
<td>cbar_pad</td>
<td>None</td>
<td>pad between image axes and colorbar axes</td>
</tr>
<tr>
<td>cbar_size</td>
<td>“5%”</td>
<td>size of the colorbar</td>
</tr>
<tr>
<td>axes_class</td>
<td>None</td>
<td></td>
</tr>
</tbody>
</table>
**rect** specifies the location of the grid. You can either specify coordinates of the rectangle to be used (e.g., (0.1, 0.1, 0.8, 0.8) as in the Axes), or the subplot-like position (e.g., “121”).

**direction** means the increasing direction of the axes number.

**aspect** By default (False), widths and heights of axes in the grid are scaled independently. If True, they are scaled according to their data limits (similar to aspect parameter in mpl).

**share_all** if True, xaxis and yaxis of all axes are shared.

**direction** direction of increasing axes number. For “row”,

```
grid[0]  grid[1]
```

For “column”,

```
grid[0]  grid[2]
```

You can also create a colorbar (or colorbars). You can have colorbar for each axes (cbar_mode=”each”), or you can have a single colorbar for the grid (cbar_mode=”single”). The colorbar can be placed on your right, or top. The axes for each colorbar is stored as a cbar_axes attribute.

The examples below show what you can do with AxesGrid.

![Example AxesGrid](image)

### 66.2.2 AxesDivider

Behind the scene, the ImageGrid class and the RGBAxes class utilize the AxesDivider class, whose role is to calculate the location of the axes at drawing time. While a more about the AxesDivider is (will be) explained in (yet to be written) AxesDividerGuide, direct use of the AxesDivider class will not be necessary for most users. The axes_divider module provides a helper function make_axes_locatable, which can be useful. It takes a existing axes instance and create a divider for it.

```
ax = subplot(1,1,1)
divider = make_axes_locatable(ax)
```

`make_axes_locatable` returns an instance of the AxesLocator class, derived from the Locator. It provides `append_axes` method that creates a new axes on the given side of (“top”, “right”, “bottom” and “left”) of the original axes.
66.2.3 colorbar whose height (or width) in sync with the master axes

```python
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import make_axes_locatable
import numpy as np

ax = plt.subplot(111)
im = ax.imshow(np.arange(100).reshape((10,10)));

# create an axes on the right side of ax. The width of cax will be 5%
# of ax and the padding between cax and ax will be fixed at 0.05 inch.
divider = make_axes_locatable(ax)
cax = divider.append_axes("right", size="5%%", pad=0.05)

plt.colorbar(im, cax=cax)
```

scatter_hist.py with AxesDivider

The “scatter_hist.py” example in mpl can be rewritten using `make_axes_locatable`.

```python
axScatter = subplot(111)
axScatter.scatter(x, y)
axScatter.set_aspect(1.)
```
The scatter_hist using the AxesDivider has some advantage over the original scatter_hist.py in mpl. For example, you can set the aspect ratio of the scatter plot, even with the x-axis or y-axis is shared accordingly.
66.2.4 ParasiteAxes

The ParasiteAxes is an axes whose location is identical to its host axes. The location is adjusted in the
drawing time, thus it works even if the host change its location (e.g., images).

In most cases, you first create a host axes, which provides a few method that can be used to create parasite
axes. They are twinx, twiny (which are similar to twinx and twiny in the matplotlib) and twin. twin takes
an arbitrary transformation that maps between the data coordinates of the host axes and the parasite axes.
draw method of the parasite axes are never called. Instead, host axes collects artists in parasite axes and
draw them as if they belong to the host axes, i.e., artists in parasite axes are merged to those of the host axes
and then drawn according to their zorder. The host and parasite axes modifies some of the axes behavior.
For example, color cycle for plot lines are shared between host and parasites. Also, the legend command in
host, creates a legend that includes lines in the parasite axes. To create a host axes, you may use host_subplot
or host_axes command.

Example 1. twinx

```python
from mpl_toolkits.axes_grid1 import host_subplot
import matplotlib.pyplot as plt

host = host_subplot(111)
par = host.twinx()

host.set_xlabel("Distance")
host.set_ylabel("Density")
par.set_ylabel("Temperature")

p1, = host.plot([0, 1, 2], [0, 1, 2], label="Density")
p2, = par.plot([0, 1, 2], [0, 3, 2], label="Temperature")

leg = plt.legend()

host.yaxis.get_label().set_color(p1.get_color())
leg.texts[0].set_color(p1.get_color())

par.yaxis.get_label().set_color(p2.get_color())
leg.texts[1].set_color(p2.get_color())

plt.show()
```
Example 2. twin

twin without a transform argument treat the parasite axes to have a same data transform as the host. This can be useful when you want the top(or right)-axis to have different tick-locations, tick-labels, or tick-formatter for bottom(or left)-axis.

```python
ax2 = ax.twin()  # now, ax2 is responsible for "top" axis and "right" axis
ax2.set_xticks([0., .5*np.pi, np.pi, 1.5*np.pi, 2*np.pi])
ax2.set_xticklabels(["0", r"\frac{1}{2}\pi", r"\pi", r"\frac{3}{2}\pi", r"2\pi"])
```
A more sophisticated example using twin. Note that if you change the x-limit in the host axes, the x-limit of the parasite axes will change accordingly.
### 66.2.5 AnchoredArtists

It's a collection of artists whose location is anchored to the (axes) bbox, like the legend. It is derived from `OffsetBox` in mpl, and artist need to be drawn in the canvas coordinate. But, there is a limited support for an arbitrary transform. For example, the ellipse in the example below will have width and height in the data coordinate.

```python
import matplotlib.pyplot as plt

def draw_text(ax):
    from matplotlib.offsetbox import AnchoredText
    at = AnchoredText("Figure 1a",
                      loc=2, prop=dict(size=8), frameon=True,
                      )
    at.patch.set_boxstyle("round,pad=0.,rounding_size=0.2")
    ax.add_artist(at)

    at2 = AnchoredText("Figure 1(b)",
                       loc=3, prop=dict(size=8), frameon=True,
                       bbox_to_anchor=(0., 1.),
                       bbox_transform=ax.transAxes,
                       )
    at2.patch.set_boxstyle("round,pad=0.,rounding_size=0.2")
```
```python
ax.add_artist(at2)

def draw_circle(ax):
    # circle in the canvas coordinate
    from mpl_toolkits.axes_grid1.anchored_artists import AnchoredDrawingArea
    from matplotlib.patches import Circle
    ada = AnchoredDrawingArea(20, 20, 0, 0,
                               loc=1, pad=0., frameon=False)
    p = Circle((10, 10), 10)
    ada.da.add_artist(p)
    ax.add_artist(ada)

def draw_ellipse(ax):
    # draw an ellipse of width=0.1, height=0.15 in the data coordinate
    from mpl_toolkits.axes_grid1.anchored_artists import AnchoredEllipse
    ae = AnchoredEllipse(ax.transData, width=0.1, height=0.15, angle=0.,
                         loc=3, pad=0.5, borderpad=0.4, frameon=True)
    ax.add_artist(ae)

def draw_sizebar(ax):
    from mpl_toolkits.axes_grid1.anchored_artists import AnchoredSizeBar
    asb = AnchoredSizeBar(ax.transData,
                          0.1,
                          r"1"^r"\prime$",
                          loc=8,
                          pad=0.1, borderpad=0.5, sep=5,
                          frameon=False)
    ax.add_artist(asb)

if 1:
    ax = plt.gca()
    ax.set_aspect(1.)
    draw_text(ax)
    draw_circle(ax)
    draw_ellipse(ax)
    draw_sizebar(ax)
    plt.show()
```
66.2.6 InsetLocator

mpl_toolkits.axes_grid.inset_locator provides helper classes and functions to place your (inset) axes at the anchored position of the parent axes, similarly to AnchoredArtist.

Using mpl_toolkits.axes_grid.inset_locator.inset_axes(), you can have inset axes whose size is either fixed, or a fixed proportion of the parent axes. For example:

```python
inset_axes = inset_axes(parent_axes,
                        width="30%", # width = 30% of parent_bbox
                        height=1., # height : 1 inch
                        loc=3)
```

creates an inset axes whose width is 30% of the parent axes and whose height is fixed at 1 inch.

You may creates your inset whose size is determined so that the data scale of the inset axes to be that of the parent axes multiplied by some factor. For example,

```python
inset_axes = zoomed_inset_axes(ax,
                                0.5, # zoom = 0.5
                                loc=1)
```

creates an inset axes whose data scale is half of the parent axes. Here is complete examples.
For example, `zoomed_inset_axes()` can be used when you want the inset represents the zoom-up of the small portion in the parent axes. And `mpl_toolkits/axes_grid/inset_locator` provides a helper function `mark_inset()` to mark the location of the area represented by the inset axes.

```python
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1.inset_locator import zoomed_inset_axes
from mpl_toolkits.axes_grid1.inset_locator import mark_inset
import numpy as np

def get_demo_image():
    from matplotlib.cbook import get_sample_data
    import numpy as np
    f = get_sample_data("axes_grid/bivariate_normal.npy", asfileobj=False)
    z = np.load(f)
    # z is a numpy array of 15x15
    return z, (-3, 4, -4, 3)

fig, ax = plt.subplots(figsize=[5, 4])
# prepare the demo image
Z, extent = get_demo_image()
Z2 = np.zeros([150, 150], dtype="d")
ny, nx = Z.shape
Z2[30:30 + ny, 30:30 + nx] = Z
# extent = [-3, 4, -4, 3]
ax.imshow(Z2, extent=extent, interpolation="nearest", origin="lower")

axins = zoomed_inset_axes(ax, 6, loc=1)  # zoom = 6
axins.imshow(Z2, extent=extent, interpolation="nearest", origin="lower")
```
RGB Axes

RGBAxes is a helper class to conveniently show RGB composite images. Like ImageGrid, the location of axes are adjusted so that the area occupied by them fits in a given rectangle. Also, the xaxis and yaxis of each axes are shared.
from mpl_toolkits.axes_grid1.axes_rgb import RGBAxes

fig = plt.figure(1)
ax = RGBAxes(fig, [0.1, 0.1, 0.8, 0.8])

r, g, b = get_rgb()  # r,g,b are 2-d images
ax.imshow_rgb(r, g, b,
             origin="lower", interpolation="nearest")

66.3 AXISARTIST

66.3.1 AxisArtist

AxisArtist module provides a custom (and very experimental) Axes class, where each axis (left, right, top and bottom) have a separate artist associated which is responsible to draw axis-line, ticks, ticklabels, label. Also, you can create your own axis, which can pass through a fixed position in the axes coordinate, or a fixed position in the data coordinate (i.e., the axis floats around when viewlimit changes).

The axes class, by default, have its xaxis and yaxis invisible, and has 4 additional artists which are responsible to draw axis in “left”, “right”, “bottom” and “top”. They are accessed as ax.axis["left"], ax.axis["right"], and so on, i.e., ax.axis is a dictionary that contains artists (note that ax.axis is still a callable methods and it behaves as an original Axes.axis method in mpl).
To create an axes,

```python
import mpl_toolkits.axisartist as AA
fig = plt.figure(1)
ax = AA.Axes(fig, [0.1, 0.1, 0.8, 0.8])
fig.add_axes(ax)
```
or to create a subplot

```python
ax = AA.Subplot(fig, 111)
fig.add_subplot(ax)
```

For example, you can hide the right, and top axis by

```python
ax.axis["right"].set_visible(False)
ax.axis["top"].set_visible(False)
```

It is also possible to add an extra axis. For example, you may have a horizontal axis at y=0 (in data coordinate).

```python
ax.axis["y=0"] = ax.new_floating_axis(nth_coord=0, value=0)
```

```python
import matplotlib.pyplot as plt
import mpl_toolkits.axisartist as AA
fig = plt.figure(1)
fig.subplots_adjust(right=0.85)
ax = AA.Subplot(fig, 1, 1, 1)
fig.add_subplot(ax)
# make some axis invisible
ax.axis["bottom", "top", "right"].set_visible(False)
# make an new axis along the first axis (x-axis) which pass
```
# through y=0.
ax.axis["y=0"] = ax.new_floating_axis(nth_coord=0, value=0,
    axis_direction="bottom")
ax.axis["y=0"].toggle(all=True)
ax.axis["y=0"].label.set_text("y = 0")
ax.set_yscale(-2, 4)
plt.show()

Or a fixed axis with some offset

# make new (right-side) yaxis, but wth some offset
ax.axis["right2"] = ax.new_fixed_axis(loc="right",
    offset=(20, 0))

**AxisArtist with ParasiteAxes**

Most commands in the axes_grid1 toolkit can take a axes_class keyword argument, and the commands creates an axes of the given class. For example, to create a host subplot with axisartist.Axes,

```python
import mpl_toolkits.axisartist as AA
from mpl_toolkits.axes_grid1 import host_subplot
```
host = host_subplot(111, axes_class=AA.Axes)

Here is an example that uses parasiteAxes.

66.3.2 Curvilinear Grid

The motivation behind the AxisArtist module is to support curvilinear grid and ticks.
See `AXISARTIST namespace` for more details.

### 66.3.3 Floating Axes

This also support a Floating Axes whose outer axis are defined as floating axis.
67.1 AxesDivider

The axes_divider module provides helper classes to adjust the axes positions of a set of images in the drawing time.

- `axes_size` provides a class of units that the size of each axes will be determined. For example, you can specify a fixed size.
- `Divider` this is the class that is used to calculate the axes position. It divides the given rectangular area into several areas. You initialize the divider by setting the horizontal and vertical list of sizes that the division will be based on. You then use the `new_locator` method, whose return value is a callable object that can be used to set the `axes_locator` of the axes.

You first initialize the divider by specifying its grids, i.e., horizontal and vertical. For example:

```python
rect = [0.2, 0.2, 0.6, 0.6]
horiz=[h0, h1, h2, h3]
vert=[v0, v1, v2]
divider = Divider(fig, rect, horiz, vert)
```

where, `rect` is a bounds of the box that will be divided and `h0...h3, v0...v2` need to be an instance of classes in the `axes_size`. They have `get_size` method that returns a tuple of two floats. The first float is the relative size, and the second float is the absolute size. Consider a following grid:

```
  v0  
v1   
h0,v2| h1 | h2 | h3
```

- `v0` => 0, 2
- `v1` => 2, 0
- `v2` => 3, 0
The height of the bottom row is always 2 (axes_divider internally assumes that the unit is inch). The first and the second rows with height ratio of 2:3. For example, if the total height of the grid 6, then the first and second row will each occupy 2/(2+3) and 3/(2+3) of (6-1) inches. The widths of columns (horiz) will be similarly determined. When aspect ratio is set, the total height (or width) will be adjusted accordingly.

The mpl_toolkits.axes_grid.axes_size contains several classes that can be used to set the horizontal and vertical configurations. For example, for the vertical configuration above will be:

```python
from mpl_toolkits.axes_grid.axes_size import Fixed, Scaled
vert = [Fixed(2), Scaled(2), Scaled(3)]
```

After you set up the divider object, then you create a locator instance which will be given to the axes:

```python
locator = divider.new_locator(nx=0, ny=1)
ax.set_axes_locator(locator)
```

The return value of the new_locator method is a instance of the AxesLocator class. It is a callable object that returns the location and size of the cell at the first column and the second row. You may create a locator that spans over multiple cells:

```python
locator = divider.new_locator(nx=0, nx=2, ny=1)
```

The above locator, when called, will return the position and size of the cells spanning the first and second column and the first row. You may consider it as [0:2, 1].

See the example,

```python
import mpl_toolkits.axes_grid.axes_size as Size
from mpl_toolkits.axes_grid import Divider
import matplotlib.pyplot as plt

fig1 = plt.figure(1, (5.5, 4.))

# the rect parameter will be ignore as we will set axes_locator
rect = (0.1, 0.1, 0.8, 0.8)
ax = [fig1.add_axes(rect, label="%d"%i) for i in range(4)]

horiz = [Size.Scaled(1.5), Size.Fixed(.5), Size.Scaled(1.),
         Size.Scaled(.5)]

vert = [Size.Scaled(1.), Size.Fixed(.5), Size.Scaled(1.5)]

# divide the axes rectangle into grid whose size is specified by horiz * vert
divider = Divider(fig1, rect, horiz, vert, aspect=False)

ax[0].set_axes_locator(divider.new_locator(nx=0, ny=0))
ax[1].set_axes_locator(divider.new_locator(nx=0, ny=2))
ax[2].set_axes_locator(divider.new_locator(nx=2, ny=2))
ax[3].set_axes_locator(divider.new_locator(nx=2, nx1=4, ny=0))

for ax1 in ax:
    plt.setp(ax1.get_xticklabels()+ax1.get_yticklabels(),
             visible=False)
```
You can adjust the size of the each axes according to their x or y data limits (AxesX and AxesY), similar to the axes aspect parameter.

```python
import mpl_toolkits.axes_grid.axes_size as Size
from mpl_toolkits.axes_grid import Divider
import matplotlib.pyplot as plt

fig1 = plt.figure(1, (5.5, 4))

# the rect parameter will be ignore as we will set axes locator
rect = (0.1, 0.1, 0.8, 0.8)
ax = [fig1.add_axes(rect, label="%d")%i) for i in range(4)]

horiz = [Size.AxesX(ax[0]), Size.Fixed(.5), Size.AxesX(ax[1])]
vert = [Size.AxesY(ax[0]), Size.Fixed(.5), Size.AxesY(ax[2])]

# divide the axes rectangle into grid whose size is specified by horiz * vert
divider = Divider(fig1, rect, horiz, vert, aspect=False)

ax[0].set_axes_locator(divider.new_locator(nx=0, ny=0))
ax[1].set_axes_locator(divider.new_locator(nx=2, ny=0))
```
67.2 AXISARTIST namespace

The AxisArtist namespace includes a derived Axes implementation. The biggest difference is that the artists responsible to draw axis line, ticks, ticklabel and axis labels are separated out from the mpl’s Axis class, which are much more than artists in the original mpl. This change was strongly motivated to support curvilinear grid. Here are a few things that mpl_toolkits.axisartist.Axes is different from original Axes from mpl.
• Axis elements (axis line(spine), ticks, ticklabel and axis labels) are drawn by a AxisArtist instance. Unlike Axis, left, right, top and bottom axis are drawn by separate artists. And each of them may have different tick location and different tick labels.

• gridlines are drawn by a Gridlines instance. The change was motivated that in curvilinear coordinate, a gridline may not cross axis-lines (i.e., no associated ticks). In the original Axes class, gridlines are tied to ticks.

• ticklines can be rotated if necessary (i.e, along the gridlines)

In summary, all these changes was to support

• a curvilinear grid.

• a floating axis

\[ r = 6 \]
\[ \theta = 60^\circ \]
\[ 0 \]
\[ -30^\circ \]
\[ -60^\circ \]
\[ -90^\circ \]
\[ -120^\circ \]

\[ 0 \]
\[ 6 \]
\[ 8 \]
\[ 10 \]

\[ \theta = 0 \]
\[ \theta = 30^\circ \]
\[ \theta = 60^\circ \]

mpl_toolkits.axisartist.Axes class defines a axis attribute, which is a dictionary of AxisArtist instances. By default, the dictionary has 4 AxisArtist instances, responsible for drawing of left, right, bottom and top axis.

xaxis and yaxis attributes are still available, however they are set to not visible. As separate artists are used for rendering axis, some axis-related method in mpl may have no effect. In addition to AxisArtist instances, the mpl_toolkits.axisartist.Axes will have gridlines attribute (Gridlines), which obviously draws grid lines.

In both AxisArtist and Gridlines, the calculation of tick and grid location is delegated to an instance of
GridHelper class. mpl_toolkits.axisartist.Axes class uses GridHelperRectlinear as a grid helper. The GridHelperRectlinear class is a wrapper around the xaxis and yaxis of mpl’s original Axes, and it was meant to work as the way how mpl’s original axes works. For example, tick location changes using set_ticks method and etc. should work as expected. But change in artist properties (e.g., color) will not work in general, although some effort has been made so that some often-change attributes (color, etc.) are respected.

67.2.1 AxisArtist

AxisArtist can be considered as a container artist with following attributes which will draw ticks, labels, etc.

- line
- major_ticks, major_ticklabels
- minor_ticks, minor_ticklabels
- offsetText
- label

line

Derived from Line2d class. Responsible for drawing a spinal(?) line.

major_ticks, minor_ticks

Derived from Line2d class. Note that ticks are markers.

major_ticklabels, minor_ticklabels

Derived from Text. Note that it is not a list of Text artist, but a single artist (similar to a collection).

axislabel

Derived from Text.

Default AxisArtists

By default, following for axis artists are defined:

```
ax.axis["left"], ax.axis["bottom"], ax.axis["right"], ax.axis["top"]
```

The ticklabels and axislabel of the top and the right axis are set to not visible.

For example, if you want to change the color attributes of major_ticklabels of the bottom x-axis
ax.axis["bottom"].major_ticklabels.set_color("b")

Similarly, to make ticklabels invisible

ax.axis["bottom"].major_ticklabels.set_visible(False)

AxisArtist provides a helper method to control the visibility of ticks, ticklabels, and label. To make ticklabel invisible,

ax.axis["bottom"].toggle(ticklabels=False)

To make all of ticks, ticklabels, and (axis) label invisible

ax.axis["bottom"].toggle(all=False)

To turn all off but ticks on

ax.axis["bottom"].toggle(all=False, ticks=True)

To turn all on but (axis) label off

ax.axis["bottom"].toggle(all=True, label=False)

ax.axis’s __getitem__ method can take multiple axis names. For example, to turn ticklabels of “top” and “right” axis on,

ax.axis["top","right"].toggle(ticklabels=True)

Note that ‘ax.axis["top","right"]’ returns a simple proxy object that translate above code to something like below.

for n in ["top","right"]: 
    ax.axis[n].toggle(ticklabels=True)

So, any return values in the for loop are ignored. And you should not use it anything more than a simple method.

Like the list indexing ":" means all items, i.e.,

ax.axis[:].major_ticks.set_color("r")

changes tick color in all axis.

**67.2.2 HowTo**

1. Changing tick locations and label.
   
   Same as the original mpl’s axes.
2. Changing axis properties like color, etc.

Change the properties of appropriate artists. For example, to change the color of the ticklabels:

```python
ax.axis["left"].major_ticklabels.set_color("r")
```

3. To change the attributes of multiple axis:

```python
ax.axis["left","bottom"].major_ticklabels.set_color("r")
```

or to change the attributes of all axis:

```python
ax.axis[:].major_ticklabels.set_color("r")
```

4. **To change the tick size (length), you need to use** `axis.major_ticks.set_ticksize` method. To change the direction of the ticks (ticks are in opposite direction of ticklabels by default), use `axis.major_ticks.set_tick_out` method.

   To change the pad between ticks and ticklabels, use `axis.major_ticklabels.set_pad` method.

   To change the pad between ticklabels and axis label, `axis.label.set_pad` method.

### 67.2.3 Rotation and Alignment of TickLabels

This is also quite different from the original mpl and can be confusing. When you want to rotate the ticklabels, first consider using "set_axis_direction" method.

```python
ax1.axis["left"].major_ticklabels.set_axis_direction("top")
ax1.axis["right"].label.set_axis_direction("left")
```

The parameter for `set_axis_direction` is one of ["left", “right”, “bottom”, “top”].

You must understand some underlying concept of directions.
1. There is a reference direction which is defined as the direction of the axis line with increasing coordinate. For example, the reference direction of the left x-axis is from bottom to top.

![Reference direction diagram](image)

The direction, text angle, and alignments of the ticks, ticklabels and axis-label is determined with respect to the reference direction

2. `ticklabel_direction` is either the right-hand side (+) of the reference direction or the left-hand side (-).

![Tick label direction diagram](image)

3. same for the `label_direction`

![Label direction diagram](image)
4. ticks are by default drawn toward the opposite direction of the ticklabels.

5. text rotation of ticklabels and label is determined in reference to the `ticklabel_direction` or `label_direction`, respectively. The rotation of ticklabels and label is anchored.

---

On the other hand, there is a concept of “axis_direction”. This is a default setting of above properties for each, “bottom”, “left”, “top”, and “right” axis.

<table>
<thead>
<tr>
<th></th>
<th>left</th>
<th>bottom</th>
<th>right</th>
<th>top</th>
</tr>
</thead>
<tbody>
<tr>
<td>axislabel direction</td>
<td>‘-’</td>
<td>‘+’</td>
<td>‘+’</td>
<td>‘-’</td>
</tr>
<tr>
<td>axislabel rotation</td>
<td>180</td>
<td>0</td>
<td>0</td>
<td>180</td>
</tr>
<tr>
<td>axislabel va</td>
<td>center</td>
<td>top</td>
<td>center</td>
<td>bottom</td>
</tr>
<tr>
<td>axislabel ha</td>
<td>right</td>
<td>center</td>
<td>right</td>
<td>center</td>
</tr>
<tr>
<td>ticklabel direction</td>
<td>‘-’</td>
<td>‘+’</td>
<td>‘+’</td>
<td>‘-’</td>
</tr>
<tr>
<td>ticklabels rotation</td>
<td>90</td>
<td>0</td>
<td>-90</td>
<td>180</td>
</tr>
<tr>
<td>ticklabel ha</td>
<td>right</td>
<td>center</td>
<td>right</td>
<td>center</td>
</tr>
<tr>
<td>ticklabel va</td>
<td>center</td>
<td>baseline</td>
<td>center</td>
<td>baseline</td>
</tr>
</tbody>
</table>

And, `set_axis_direction("top")` means to adjust the text rotation etc, for settings suitable for “top” axis. The concept of axis direction can be more clear with curved axis.
The axis_direction can be adjusted in the AxisArtist level, or in the level of its child arists, i.e., ticks, ticklabels, and axis-label.

```python
ax1.axis['left'].set_axis_direction('top')
```
changes axis_direction of all the associated artist with the “left” axis, while

```python
ax1.axis['left'].major_ticklabels.set_axis_direction('top')
```
changes the axis_direction of only the major_ticklabels. Note that set_axis_direction in the AxisArtist level changes the ticklabel_direction and label_direction, while changing the axis_direction of ticks, ticklabels, and axis-label does not affect them.

If you want to make ticks outward and ticklabels inside the axes, use invert_ticklabel_direction method.

```python
ax.axis[:].invert_ticklabel_direction()
```

A related method is “set_tick_out”. It makes ticks outward (as a matter of fact, it makes ticks toward the opposite direction of the default direction).

```python
ax.axis[:].major_ticks.set_tick_out(True)
```
So, in summary,

- **AxisArtist’s methods**
  - set_axis_direction: “left”, “right”, “bottom”, or “top”
  - set_ticklabel_direction: “+” or “-“
  - set_axislabel_direction: “+” or “-“
  - invert_ticklabel_direction

- **Ticks’ methods (major_ticks and minor_ticks)**
  - set_tick_out: True or False
  - set_ticksize: size in points

- **TickLabels’ methods (major_ticklabels and minor_ticklabels)**
  - set_axis_direction: “left”, “right”, “bottom”, or “top”
  - set_rotation: angle with respect to the reference direction
  - set_ha and set_va: see below

- **AxisLabels’ methods (label)**
  - set_axis_direction: “left”, “right”, “bottom”, or “top”
  - set_rotation: angle with respect to the reference direction
  - set_ha and set_va

**Adjusting ticklabels alignment**

Alignment of TickLabels are treated specially. See below
Adjusting pad

To change the pad between ticks and ticklabels

```python
ax.axis["left"].major_ticklabels.set_pad(10)
```

Or ticklabels and axis-label

```python
ax.axis["left"].label.set_pad(10)
```
67.2.4 GridHelper

To actually define a curvilinear coordinate, you have to use your own grid helper. A generalised version of grid helper class is supplied and this class should suffice in most of cases. A user may provide two functions which define a transformation (and its inverse pair) from the curved coordinate to (rectilinear) image coordinate. Note that while ticks and grids are drawn for curved coordinate, the data transform of the axes itself (ax.transData) is still rectilinear (image) coordinate.

```python
from mpl_toolkits.axisartist.grid_helper_curvelinear import GridHelperCurveLinear
from mpl_toolkits.axisartist import Subplot

# from curved coordinate to rectlinear coordinate.
def tr(x, y):
    x, y = np.asarray(x), np.asarray(y)
    return x, y-x

# from rectlinear coordinate to curved coordinate.
def inv_tr(x,y):
    x, y = np.asarray(x), np.asarray(y)
    return x, y+x

grid_helper = GridHelperCurveLinear((tr, inv_tr))
ax1 = Subplot(fig, 1, 1, 1, grid_helper=grid_helper)
fig.add_subplot(ax1)
```

You may use matplotlib’s Transform instance instead (but a inverse transformation must be defined). Often, coordinate range in a curved coordinate system may have a limited range, or may have cycles. In those cases, a more customized version of grid helper is required.

```python
import mpl_toolkits.axisartist.angle_helper as angle_helper

# PolarAxes.PolarTransform takes radian. However, we want our coordinate
# system in degree
```
tr = Affine2D().scale(np.pi/180., 1.) + PolarAxes.PolarTransform()

# extreme finder : find a range of coordinate.
# 20, 20 : number of sampling points along x, y direction
# The first coordinate (longitude, but theta in polar)
# has a cycle of 360 degree.
# The second coordinate (latitude, but radius in polar) has a minimum of 0
extreme_finder = angle_helper.ExtremeFinderCycle(20, 20,
    lon_cycle = 360,
    lat_cycle = None,
    lon_minmax = None,
    lat_minmax = (0, np.inf),
)

# Find a grid values appropriate for the coordinate (degree,
# minute, second). The argument is a approximate number of grids.
grid_locator1 = angle_helper.LocatorDMS(12)

# And also uses an appropriate formatter. Note that, the
# acceptable Locator and Formatter class is a bit different than
# that of mpl's, and you cannot directly use mpl's Locator and
# Formatter here (but may be possible in the future).
tick_formatter1 = angle_helper.FormatterDMS()

grid_helper = GridHelperCurveLinear(tr,
    extreme_finder=extreme_finder,
    grid_locator1=grid_locator1,
    tick_formatter1=tick_formatter1)

Again, the transData of the axes is still a rectilinear coordinate (image coordinate). You may manually do conversion between two coordinates, or you may use Parasite Axes for convenience.:

ax1 = SubplotHost(fig, 1, 2, 2, grid_helper=grid_helper)

# A parasite axes with given transform
ax2 = ParasiteAxesAuxTrans(ax1, tr, "equal")
# note that ax2.transData == tr + ax1.transData
# Anything you draw in ax2 will match the ticks and grids of ax1.
ax1.parasites.append(ax2)
67.2.5 FloatingAxis

A floating axis is an axis one of whose data coordinate is fixed, i.e., its location is not fixed in Axes coordinate but changes as axes data limits changes. A floating axis can be created using `new_floating_axis` method. However, it is your responsibility that the resulting AxisArtist is properly added to the axes. A recommended way is to add it as an item of Axes's axis attribute:

```python
# floating axis whose first (index starts from 0) coordinate
# (theta) is fixed at 60
ax1.axis["lat"] = axis = ax1.new_floating_axis(0, 60)
axis.label.set_text(r"$\theta = 60^\circ$")
axis.label.set_visible(True)
```

See the first example of this page.

67.2.6 Current Limitations and TODO's

The code need more refinement. Here is a incomplete list of issues and TODO's

- No easy way to support a user customized tick location (for curvilinear grid). A new Locator class needs to be created.
- FloatingAxis may have coordinate limits, e.g., a floating axis of $x = 0$, but $y$ only spans from 0 to 1.
- The location of axislabel of FloatingAxis needs to be optionally given as a coordinate value. ex, a floating axis of $x=0$ with label at $y=1$
68.1 `mpl_toolkits.axes_grid.axes_size`

```python
class mpl_toolkits.axes_grid.axes_size.Fixed(fixed_size)
    Simple fixed size with absolute part = fixed_size and relative part = 0

class mpl_toolkits.axes_grid.axes_size.Scaled(scalable_size)
    Simple scaled size with absolute part = 0 and relative part = scalable_size

class mpl_toolkits.axes_grid.axes_size.AxesX(axes, aspect=1.0, ref_ax=None)
    Scaled size whose relative part corresponds to the data width of the axes multiplied by the aspect.

class mpl_toolkits.axes_grid.axes_size.AxesY(axes, aspect=1.0, ref_ax=None)
    Scaled size whose relative part corresponds to the data height of the axes multiplied by the aspect.

class mpl_toolkits.axes_grid.axes_size.MaxWidth(artist_list)
    Size whose absolute part is the largest width of the given artist_list.

class mpl_toolkits.axes_grid.axes_size.MaxHeight(artist_list)
    Size whose absolute part is the largest height of the given artist_list.

class mpl_toolkits.axes_grid.axes_size.Fraction(fraction, ref_size)
    An instance whose size is a fraction of the ref_size.
```

```python
>>> s = Fraction(0.3, AxesX(ax))
```

```python
class mpl_toolkits.axes_grid.axes_size.Padded(size, pad)
    Return a instance where the absolute part of size is increase by the amount of pad.
```

```python
mpl_toolkits.axes_grid.axes_size.from_any(size, fraction_ref=None)
    Creates Fixed unit when the first argument is a float, or a Fraction unit if that is a string that ends with %.
    The second argument is only meaningful when Fraction unit is created.
```

```python
>>> a = Size.from_any(1.2)  # => Size.Fixed(1.2)
>>> Size.from_any("50%", a)  # => Size.Fraction(0.5, a)
```
68.2 `mpl_toolkits.axes_grid.axes_divider`

class `mpl_toolkits.axes_grid.axes_divider.Divider(fig, pos, horizontal, vertical, aspect=None, anchor='C')`

This is the class that is used calculates the axes position. It divides the given rectangular area into several sub-rectangles. You initialize the divider by setting the horizontal and vertical lists of sizes (`mpl_toolkits.axes_grid.axes_size`) that the division will be based on. You then use the new_locator method to create a callable object that can be used as the axes_locator of the axes.

**Parameters**

- **fig**: Figure
- **pos**: tuple of 4 floats
  - position of the rectangle that will be divided
- **horizontal**: list of `axes_size`
  - sizes for horizontal division
- **vertical**: list of `axes_size`
  - sizes for vertical division
- **aspect**: bool
  - if True, the overall rectangular area is reduced so that the relative part of the horizontal and vertical scales have the same scale.
- **anchor**: {'C', 'SW', 'S', 'SE', 'E', 'NE', 'N', 'NW', 'W'}
  - placement of the reduced rectangle when `aspect` is True

**Methods**

- **add_auto_adjustable_area**(use_axes, pad=0.1, adjust_dirs=None)

- **append_size**(position, size)

- **get_anchor**()
  - return the anchor

- **get_aspect**()
  - return aspect

- **get_horizontal**()
  - return horizontal sizes

- **get_horizontal_sizes**(renderer)

- **get_locator**()

- **get_position**()
  - return the position of the rectangle.
**get_position_runtime**(*ax*, *renderer*)

**get_vertical()**

    return vertical sizes

**get_vertical_sizes**(*renderer*)

**get_vsize_hsize()**

**locate**(*nx*, *ny*, *nx1=None*, *ny1=None*, *axes=None*, *renderer=None*)

Parameters *nx*, *nx1* : int

    Integers specifying the column-position of the cell. When *nx1* is None, a
    single *nx*-th column is specified. Otherwise location of columns spanning
    between *nx* to *nx1* (but excluding *nx1*-th column) is specified.

*ny*, *ny1* : int

    Same as *nx* and *nx1*, but for row positions.

*axes*

*renderer*

**new_locator**(*nx*, *ny*, *nx1=None*, *ny1=None*)

    Returns a new locator (*mpl_toolkits.axes_grid.axes_divider.AxesLocator*) for spec-
    ified cell.

Parameters *nx*, *nx1* : int

    Integers specifying the column-position of the cell. When *nx1* is None, a
    single *nx*-th column is specified. Otherwise location of columns spanning
    between *nx* to *nx1* (but excluding *nx1*-th column) is specified.

*ny*, *ny1* : int

    Same as *nx* and *nx1*, but for row positions.

**set_anchor**(*anchor*)

Parameters *anchor* : {'C', 'SW', 'S', 'SE', 'E', 'NE', 'N', 'NW', 'W'}

    anchor position
<table>
<thead>
<tr>
<th>value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘C’</td>
<td>Center</td>
</tr>
<tr>
<td>‘SW’</td>
<td>bottom left</td>
</tr>
<tr>
<td>‘S’</td>
<td>bottom</td>
</tr>
<tr>
<td>‘SE’</td>
<td>bottom right</td>
</tr>
<tr>
<td>‘E’</td>
<td>right</td>
</tr>
<tr>
<td>‘NE’</td>
<td>top right</td>
</tr>
<tr>
<td>‘N’</td>
<td>top</td>
</tr>
<tr>
<td>‘NW’</td>
<td>top left</td>
</tr>
<tr>
<td>‘W’</td>
<td>left</td>
</tr>
</tbody>
</table>

`set_aspect(aspect=False)`

**Parameters** `aspect` : bool

`set_horizontal(h)`

**Parameters** `h` : list of `axes_size`

sizes for horizontal division

`set_locator(_locator)`

`set_position(pos)`

set the position of the rectangle.

**Parameters** `pos` : tuple of 4 floats

position of the rectangle that will be divided

`set_vertical(v)`

**Parameters** `v` : list of `axes_size`

sizes for vertical division

```python
class mpl_toolkits.axes_grid.axes_divider.AxesLocator(axes_divider, nx, ny, nx1=None, ny1=None)
```

A simple callable object, initialized with AxesDivider class, returns the position and size of the given cell.

**Parameters** `axes_divider` : AxesDivider

`nx, nx1` : int

Integers specifying the column-position of the cell. When `nx1` is None, a single `nx`-th column is specified. Otherwise location of columns spanning between `nx` to `nx1` (but excluding `nx1`-th column) is specified.

`ny, ny1` : int

Same as `nx` and `nx1`, but for row positions.
get_subplotspec()

class mpl_toolkits.axes_grid.axes_divider.SubplotDivider(fig, *args, **kwargs)
The Divider class whose rectangle area is specified as a subplot geometry.

Parameters fig : matplotlib.figure.Figure
    args : tuple (numRows, numCols, plotNum)
        The array of subplots in the figure has dimensions numRows, numCols, and
        plotNum is the number of the subplot being created. plotNum starts at 1 in the
        upper left corner and increases to the right.
        If numRows <= numCols <= plotNum < 10, args can be the decimal integer
        numRows * 100 + numCols * 10 + plotNum.

change_geometry(numrows, numcols, num)
    change subplot geometry, e.g., from 1,1,1 to 2,2,3

get_geometry()
    get the subplot geometry, e.g., 2,2,3

get_position()
    return the bounds of the subplot box

get_subplotspec()
    get the SubplotSpec instance

set_subplotspec(subplotspec)
    set the SubplotSpec instance

update_params()
    update the subplot position from fig.subplotpars

class mpl_toolkits.axes_grid.axes_divider.AxesDivider(axes, xref=None, yref=None)
Divider based on the pre-existing axes.

Parameters axes : Axes
    xref
    yref

append_axes(position, size, pad=None, add_to_figure=True, **kwargs)
    create an axes at the given position with the same height (or width) of the main axes.
    position ["left","right","bottom","top"]
    size and pad should be axes_grid.axes_size compatible.

new_horizontal(size, pad=None, pack_start=False, **kwargs)
    Add a new axes on the right (or left) side of the main axes.
    Parameters size : axes_size or float or string
        A width of the axes. If float or string is given, from_any function is used to
        create the size, with ref_size set to AxesX instance of the current axes.
pad : *axes_size* or float or string

   Pad between the axes. It takes same argument as *size*.

pack_start : bool

   If False, the new axes is appended at the end of the list, i.e., it became the
   right-most axes. If True, it is inserted at the start of the list, and becomes the
   left-most axes.

kw_args

   All extra keywords arguments are passed to the created axes. If *axes_class* is
given, the new axes will be created as an instance of the given class. Other-
wise, the same class of the main axes will be used.

    new_vertical(*size*, *pad*=None, *pack_start*=False, **kw_args)

   Add a new axes on the top (or bottom) side of the main axes.

   Parameters *size* : *axes_size* or float or string

      A height of the axes. If float or string is given, *from_any* function is used to
      create the size, with *ref_size* set to *AxesX* instance of the current axes.

    pad : *axes_size* or float or string

      Pad between the axes. It takes same argument as *size*.

    pack_start : bool

      If False, the new axes is appended at the end of the list, i.e., it became the
      right-most axes. If True, it is inserted at the start of the list, and becomes the
      left-most axes.

kw_args

   All extra keywords arguments are passed to the created axes. If *axes_class* is
given, the new axes will be created as an instance of the given class. Other-
wise, the same class of the main axes will be used.

68.3 mpl_toolkits.axes_grid.axes_grid Grid

   class mpl_toolkits.axes_grid.axes_grid.Grid(*fig*, *rect*, *nrows_ncols*, *ngrids*=None, *direction*='row', *axes_pad*=0.02, *add_all*=True,
       *share_all*=False, *share_x*=True, *share_y*=True, *label_mode*='L',
       *axes_class*=None)

   Build an Grid instance with a grid *nrows* *ncols* *Axes* in *Figure* *fig* with *rect*=[*left*, *bottom*, *width*,
   *height*] (in *Figure* coordinates) or the subplot position code (e.g., “121”).

   Optional keyword arguments:
Matplotlib, Release 2.0.2

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>direction</td>
<td>&quot;row&quot;</td>
<td>[ &quot;row&quot;</td>
</tr>
<tr>
<td>axes_pad</td>
<td>0.02</td>
<td>float</td>
</tr>
<tr>
<td>add_all</td>
<td>True</td>
<td>[ True</td>
</tr>
<tr>
<td>share_all</td>
<td>False</td>
<td>[ True</td>
</tr>
<tr>
<td>share_x</td>
<td>True</td>
<td>[ True</td>
</tr>
<tr>
<td>share_y</td>
<td>True</td>
<td>[ True</td>
</tr>
<tr>
<td>label_mode</td>
<td>“L”</td>
<td>[ “L”</td>
</tr>
<tr>
<td>axes_class</td>
<td>None</td>
<td>a type object which must be a subclass of Axes</td>
</tr>
</tbody>
</table>

```python
class mpl_toolkits.axes_grid.axes_grid.ImageGrid(fig, rect, nrows_ncols, ngrids=None, direction='row', axes_pad=0.02, add_all=True, share_all=False, aspect=True, label_mode='L', cbar_mode=None, cbar_location='right', cbar_pad=None, cbar_size='5%', cbar_set_cax=True, axes_class=None)
```

Build an `ImageGrid` instance with a grid `nrows*ncols` `Axes` in `Figure` `fig` with `rect=[left, bottom, width, height]` (in `Figure` coordinates) or the subplot position code (e.g., “121”).

Optional keyword arguments:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>direction</td>
<td>&quot;row&quot;</td>
<td>[ &quot;row&quot;</td>
</tr>
<tr>
<td>axes_pad</td>
<td>0.02</td>
<td>float</td>
</tr>
<tr>
<td>add_all</td>
<td>True</td>
<td>[ True</td>
</tr>
<tr>
<td>share_all</td>
<td>False</td>
<td>[ True</td>
</tr>
<tr>
<td>aspect</td>
<td>True</td>
<td>[ True</td>
</tr>
<tr>
<td>label_mode</td>
<td>“L”</td>
<td>[ “L”</td>
</tr>
<tr>
<td>cbar_mode</td>
<td>None</td>
<td>[ “each”</td>
</tr>
<tr>
<td>cbar_location</td>
<td>&quot;right&quot;</td>
<td>[ “left”</td>
</tr>
<tr>
<td>cbar_pad</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>cbar_size</td>
<td>&quot;5%&quot;</td>
<td></td>
</tr>
<tr>
<td>cbar_set_cax</td>
<td>True</td>
<td>[ True</td>
</tr>
<tr>
<td>axes_class</td>
<td>None</td>
<td>a type object which must be a subclass of axes_grid’s subclass of Axes</td>
</tr>
</tbody>
</table>

`cbar_set_cax` [if True, each axes in the grid has a cax] attribute that is bind to associated cbar_axes.
68.4 mpl_toolkits.axes_grid.axis_artist

class mpl_toolkits.axes_grid.axis_artist.AxisArtist(axes, helper, offset=None, 
    axis_direction='bottom', **kw)

An artist which draws axis (a line along which the n-th axes coord is constant) line, ticks, ticklabels, 
and axis label.

axes : axes  helper : an AxisArtistHelper instance.

LABELPAD

ZORDER = 2.5

draw(artist, renderer, *args, **kwargs)

    Draw the axis lines, tick lines and labels

get_axisline_style()

    return the current axisline style.

get_helper()

    Return axis artist helper instance.

get_tightbbox(renderer)

get_transform()

invert_ticklabel_direction()

set_axis_direction(axis_direction)

    Adjust the direction, text angle, text alignment of ticklabels, labels following the matplotlib 
    convention for the rectangle axes.

The axis_direction must be one of [left, right, bottom, top].

<table>
<thead>
<tr>
<th>property</th>
<th>left</th>
<th>bottom</th>
<th>right</th>
<th>top</th>
</tr>
</thead>
<tbody>
<tr>
<td>ticklabels location</td>
<td>&quot;-&quot;</td>
<td>&quot;+&quot;</td>
<td>&quot;+&quot;</td>
<td>&quot;+&quot;</td>
</tr>
<tr>
<td>axislabel location</td>
<td>&quot;-&quot;</td>
<td>&quot;+&quot;</td>
<td>&quot;+&quot;</td>
<td>&quot;+&quot;</td>
</tr>
<tr>
<td>ticklabels angle</td>
<td>90</td>
<td>0</td>
<td>-90</td>
<td>180</td>
</tr>
<tr>
<td>ticklabel va</td>
<td>center</td>
<td>baseline</td>
<td>center</td>
<td>baseline</td>
</tr>
<tr>
<td>ticklabel ha</td>
<td>right</td>
<td>center</td>
<td>right</td>
<td>center</td>
</tr>
<tr>
<td>axislabel angle</td>
<td>180</td>
<td>0</td>
<td>0</td>
<td>180</td>
</tr>
<tr>
<td>axislabel va</td>
<td>center</td>
<td>top</td>
<td>center</td>
<td>bottom</td>
</tr>
<tr>
<td>axislabel ha</td>
<td>right</td>
<td>center</td>
<td>right</td>
<td>center</td>
</tr>
</tbody>
</table>

Note that the direction “+” and “-” are relative to the direction of the increasing coordinate. 
Also, the text angles are actually relative to (90 + angle of the direction to the ticklabel), which 
gives 0 for bottom axis.
**set_axislabel_direction**(label_direction)
Adjust the direction of the axislabel.

ACCEPES: [ ‘+’ | ‘-’ ]

Note that the label_direction ‘+’ and ‘-’ are relative to the direction of the increasing coordinate.

**set_axisline_style**(axisline_style=None,**kw)
Set the axisline style.

**axisline_style** can be a string with axisline style name with optional comma-separated attributes. Alternatively, the attrs can be provided as keywords.

set_arrowstyle("->",size=1.5) set_arrowstyle("->", size=1.5)

Old attrs simply are forgotten.

Without argument (or with arrowstyle=None), return available styles as a list of strings.

**set_label**(s)

**set_ticklabel_direction**(tick_direction)
Adjust the direction of the ticklabel.

ACCEPES: [ ‘+’ | ‘-’ ]

Note that the label_direction ‘+’ and ‘-’ are relative to the direction of the increasing coordinate.

**toggle**(all=None,ticks=None,ticklabels=None,label=None)
Toggle visibility of ticks, ticklabels, and (axis) label. To turn all off,

```
axis.toggle(all=False)
```

To turn all off but ticks on

```
axis.toggle(all=False, ticks=True)
```

To turn all on but (axis) label off

```
axis.toggle(all=True, label=False))
```

class mpl_toolkits.axes_grid.axis_artist.Ticks(ticksize,tick_out=False,**kwargs)
Ticks are derived from Line2D, and note that ticks themselves are markers. Thus, you should use set_mec, set_mew, etc.

To change the tick size (length), you need to use set_ticksize. To change the direction of the ticks (ticks are in opposite direction of ticklabels by default), use set_tick_out(False).

**get_tick_out**()
Return True if the tick will be rotated by 180 degree.

**get_ticksize**()
Return length of the ticks in points.

**set_tick_out**(b)
set True if tick need to be rotated by 180 degree.
set_ticksize(ticksize)
set length of the ticks in points.

class mpl_toolkits.axes_grid.axis_artist.AxisLabel(*kl, **kwargs)
Axis Label. Derived from Text. The position of the text is updated in the fly, so changing text position has no effect. Otherwise, the properties can be changed as a normal Text.

To change the pad between ticklabels and axis label, use set_pad.

get_pad()
return pad in points. See set_pad for more details.

set_axis_direction(d)
Adjust the text angle and text alignment of axis label according to the matplotlib convention.

<table>
<thead>
<tr>
<th>property</th>
<th>left</th>
<th>bottom</th>
<th>right</th>
<th>top</th>
</tr>
</thead>
<tbody>
<tr>
<td>axislabel angle</td>
<td>180</td>
<td>0</td>
<td>0</td>
<td>180</td>
</tr>
<tr>
<td>axislabel va</td>
<td>center</td>
<td>top</td>
<td>center</td>
<td>bottom</td>
</tr>
<tr>
<td>axislabel ha</td>
<td>right</td>
<td>center</td>
<td>right</td>
<td>center</td>
</tr>
</tbody>
</table>

Note that the text angles are actually relative to (90 + angle of the direction to the ticklabel), which gives 0 for bottom axis.

set_pad(pad)
Set the pad in points. Note that the actual pad will be the sum of the internal pad and the external pad (that are set automatically by the AxisArtist), and it only set the internal pad

class mpl_toolkits.axes_grid.axis_artist.TickLabels(**kwargs)
Tick Labels. While derived from Text, this single artist draws all ticklabels. As in AxisLabel, the position of the text is updated in the fly, so changing text position has no effect. Otherwise, the properties can be changed as a normal Text. Unlike the ticklabels of the mainline matplotlib, properties of single ticklabel alone cannot modified.

To change the pad between ticks and ticklabels, use set_pad.

get_texts_widths_heights_descents(renderer)
return a list of width, height, descent for ticklabels.

set_axis_direction(label_direction)
Adjust the text angle and text alignment of ticklabels according to the matplotlib convention.

The label_direction must be one of [left, right, bottom, top].

<table>
<thead>
<tr>
<th>property</th>
<th>left</th>
<th>bottom</th>
<th>right</th>
<th>top</th>
</tr>
</thead>
<tbody>
<tr>
<td>ticklabels angle</td>
<td>90</td>
<td>0</td>
<td>-90</td>
<td>180</td>
</tr>
<tr>
<td>ticklabel va</td>
<td>center</td>
<td>baseline</td>
<td>center</td>
<td>baseline</td>
</tr>
<tr>
<td>ticklabel ha</td>
<td>right</td>
<td>center</td>
<td>right</td>
<td>center</td>
</tr>
</tbody>
</table>

Note that the text angles are actually relative to (90 + angle of the direction to the ticklabel), which gives 0 for bottom axis.
CHAPTER SIXTYNINE

THE MATPLOTLIB AXES_GRID1 TOOLKIT API

Release 2.0.2
Date May 10, 2017

69.1 mpl_toolkits.axes_grid1.anchored_artists

class mpl_toolkits.axes_grid1.anchored_artists.AnchoredAuxTransformBox(transform, loc, pad=None, borderpad=0.5, prop=None, frameon=True, **kwargs)

Bases: matplotlib.offsetbox.AnchoredOffsetbox

An anchored container with transformed coordinates.

Artists added to the drawing_area are scaled according to the coordinates of the transformation used. The dimensions of this artist will scale to contain the artists added.

Parameters transform : matplotlib.transforms.Transform

The transformation object for the coordinate system in use, i.e., matplotlib.axes.Axes.transData.

loc : int

Location of this artist. Valid location codes are:

<table>
<thead>
<tr>
<th>Location</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>upper right</td>
<td>1</td>
</tr>
<tr>
<td>upper left</td>
<td>2</td>
</tr>
<tr>
<td>lower left</td>
<td>3</td>
</tr>
<tr>
<td>lower right</td>
<td>4</td>
</tr>
<tr>
<td>right</td>
<td>5</td>
</tr>
<tr>
<td>center left</td>
<td>6</td>
</tr>
<tr>
<td>center right</td>
<td>7</td>
</tr>
<tr>
<td>lower center</td>
<td>8</td>
</tr>
</tbody>
</table>
'upper center' : 9,
'center' : 10

**pad** : int or float, optional

Padding around the child objects, in fraction of the font size. Defaults to 0.4.

**borderpad** : int or float, optional

Border padding, in fraction of the font size. Defaults to 0.5.

**prop** : **matplotlib.font_manager.FontProperties**, optional

Font property used as a reference for paddings.

**frameon** : bool, optional

If True, draw a box around this artists. Defaults to True.

**kwargs** :

Keyworded arguments to pass to **matplotlib.offsetbox. AnchoredOffsetbox**.

**Examples**

To display an ellipse in the upper left, with a width of 0.1 and height of 0.4 in data coordinates:

```python
>>> box = AnchoredAuxTransformBox(ax.transData, loc=2)
>>> el = Ellipse((0,0), width=0.1, height=0.4, angle=30)
>>> box.drawing_area.add_artist(el)
>>> ax.add_artist(box)
```

**Attributes**

| drawing_area | (**matplotlib.offsetbox.AuxTransformBox**) A container for artists to display. |

**class** **mpl_toolkits.axes_grid1.anchored_artists.AnchoredDrawingArea**(width, height, xdescent, ydescent, loc, pad=0.4, borderpad=0.5, prop=None, frameon=True, **kwargs)

An anchored container with a fixed size and fillable DrawingArea.

Artists added to the drawing_area will have their coordinates interpreted as pixels. Any transformations set on the artists will be overridden.
**Parameters**

**width, height** : int or float
   width and height of the container, in pixels.

**xdescent, ydescent** : int or float
   descent of the container in the x- and y- direction, in pixels.

**loc** : int
   Location of this artist. Valid location codes are:

   ```
   'upper right' : 1,
   'upper left'  : 2,
   'lower left'  : 3,
   'lower right' : 4,
   'right'      : 5,
   'center left' : 6,
   'center right': 7,
   'lower center': 8,
   'upper center': 9,
   'center'     : 10
   ```

**pad** : int or float, optional
   Padding around the child objects, in fraction of the font size. Defaults to 0.4.

**borderpad** : int or float, optional
   Border padding, in fraction of the font size. Defaults to 0.5.

**prop** : `matplotlib.font_manager.FontProperties`, optional
   Font property used as a reference for paddings.

**frameon** : bool, optional
   If True, draw a box around this artists. Defaults to True.

**kwargs** :
   Keyworded arguments to pass to `matplotlib.offsetbox.AnchoredOffsetbox`.

**Examples**

To display blue and red circles of different sizes in the upper right of an axes `ax`:

```python
>>> ada = AnchoredDrawingArea(20, 20, 0, 0, loc=1, frameon=False)
>>> ada.drawing_area.add_artist(Circle((10, 10), 10, fc="b"))
>>> ada.drawing_area.add_artist(Circle((30, 10), 5, fc="r"))
>>> ax.add_artist(ada)
```
Attributes

drawing_area (matplotlib.offsetbox.DrawingArea) A container for artists to display.

class mpl_toolkits.axes_grid1.anchored_artists.AnchoredEllipse(transform, width, height, angle, loc, pad=0.1, borderpad=0.1, prop=None, frameon=True, **kwargs)

Bases: matplotlib.offsetbox.AnchoredOffsetbox

Draw an anchored ellipse of a given size.

Parameters

transform : matplotlib.transforms.Transform

The transformation object for the coordinate system in use, i.e., matplotlib.axes.Axes.transData.

width, height : int or float

Width and height of the ellipse, given in coordinates of transform.

angle : int or float

Rotation of the ellipse, in degrees, anti-clockwise.

loc : int

Location of this size bar. Valid location codes are:

<table>
<thead>
<tr>
<th>Location</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>'upper right'</td>
<td>1</td>
</tr>
<tr>
<td>'upper left'</td>
<td>2</td>
</tr>
<tr>
<td>'lower left'</td>
<td>3</td>
</tr>
<tr>
<td>'lower right'</td>
<td>4</td>
</tr>
<tr>
<td>'right'</td>
<td>5</td>
</tr>
<tr>
<td>'center left'</td>
<td>6</td>
</tr>
<tr>
<td>'center right'</td>
<td>7</td>
</tr>
<tr>
<td>'lower center'</td>
<td>8</td>
</tr>
<tr>
<td>'upper center'</td>
<td>9</td>
</tr>
<tr>
<td>'center'</td>
<td>10</td>
</tr>
</tbody>
</table>

pad : int or float, optional

Padding around the ellipse, in fraction of the font size. Defaults to 0.1.

borderpad : int or float, optional

Border padding, in fraction of the font size. Defaults to 0.1.

frameon : bool, optional

If True, draw a box around the ellipse. Defaults to True.

prop : matplotlib.font_manager.FontProperties, optional
Font property used as a reference for paddings.

**kwargs :

Keyworded arguments to pass to `matplotlib.offsetbox.AnchoredOffsetbox`.

**Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ellipse</td>
<td>Ellipse patch drawn.</td>
</tr>
</tbody>
</table>

**mpl_toolkits.axes_grid1.anchored_artists.AnchoredSizeBar**

```python
class mpl_toolkits.axes_grid1.anchored_artists.AnchoredSizeBar(transform, size, label, loc, pad=0.1, borderpad=0.1, sep=2, frameon=True, size_vertical=0, color='black', label_top=False, fontproperties=None, **kwargs)
```

Bases: `matplotlib.offsetbox.AnchoredOffsetbox`  

Draw a horizontal scale bar with a center-aligned label underneath.

**Parameters**

- **transform** : `matplotlib.transforms.Transform`  
  The transformation object for the coordinate system in use, i.e., `matplotlib.axes.Axes.transData`.

- **size** : int or float  
  Horizontal length of the size bar, given in coordinates of `transform`.

- **label** : str  
  Label to display.

- **loc** : int  
  Location of this size bar. Valid location codes are:

<table>
<thead>
<tr>
<th>Location</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>upper right</td>
<td>1</td>
</tr>
<tr>
<td>upper left</td>
<td>2</td>
</tr>
<tr>
<td>lower left</td>
<td>3</td>
</tr>
<tr>
<td>lower right</td>
<td>4</td>
</tr>
<tr>
<td>right</td>
<td>5</td>
</tr>
<tr>
<td>center left</td>
<td>6</td>
</tr>
<tr>
<td>center right</td>
<td>7</td>
</tr>
<tr>
<td>lower center</td>
<td>8</td>
</tr>
<tr>
<td>upper center</td>
<td>9</td>
</tr>
<tr>
<td>center</td>
<td>10</td>
</tr>
</tbody>
</table>

- **pad** : int or float, optional
Padding around the label and size bar, in fraction of the font size. Defaults to 0.1.

**borderpad**: int or float, optional

Border padding, in fraction of the font size. Defaults to 0.1.

**sep**: int or float, optional

Separation between the label and the size bar, in points. Defaults to 2.

**frameon**: bool, optional

If True, draw a box around the horizontal bar and label. Defaults to True.

**size_vertical**: int or float, optional

Vertical length of the size bar, given in coordinates of *transform*. Defaults to 0.

**color**: str, optional

Color for the size bar and label. Defaults to black.

**label_top**: bool, optional

If True, the label will be over the size bar. Defaults to False.

**fontproperties**: *matplotlib.font_manager.FontProperties*, optional

Font properties for the label text.

**kwargs**

Keyworded arguments to pass to *matplotlib.offsetbox.AnchoredOffsetbox*.

**Notes**

If *prop* is passed as a keyworded argument, but *fontproperties* is not, then *prop* is be assumed to be the intended *fontproperties*. Using both *prop* and *fontproperties* is not supported.

**Examples**

```python
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> from mpl_toolkits.axes_grid1.anchored_artists import AnchoredSizeBar
>>> fig, ax = plt.subplots()
>>> ax.imshow(np.random.random((10,10)))
>>> bar = AnchoredSizeBar(ax.transData, 3, '3 data units', 4)
>>> ax.add_artist(bar)
>>> fig.show()
```

Using all the optional parameters
>>> import matplotlib.font_manager as fm
>>> fontprops = fm.FontProperties(size=14, family='monospace')
>>> bar = AnchoredSizeBar(ax.transData, 3, '3 units', 4, pad=0.5, sep=5,
borderpad=0.5, frameon=False, size_vertical=0.5, color='white',
fontproperties=fontprops)

Attributes

<table>
<thead>
<tr>
<th>size_bar</th>
<th>(matplotlib.offsetbox.AuxTransformBox)</th>
<th>Container for the size bar.</th>
</tr>
</thead>
<tbody>
<tr>
<td>txt_label</td>
<td>(matplotlib.offsetbox.TextArea)</td>
<td>Container for the label of the size bar.</td>
</tr>
</tbody>
</table>

### 69.2 mpl_toolkits.axes_grid1.inset_locator

A collection of functions and objects for creating or placing inset axes.

```python
class mpl_toolkits.axes_grid1.inset_locator.AnchoredLocatorBase
    (bbox_to_anchor,
     offsetbox, loc,
     borderpad=0.5,
     bbox_transform=None)

Bases: matplotlib.offsetbox.AnchoredOffsetbox
draw(renderer)
```

```python
class mpl_toolkits.axes_grid1.inset_locator.AnchoredSizeLocator
    (bbox_to_anchor,
     x_size, y_size, loc,
     borderpad=0.5,
     bbox_transform=None)

Bases: mpl_toolkits.axes_grid1.inset_locator.AnchoredLocatorBase
get_extent(renderer)
```

```python
class mpl_toolkits.axes_grid1.inset_locator.AnchoredZoomLocator
    (parent_axes,
     zoom, loc, borderpad=0.5,
     bbox_to_anchor=None,
     bbox_transform=None)

Bases: mpl_toolkits.axes_grid1.inset_locator.AnchoredLocatorBase
get_extent(renderer)
```

```python
class mpl_toolkits.axes_grid1.inset_locator.BboxConnector
    (bbox1, bbox2, loc1,
     loc2=None, **kwargs)

Bases: matplotlib.patches.Patch
Connect two bboxes with a straight line.
```

Parameters:

- `bbox1`, `bbox2`: `matplotlib.transforms.Bbox`
Bounding boxes to connect.

**loc1** : {1, 2, 3, 4}

Corner of `bbox1` to draw the line. Valid values are:

<table>
<thead>
<tr>
<th>Corner</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>'upper right'</td>
<td>1</td>
</tr>
<tr>
<td>'upper left'</td>
<td>2</td>
</tr>
<tr>
<td>'lower left'</td>
<td>3</td>
</tr>
<tr>
<td>'lower right'</td>
<td>4</td>
</tr>
</tbody>
</table>

**loc2** : {1, 2, 3, 4}, optional

Corner of `bbox2` to draw the line. If None, defaults to `loc1`. Valid values are:

<table>
<thead>
<tr>
<th>Corner</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>'upper right'</td>
<td>1</td>
</tr>
<tr>
<td>'upper left'</td>
<td>2</td>
</tr>
<tr>
<td>'lower left'</td>
<td>3</td>
</tr>
<tr>
<td>'lower right'</td>
<td>4</td>
</tr>
</tbody>
</table>

****kwargs

Patch properties for the line drawn. Valid arguments include:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>agg_filter</em></td>
<td>unknown</td>
</tr>
<tr>
<td><em>alpha</em></td>
<td>float or None</td>
</tr>
<tr>
<td><em>animated</em></td>
<td>[True</td>
</tr>
<tr>
<td><em>antialiased</em> or <em>aa</em></td>
<td>[True</td>
</tr>
<tr>
<td><em>axes</em></td>
<td>an <em>Axes</em> instance</td>
</tr>
<tr>
<td><em>capstyle</em></td>
<td>['butt'</td>
</tr>
<tr>
<td><em>clip_box</em></td>
<td>a <em>matplotlib.transforms.Bbox</em> instance</td>
</tr>
<tr>
<td><em>clip_on</em></td>
<td>[True</td>
</tr>
<tr>
<td><em>clip_path</em></td>
<td>((Path, Transform)</td>
</tr>
<tr>
<td><em>color</em></td>
<td><em>matplotlib</em> color spec</td>
</tr>
<tr>
<td><em>contains</em></td>
<td>a callable function</td>
</tr>
<tr>
<td><em>edgecolor</em> or <em>ec</em></td>
<td><em>mpl</em> color spec, None, ‘none’, or ‘auto’</td>
</tr>
<tr>
<td><em>facecolor</em> or <em>fc</em></td>
<td><em>mpl</em> color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td><em>figure</em></td>
<td>a <em>matplotlib.figure.Figure</em> instance</td>
</tr>
<tr>
<td><em>fill</em></td>
<td>[True</td>
</tr>
<tr>
<td><em>gid</em></td>
<td>an id string</td>
</tr>
<tr>
<td><em>hatch</em></td>
<td>[''</td>
</tr>
<tr>
<td><em>joinstyle</em></td>
<td>['miter'</td>
</tr>
<tr>
<td><em>label</em></td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td><em>linestyle</em> or <em>ls</em></td>
<td>['solid'</td>
</tr>
<tr>
<td><em>linewidth</em> or <em>lw</em></td>
<td>float or None for default</td>
</tr>
<tr>
<td><em>path_effects</em></td>
<td>unknown</td>
</tr>
<tr>
<td><em>picker</em></td>
<td>[None</td>
</tr>
<tr>
<td><em>rasterized</em></td>
<td>[True</td>
</tr>
</tbody>
</table>
Table 69.1 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

**static connect_bbox** *(bbox1, bbox2, loc1, loc2= None)*

Helper function to obtain a Path from one bbox to another.

**Parameters bbox1, bbox2 : matplotlib.transforms.Bbox**

Bounding boxes to connect.

**loc1 : {1, 2, 3, 4}**

Corner of bbox1 to use. Valid values are:

```python
{'upper_right' : 1,
 'upper left'   : 2,
 'lower left'   : 3,
 'lower right' : 4}
```

**loc2 : {1, 2, 3, 4}, optional**

Corner of bbox2 to use. If None, defaults to loc1. Valid values are:

```python
{'upper_right' : 1,
 'upper left'   : 2,
 'lower left'   : 3,
 'lower right' : 4}
```

**Returns path : matplotlib.path.Path**

A line segment from the loc1 corner of bbox1 to the loc2 corner of bbox2.

**static get_bbox_edge_pos** *(bbox, loc)*

Helper function to obtain the location of a corner of a bbox

**Parameters bbox : matplotlib.transforms.Bbox**

**loc : {1, 2, 3, 4}**

Corner of bbox. Valid values are:

```python
{'upper_right' : 1,
 'upper left'   : 2,
 'lower left'   : 3,
 'lower right' : 4}
```

**Returns x, y : float**

Coordinates of the corner specified by loc.
get_path()

Return the path of this patch

class mpl_toolkits.axes_grid1.inset_locator.BboxConnectorPatch(bbox1, bbox2, loc1a, loc2a, loc1b, loc2b, **kwargs)

Bases: mpl_toolkits.axes_grid1.inset_locator.BboxConnector

Connect two bboxes with a quadrilateral.

The quadrilateral is specified by two lines that start and end at corners of the bboxes. The four sides of the quadrilateral are defined by the two lines given, the line between the two corners specified in `bbox1` and the line between the two corners specified in `bbox2`.

**Parameters**

- **bbox1, bbox2**: `matplotlib.transforms.Bbox`  
  Bounding boxes to connect.

- **loc1a, loc2a**: `{1, 2, 3, 4}`  
  Corners of `bbox1` and `bbox2` to draw the first line. Valid values are:

  | 'upper right' | 1 |
  | 'upper left'  | 2 |
  | 'lower left'  | 3 |
  | 'lower right' | 4 |

- **loc1b, loc2b**: `{1, 2, 3, 4}`  
  Corners of `bbox1` and `bbox2` to draw the second line. Valid values are:

  | 'upper right' | 1 |
  | 'upper left'  | 2 |
  | 'lower left'  | 3 |
  | 'lower right' | 4 |

- **kwargs**

  Patch properties for the line drawn:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float or None</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>antialiased or aa</td>
<td>[True</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>capstyle</td>
<td>['butt'</td>
</tr>
<tr>
<td>clip_box</td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color</td>
<td><code>matplotlib</code> color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
</tbody>
</table>

Continued on next page
Table 69.2 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>edgecolor</strong></td>
<td>mpl color spec, None, ‘none’, or ‘auto’</td>
</tr>
<tr>
<td><strong>facecolor</strong></td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td><strong>figure</strong></td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td><strong>fill</strong></td>
<td>[True</td>
</tr>
<tr>
<td><strong>gid</strong></td>
<td>an id string</td>
</tr>
<tr>
<td><strong>hatch</strong></td>
<td>[’/’</td>
</tr>
<tr>
<td><strong>joinstyle</strong></td>
<td>[’miter’</td>
</tr>
<tr>
<td><strong>label</strong></td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td><strong>linestyle</strong></td>
<td>[’solid’</td>
</tr>
<tr>
<td><strong>linewidth</strong></td>
<td>float or None for default</td>
</tr>
<tr>
<td><strong>path_effects</strong></td>
<td>unknown</td>
</tr>
<tr>
<td><strong>picker</strong></td>
<td>[None</td>
</tr>
<tr>
<td><strong>rasterized</strong></td>
<td>[True</td>
</tr>
<tr>
<td><strong>sketch_params</strong></td>
<td>unknown</td>
</tr>
<tr>
<td><strong>snap</strong></td>
<td>unknown</td>
</tr>
<tr>
<td><strong>transform</strong></td>
<td>Transform instance</td>
</tr>
<tr>
<td><strong>url</strong></td>
<td>a url string</td>
</tr>
<tr>
<td><strong>visible</strong></td>
<td>[True</td>
</tr>
<tr>
<td><strong>zorder</strong></td>
<td>any number</td>
</tr>
</tbody>
</table>

**get_path()**

Return the path of this patch

**class** mpl_toolkits.axes_grid1.inset_locator.BboxPatch(bbox, **kwargs)

Bases: matplotlib.patches.Patch

Patch showing the shape bounded by a Bbox.

**Parameters bbox**: matplotlib.transforms.Bbox

Bbox to use for the extents of this patch.

**kwargs**

Patch properties. Valid arguments include:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>agg_filter</strong></td>
<td>unknown</td>
</tr>
<tr>
<td><strong>alpha</strong></td>
<td>float or None</td>
</tr>
<tr>
<td><strong>animated</strong></td>
<td>[True</td>
</tr>
<tr>
<td><strong>antialiased</strong></td>
<td>[True</td>
</tr>
<tr>
<td><strong>axes</strong></td>
<td>an Axes instance</td>
</tr>
<tr>
<td><strong>capstyle</strong></td>
<td>[’butt’</td>
</tr>
<tr>
<td><strong>clip_box</strong></td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td><strong>clip_on</strong></td>
<td>[True</td>
</tr>
<tr>
<td><strong>clip_path</strong></td>
<td>(Path, Transform)</td>
</tr>
</tbody>
</table>
Table 69.3 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>color</td>
<td>matplotlib color spec</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>edgecolor or ec</td>
<td>mpl color spec, None, ‘none’, or ‘auto’</td>
</tr>
<tr>
<td>facecolor or fc</td>
<td>mpl color spec, or None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td>figure</td>
<td>a matplotlib.figure.Figure instance</td>
</tr>
<tr>
<td>fill</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>hatch</td>
<td>[‘/’</td>
</tr>
<tr>
<td>joinstyle</td>
<td>[‘miter’</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>linestyle or ls</td>
<td>[‘solid’</td>
</tr>
<tr>
<td>linewidth or lw</td>
<td>float or None for default</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

**get_path()**

Return the path of this patch

class mpl_toolkits.axes_grid1.inset_locator.InsetPosition(parent, lbwh)

Bases: object

An object for positioning an inset axes.

This is created by specifying the normalized coordinates in the axes, instead of the figure.

**Parameters parent** : matplotlib.axes.Axes

Axes to use for normalizing coordinates.

**lbwh** : iterable of four floats

The left edge, bottom edge, width, and height of the inset axes, in units of the normalized coordinate of the parent axes.

**See also:**

matplotlib.axes.Axes.set_axes_locator()
Examples

The following bounds the inset axes to a box with 20% of the parent axes’s height and 40% of the width. The size of the axes specified ([0, 0, 1, 1]) ensures that the axes completely fills the bounding box:

```python
>>> parent_axes = plt.gca()
>>> ax_ins = plt.axes([0, 0, 1, 1])
>>> ip = InsetPosition(ax, [0.5, 0.1, 0.4, 0.2])
>>> ax_ins.set_axes_locator(ip)
```

mpl_toolkits.axes_grid1.inset_locator.inset_axes(parent_axes, width, height, loc=1, bbox_to_anchor=None, bbox_transform=None, axes_class=None, axes_kwargs=None, borderpad=0.5)

Create an inset axes with a given width and height.

Both sizes used can be specified either in inches or percentage of the parent axes.

**Parameters**

parent_axes : matplotlib.axes.Axes
    Axes to place the inset axes.

width, height : float or str
    Size of the inset axes to create.

loc : int or string, optional, default to 1
    Location to place the inset axes. The valid locations are:

    ```
    'upper right' : 1,
    'upper left' : 2,
    'lower left' : 3,
    'lower right' : 4,
    'right' : 5,
    'center left' : 6,
    'center right' : 7,
    'lower center' : 8,
    'upper center' : 9,
    'center' : 10
    ```

bbox_to_anchor : tuple or matplotlib.transforms.BboxBase, optional
    Bbox that the inset axes will be anchored. Can be a tuple of [left, bottom, width, height], or a tuple of [left, bottom].

bbox_transform : matplotlib.transforms.Transform, optional
    Transformation for the bbox. if None, parent_axes.transAxes is used.

axes_class : matplotlib.axes.Axes, optional
    If specified, the inset axes created with be created with this class’s constructor.
axes_kargs : dict, optional

Keyworded arguments to pass to the constructor of the inset axes. Valid arguments include:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>adjustable</td>
<td>['box', 'datalim', 'box-forced']</td>
</tr>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>anchor</td>
<td>unknown</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>aspect</td>
<td>unknown</td>
</tr>
<tr>
<td>autoscale_on</td>
<td>unknown</td>
</tr>
<tr>
<td>autoscalex_on</td>
<td>unknown</td>
</tr>
<tr>
<td>autoscaley_on</td>
<td>unknown</td>
</tr>
<tr>
<td>axes</td>
<td>an Axes instance</td>
</tr>
<tr>
<td>axes_locator</td>
<td>unknown</td>
</tr>
<tr>
<td>axisbelow</td>
<td>[True</td>
</tr>
<tr>
<td>clip_box</td>
<td>a matplotlib.transforms.Bbox instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color_cycle</td>
<td>unknown</td>
</tr>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>facecolor</td>
<td>unknown</td>
</tr>
<tr>
<td>fc</td>
<td>unknown</td>
</tr>
<tr>
<td>figure</td>
<td>unknown</td>
</tr>
<tr>
<td>frame_on</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>navigate</td>
<td>[True</td>
</tr>
<tr>
<td>navigate_mode</td>
<td>unknown</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>position</td>
<td>unknown</td>
</tr>
<tr>
<td>rasterization_zorder</td>
<td>unknown</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>title</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xbound</td>
<td>unknown</td>
</tr>
<tr>
<td>xlabel</td>
<td>unknown</td>
</tr>
<tr>
<td>xlim</td>
<td>unknown</td>
</tr>
<tr>
<td>xmargin</td>
<td>unknown</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xscale</td>
<td>['linear', 'log', 'logit', 'symlog']</td>
</tr>
<tr>
<td>xticklabels</td>
<td>sequence of strings</td>
</tr>
<tr>
<td>xticks</td>
<td>sequence of floats</td>
</tr>
<tr>
<td>ybound</td>
<td>unknown</td>
</tr>
<tr>
<td>ylabel</td>
<td>unknown</td>
</tr>
<tr>
<td>ylim</td>
<td>unknown</td>
</tr>
<tr>
<td>ymargin</td>
<td>unknown</td>
</tr>
<tr>
<td>yscale</td>
<td>['linear', 'log', 'logit', 'symlog']</td>
</tr>
<tr>
<td>yticklabels</td>
<td>sequence of strings</td>
</tr>
<tr>
<td>yticks</td>
<td>sequence of floats</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

**borderpad**: float, optional

Padding between inset axes and the bbox_to_anchor. Defaults to 0.5.

**Returns inset_axes**: axes_class

Inset axes object created.

```python
mpl_toolkits.axes_grid1.inset_locator.mark_inset(parent_axes, inset_axes, loc1, loc2, **kwargs)
```

Draw a box to mark the location of an area represented by an inset axes.

This function draws a box in `parent_axes` at the bounding box of `inset_axes`, and shows a connection with the inset axes by drawing lines at the corners, giving a “zoomed in” effect.

**Parameters**
- `parent_axes`: `matplotlib.axes.Axes`

Axes which contains the area of the inset axes.

- `inset_axes`: `matplotlib.axes.Axes`

The inset axes.

- `loc1, loc2`: {1, 2, 3, 4}

Corners to use for connecting the inset axes and the area in the parent axes.

**kwargs

Patch properties for the lines and box drawn:
Table 69.5 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>clip_box</code></td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td><code>clip_on</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>clip_path</code></td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td><code>color</code></td>
<td><code>matplotlib</code> color spec</td>
</tr>
<tr>
<td><code>contains</code></td>
<td>a callable function</td>
</tr>
<tr>
<td><code>edgecolor</code></td>
<td>or <code>ec</code> mpl color spec, None, ‘none’, or ‘auto’</td>
</tr>
<tr>
<td><code>facecolor</code></td>
<td>or <code>fc</code> mpl color spec, None for default, or ‘none’ for no color</td>
</tr>
<tr>
<td><code>figure</code></td>
<td>a <code>matplotlib.figure.Figure</code> instance</td>
</tr>
<tr>
<td><code>fill</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>gid</code></td>
<td>an id string</td>
</tr>
<tr>
<td><code>hatch</code></td>
<td>[‘/’</td>
</tr>
<tr>
<td><code>joinstyle</code></td>
<td>[‘miter’</td>
</tr>
<tr>
<td><code>label</code></td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td><code>linestyle</code></td>
<td>or <code>ls</code> [‘solid’</td>
</tr>
<tr>
<td><code>linewidth</code></td>
<td>or <code>lw</code> float or None for default</td>
</tr>
<tr>
<td><code>path_effects</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>picker</code></td>
<td>[None</td>
</tr>
<tr>
<td><code>rasterized</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>sketch_params</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>snap</code></td>
<td>unknown</td>
</tr>
<tr>
<td><code>transform</code></td>
<td><code>Transform</code> instance</td>
</tr>
<tr>
<td><code>url</code></td>
<td>a url string</td>
</tr>
<tr>
<td><code>visible</code></td>
<td>[True</td>
</tr>
<tr>
<td><code>zorder</code></td>
<td>any number</td>
</tr>
</tbody>
</table>

**Returns** `pp` : `matplotlib.patches.Patch`

The patch drawn to represent the area of the inset axes.

`p1`, `p2` : `matplotlib.patches.Patch`

The patches connecting two corners of the inset axes and its area.

```
mpl_toolkits.axes_grid1.inset_locator.zoomed_inset_axes(parent_axes, zoom, loc=1, bbox_to_anchor=None, bbox_transform=None, axes_class=None, axes_kw_params=None, borderpad=0.5)
```

Create an anchored inset axes by scaling a parent axes.

**Parameters** `parent_axes` : `matplotlib.axes.Axes`

Axes to place the inset axes.

`zoom` : float

Scaling factor of the data axes. `zoom > 1` will enlarge the coordinates (i.e., “zoomed in”), while `zoom < 1` will shrink the coordinates (i.e., “zoomed out”).
loc : int or string, optional, default to 1

Location to place the inset axes. The valid locations are:

```
'upper right' : 1,
'upper left'  : 2,
'lower left'  : 3,
'lower right' : 4,
'right'      : 5,
'center left' : 6,
'center right': 7,
'lower center': 8,
'upper center': 9,
'center'     : 10
```

bbox_to_anchor : tuple or `matplotlib.transforms.BboxBase`, optional

Bbox that the inset axes will be anchored. Can be a tuple of [left, bottom, width, height], or a tuple of [left, bottom].

bbox_transform : `matplotlib.transforms.Transform`, optional

Transformation for the bbox. If None, `parent_axes.transAxes` is used.

axes_class : `matplotlib.axes.Axes` type, optional

If specified, the inset axes created with be created with this class’s constructor.

axes_kwargs : dict, optional

Keyworded arguments to pass to the constructor of the inset axes. Valid arguments include:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>adjustable</td>
<td>[‘box’</td>
</tr>
<tr>
<td>agg_filter</td>
<td>unknown</td>
</tr>
<tr>
<td>alpha</td>
<td>float (0.0 transparent through 1.0 opaque)</td>
</tr>
<tr>
<td>anchor</td>
<td>unknown</td>
</tr>
<tr>
<td>animated</td>
<td>[True</td>
</tr>
<tr>
<td>aspect</td>
<td>unknown</td>
</tr>
<tr>
<td>autoscale_on</td>
<td>unknown</td>
</tr>
<tr>
<td>autoscalex_on</td>
<td>unknown</td>
</tr>
<tr>
<td>autoscaley_on</td>
<td>unknown</td>
</tr>
<tr>
<td>axes</td>
<td>an <code>Axes</code> instance</td>
</tr>
<tr>
<td>axes_locator</td>
<td>unknown</td>
</tr>
<tr>
<td>axisbelow</td>
<td>[ True</td>
</tr>
<tr>
<td>clip_box</td>
<td>a <code>matplotlib.transforms.Bbox</code> instance</td>
</tr>
<tr>
<td>clip_on</td>
<td>[True</td>
</tr>
<tr>
<td>clip_path</td>
<td>[(Path, Transform)</td>
</tr>
<tr>
<td>color_cycle</td>
<td>unknown</td>
</tr>
</tbody>
</table>

Continued on next page
Table 69.6 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>contains</td>
<td>a callable function</td>
</tr>
<tr>
<td>facecolor</td>
<td>unknown</td>
</tr>
<tr>
<td>fc</td>
<td>unknown</td>
</tr>
<tr>
<td>figure</td>
<td>unknown</td>
</tr>
<tr>
<td>frame_on</td>
<td>[True</td>
</tr>
<tr>
<td>gid</td>
<td>an id string</td>
</tr>
<tr>
<td>label</td>
<td>string or anything printable with ‘%s’ conversion.</td>
</tr>
<tr>
<td>navigate</td>
<td>[True</td>
</tr>
<tr>
<td>navigate_mode</td>
<td>unknown</td>
</tr>
<tr>
<td>path_effects</td>
<td>unknown</td>
</tr>
<tr>
<td>picker</td>
<td>[None</td>
</tr>
<tr>
<td>position</td>
<td>unknown</td>
</tr>
<tr>
<td>rasterization_zorder</td>
<td>unknown</td>
</tr>
<tr>
<td>rasterized</td>
<td>[True</td>
</tr>
<tr>
<td>sketch_params</td>
<td>unknown</td>
</tr>
<tr>
<td>snap</td>
<td>unknown</td>
</tr>
<tr>
<td>title</td>
<td>unknown</td>
</tr>
<tr>
<td>transform</td>
<td>Transform instance</td>
</tr>
<tr>
<td>url</td>
<td>a url string</td>
</tr>
<tr>
<td>visible</td>
<td>[True</td>
</tr>
<tr>
<td>xbound</td>
<td>unknown</td>
</tr>
<tr>
<td>xlabel</td>
<td>unknown</td>
</tr>
<tr>
<td>xlim</td>
<td>unknown</td>
</tr>
<tr>
<td>xmargin</td>
<td>unknown</td>
</tr>
<tr>
<td>xscale</td>
<td>['linear'</td>
</tr>
<tr>
<td>xticklabels</td>
<td>sequence of strings</td>
</tr>
<tr>
<td>xticks</td>
<td>sequence of floats</td>
</tr>
<tr>
<td>ybound</td>
<td>unknown</td>
</tr>
<tr>
<td>ylabel</td>
<td>unknown</td>
</tr>
<tr>
<td>ylim</td>
<td>unknown</td>
</tr>
<tr>
<td>ymargin</td>
<td>unknown</td>
</tr>
<tr>
<td>yscale</td>
<td>['linear'</td>
</tr>
<tr>
<td>yticklabels</td>
<td>sequence of strings</td>
</tr>
<tr>
<td>yticks</td>
<td>sequence of floats</td>
</tr>
<tr>
<td>zorder</td>
<td>any number</td>
</tr>
</tbody>
</table>

**borderpad**: float, optional

Padding between inset axes and the bbox_to_anchor. Defaults to 0.5.

**Returns** inset_axes: axes_class

Inset axes object created.
Part V

mplot3d
The mplot3d toolkit adds simple 3D plotting capabilities to matplotlib by supplying an axes object that can create a 2D projection of a 3D scene. The resulting graph will have the same look and feel as regular 2D plots.

The interactive backends also provide the ability to rotate and zoom the 3D scene. One can rotate the 3D scene by simply clicking-and-dragging the scene.Zooming is done by right-clicking the scene and dragging the mouse up and down. Note that one does not use the zoom button like one would use for regular 2D plots.
70.1 mplot3d tutorial

Contents

- mplot3d tutorial
  - Getting started
  - Line plots
  - Scatter plots
  - Wireframe plots
  - Surface plots
  - Tri-Surface plots
  - Contour plots
  - Filled contour plots
  - Polygon plots
  - Bar plots
  - Quiver
  - 2D plots in 3D
  - Text
  - Subplotting

70.1.1 Getting started

An Axes3D object is created just like any other axes using the projection='3d' keyword. Create a new
matplotlib.figure.Figure and add a new axes to it of type Axes3D:

```python
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
```

New in version 1.0.0: This approach is the preferred method of creating a 3D axes.

Note: Prior to version 1.0.0, the method of creating a 3D axes was different. For those using older versions
of matplotlib, change `ax = fig.add_subplot(111, projection='3d')` to `ax = Axes3D(fig)`.
70.1.2 Line plots

**Axes3D.plot**(*xs, ys, *args, **kwargs*)

Plot 2D or 3D data.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>xs, ys</em></td>
<td>x, y coordinates of vertices</td>
</tr>
<tr>
<td><em>zs</em></td>
<td>z value(s), either one for all points or one for each point.</td>
</tr>
<tr>
<td><em>zdir</em></td>
<td>Which direction to use as z (‘x’, ‘y’ or ‘z’) when plotting a 2D set.</td>
</tr>
</tbody>
</table>

Other arguments are passed on to *plot*().

![Parametric curve](image)

70.1.3 Scatter plots

**Axes3D.scatter**(*xs, ys, zs=0, zdir='z', s=20, c=None, depthshade=True, *args, **kwargs*)

Create a scatter plot.
<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$xs, ys$</td>
<td>Positions of data points.</td>
</tr>
<tr>
<td>$zs$</td>
<td>Either an array of the same length as $xs$ and $ys$ or a single value to place all points in the same plane. Default is 0.</td>
</tr>
<tr>
<td>$zdir$</td>
<td>Which direction to use as $z$ (‘x’, ‘y’ or ‘z’) when plotting a 2D set.</td>
</tr>
<tr>
<td>$s$</td>
<td>Size in points$^2$. It is a scalar or an array of the same length as $x$ and $y$.</td>
</tr>
<tr>
<td>$c$</td>
<td>A color. $c$ can be a single color format string, or a sequence of color specifications of length $N$, or a sequence of $N$ numbers to be mapped to colors using the $cmap$ and $norm$ specified via kwargs (see below). Note that $c$ should not be a single numeric RGB or RGBA sequence because that is indistinguishable from an array of values to be colormapped. $c$ can be a 2-D array in which the rows are RGB or RGBA, however, including the case of a single row to specify the same color for all points.</td>
</tr>
<tr>
<td>$depthshade$</td>
<td>Whether or not to shade the scatter markers to give the appearance of depth. Default is True.</td>
</tr>
</tbody>
</table>

Keyword arguments are passed on to `scatter()`. 

Returns a `Patch3DCollection`.
### 70.1.4 Wireframe plots

**Axes3D.plot_wireframe(X, Y, Z, *args, **kwargs)**

Plot a 3D wireframe.

The `rstride` and `cstride` kwargs set the stride used to sample the input data to generate the graph. If either is 0 the input data is not sampled along this direction producing a 3D line plot rather than a wireframe plot. The stride arguments are only used by default if in the 'classic' mode. They are now superseded by `rcount` and `ccount`. Will raise ValueError if both stride and count are used.

**The `rcount` and `ccount` kwargs supersedes `rstride` and `cstride` for default sampling method for wireframe plotting. These arguments will determine at most how many evenly spaced samples will be taken from the input data to generate the graph. This is the default sampling method unless using the 'classic' style. Will raise ValueError if both stride and count are specified. If either is zero, then the input data is not sampled along this direction, producing a 3D line plot rather than a wireframe plot. Added in v2.0.0.**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>X, Y, Z</code></td>
<td>Data values as 2D arrays</td>
</tr>
<tr>
<td><code>rstride</code></td>
<td>Array row stride (step size), defaults to 1</td>
</tr>
<tr>
<td><code>cstride</code></td>
<td>Array column stride (step size), defaults to 1</td>
</tr>
<tr>
<td><code>rcount</code></td>
<td>Use at most this many rows, defaults to 50</td>
</tr>
<tr>
<td><code>ccount</code></td>
<td>Use at most this many columns, defaults to 50</td>
</tr>
</tbody>
</table>

Keyword arguments are passed on to `LineCollection`.

Returns a `Line3DCollection`.
70.1.5 Surface plots

```
Axes3D.plot_surface(X, Y, Z, *args, **kwargs)
```

Create a surface plot.

By default it will be colored in shades of a solid color, but it also supports color mapping by supplying the `cmap` argument.

The `rstride` and `cstride` keywords set the stride used to sample the input data to generate the graph. If 1k by 1k arrays are passed in, the default values for the strides will result in a 100x100 grid being plotted. Defaults to 10. Raises a `ValueError` if both stride and count are specified.

The `rcount` and `ccount` keywords supersede `rstride` and `cstride` for default sampling method for surface plotting. These arguments will determine at most how many evenly spaced samples will be taken from the input data to generate the graph. This is the default sampling method unless using the 'classic' style. Will raise `ValueError` if both stride and count are specified. Added in v2.0.0.
<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, Y, Z</td>
<td>Data values as 2D arrays</td>
</tr>
<tr>
<td>rstride</td>
<td>Array row stride (step size)</td>
</tr>
<tr>
<td>cstride</td>
<td>Array column stride (step size)</td>
</tr>
<tr>
<td>rcount</td>
<td>Use at most this many rows, defaults to 50</td>
</tr>
<tr>
<td>ccount</td>
<td>Use at most this many columns, defaults to 50</td>
</tr>
<tr>
<td>color</td>
<td>Color of the surface patches</td>
</tr>
<tr>
<td>cmap</td>
<td>A colormap for the surface patches</td>
</tr>
<tr>
<td>facecolors</td>
<td>Face colors for the individual patches</td>
</tr>
<tr>
<td>norm</td>
<td>An instance of Normalize to map values to colors</td>
</tr>
<tr>
<td>vmin</td>
<td>Minimum value to map</td>
</tr>
<tr>
<td>vmax</td>
<td>Maximum value to map</td>
</tr>
<tr>
<td>shade</td>
<td>Whether to shade the facecolors</td>
</tr>
</tbody>
</table>

Other arguments are passed on to *Poly3DCollection*
70.1.6 Tri-Surface plots

Axes3D.plot_trisurf(*args, **kwargs)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, Y, Z</td>
<td>Data values as 1D arrays</td>
</tr>
<tr>
<td>color</td>
<td>Color of the surface patches</td>
</tr>
<tr>
<td>cmap</td>
<td>A colormap for the surface patches.</td>
</tr>
<tr>
<td>norm</td>
<td>An instance of Normalize to map values to colors</td>
</tr>
<tr>
<td>vmin</td>
<td>Minimum value to map</td>
</tr>
<tr>
<td>vmax</td>
<td>Maximum value to map</td>
</tr>
<tr>
<td>shade</td>
<td>Whether to shade the facecolors</td>
</tr>
</tbody>
</table>

The (optional) triangulation can be specified in one of two ways; either:

```python
plot_trisurf(triangulation, ...)
```

where triangulation is a `Triangulation` object, or:

```python
plot_trisurf(X, Y, ...)
plot_trisurf(X, Y, triangles, ...)
plot_trisurf(X, Y, triangles=triangles, ...)
```

in which case a Triangulation object will be created. See `Triangulation` for a explanation of these
possibilities.
The remaining arguments are:

```python
plot_trisurf(..., Z)
```

where Z is the array of values to contour, one per point in the triangulation.

Other arguments are passed on to `Poly3DCollection`

**Examples:**

![3D Surface Plot](image-url)
New in version 1.2.0: This plotting function was added for the v1.2.0 release.
70.1.7 Contour plots

Axes3D.contour(X, Y, Z, *args, **kwargs)
Create a 3D contour plot.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, Y, Z</td>
<td>Data values as numpy.arrays</td>
</tr>
<tr>
<td>extend3d</td>
<td>Whether to extend contour in 3D (default: False)</td>
</tr>
<tr>
<td>stride</td>
<td>Stride (step size) for extending contour</td>
</tr>
<tr>
<td>zdir</td>
<td>The direction to use: x, y or z (default)</td>
</tr>
<tr>
<td>offset</td>
<td>If specified plot a projection of the contour lines on this position in plane normal to zdir</td>
</tr>
</tbody>
</table>

The positional and other keyword arguments are passed on to contour()

Returns a contour
70.1.8 Filled contour plots

Axes3D.contourf(X, Y, Z, *args, **kwargs)
Create a 3D contourf plot.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, Y, Z</td>
<td>Data values as numpy.arrays</td>
</tr>
<tr>
<td>zdir</td>
<td>The direction to use: x, y or z (default)</td>
</tr>
<tr>
<td>offset</td>
<td>If specified plot a projection of the filled contour on this position in plane normal to zdir</td>
</tr>
</tbody>
</table>

The positional and keyword arguments are passed on to contourf()

Returns a contourf

Changed in version 1.1.0: The zdir and offset kwargs were added.
New in version 1.1.0: The feature demoed in the second contourf3d example was enabled as a result of a bugfix for version 1.1.0.

### 70.1.9 Polygon plots

```python
Axes3D.add_collection3d(col, zs=0, zdir='z')
```

Add a 3D collection object to the plot.

2D collection types are converted to a 3D version by modifying the object and adding z coordinate information.

**Supported are:**

- PolyCollection
- LineCollection
- PatchCollection
70.1.10 Bar plots

Axes3D.bar(left, height, zs=0, zdir=’z’, *args, **kwargs)
Add 2D bar(s).

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>left</td>
<td>The x coordinates of the left sides of the bars.</td>
</tr>
<tr>
<td>height</td>
<td>The height of the bars.</td>
</tr>
<tr>
<td>zs</td>
<td>Z coordinate of bars, if one value is specified they will all be placed at the same z.</td>
</tr>
<tr>
<td>zdir</td>
<td>Which direction to use as z (‘x’, ‘y’ or ‘z’) when plotting a 2D set.</td>
</tr>
</tbody>
</table>

Keyword arguments are passed onto bar().

Returns a Patch3DCollection
70.1.11 Quiver

Axes3D.quiver(*args, **kwargs)

Plot a 3D field of arrows.

call signatures:

```
quiver(X, Y, Z, U, V, W, **kwargs)
```

Arguments:

- **X, Y, Z**: The x, y and z coordinates of the arrow locations (default is tail of arrow; see `pivot` kwarg)
- **U, V, W**: The x, y and z components of the arrow vectors

The arguments could be array-like or scalars, so long as they can be broadcast together. The arguments can also be masked arrays. If an element in any of argument is masked, then that corresponding quiver element will not be plotted.

Keyword arguments:

- **length**: [1.0 | float] The length of each quiver, default to 1.0, the unit is the same with the axes
**arrow_length_ratio**: [0.3 | float] The ratio of the arrow head with respect to the quiver, default to 0.3

**pivot**: ['tail' | 'middle' | 'tip'] The part of the arrow that is at the grid point; the arrow rotates about this point, hence the name pivot. Default is 'tail'

**normalize**: [False | True] When True, all of the arrows will be the same length. This defaults to False, where the arrows will be different lengths depending on the values of u,v,w.

Any additional keyword arguments are delegated to LineCollection
70.1.12 2D plots in 3D

70.1.13 Text

Axes3D.text(x, y, z, zdir=None, **kwargs)

Add text to the plot. kwargs will be passed on to Axes.text, except for the zdir keyword, which sets the direction to be used as the z direction.
70.1.14 Subplotting

Having multiple 3D plots in a single figure is the same as it is for 2D plots. Also, you can have both 2D and 3D plots in the same figure.

New in version 1.0.0: Subplotting 3D plots was added in v1.0.0. Earlier version can not do this.
Chapter 70. Matplotlib mplot3d toolkit
A tale of 2 subplots

70.2 mplot3d API
Contents

- mplot3d API
  - axes3d
  - axis3d
  - art3d
  - proj3d

70.2.1 axes3d

Note: Significant effort went into bringing axes3d to feature-parity with regular axes objects for version 1.1.0. However, more work remains. Please report any functions that do not behave as expected as a bug. In addition, help and patches would be greatly appreciated!

Module containing Axes3D, an object which can plot 3D objects on a 2D matplotlib figure.

class mpl_toolkits.mplot3d.axes3d.Axes3D(fig, rect=None, *args, **kwargs)

3D axes object.

add_collection3d(col, zs=0, zdir='z')
Add a 3D collection object to the plot.

2D collection types are converted to a 3D version by modifying the object and adding z coordinate information.

Supported are:

- PolyCollection
- LineCollection
- PatchCollection

add_contour_set(cset, extend3d=False, stride=5, zdir='z', offset=None)

add_contourf_set(cset, zdir='z', offset=None)

auto_scale_xyz(X, Y, Z=None, had_data=None)

autoscale(enable=True, axis='both', tight=None)

Convenience method for simple axis view autoscaling. See matplotlib.axes.Axes.autoscale() for full explanation. Note that this function behaves the same, but for all three axes. Therefore, ‘z’ can be passed for axis, and ‘both’ applies to all three axes.
New in version 1.1.0: This function was added, but not tested. Please report any bugs.

**autoscale_view(tight=None, scalex=True, scaley=True, scalez=True)**

Autoscale the view limits using the data limits. See `matplotlib.axes.Axes.autoscale_view()` for documentation. Note that this function applies to the 3D axes, and as such adds the `scalez` to the function arguments.

Changed in version 1.1.0: Function signature was changed to better match the 2D version. `tight` is now explicitly a kwarg and placed first.

Changed in version 1.2.1: This is now fully functional.

**bar(left, height, zs=0, zdir='z', *args, **kwargs)**

Add 2D bar(s).

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>left</code></td>
<td>The x coordinates of the left sides of the bars.</td>
</tr>
<tr>
<td><code>height</code></td>
<td>The height of the bars.</td>
</tr>
<tr>
<td><code>zs</code></td>
<td>Z coordinate of bars, if one value is specified they will all be placed at the same z.</td>
</tr>
<tr>
<td><code>zdir</code></td>
<td>Which direction to use as z (‘x’, ‘y’ or ‘z’) when plotting a 2D set.</td>
</tr>
</tbody>
</table>

Keyword arguments are passed onto `bar()`.

Returns a `Patch3DCollection`

**bar3d(x, y, z, dx, dy, dz, color=None, zsort='average', *args, **kwargs)**

Generate a 3D bar, or multiple bars.

When generating multiple bars, x, y, z have to be arrays. dx, dy, dz can be arrays or scalars. `color` can be:

- A single color value, to color all bars the same color.
- An array of colors of length N bars, to color each bar independently.
- An array of colors of length 6, to color the faces of the bars similarly.
- An array of colors of length 6 * N bars, to color each face independently.

When coloring the faces of the boxes specifically, this is the order of the coloring:

1.-Z (bottom of box)
2.+Z (top of box)
3.-Y
4.+Y
5.-X
6.+X

Keyword arguments are passed onto `Poly3DCollection()`
**can_pan()**
Return True if this axes supports the pan/zoom button functionality.
3D axes objects do not use the pan/zoom button.

**can_zoom()**
Return True if this axes supports the zoom box button functionality.
3D axes objects do not use the zoom box button.

**cla()**
Clear axes

**clabel(*args, **kwargs)**
This function is currently not implemented for 3D axes. Returns None.

**contour(X, Y, Z, *args, **kwargs)**
Create a 3D contour plot.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, Y, Z</td>
<td>Data values as numpy.arrays</td>
</tr>
<tr>
<td>extend3d</td>
<td>Whether to extend contour in 3D (default: False)</td>
</tr>
<tr>
<td>stride</td>
<td>Stride (step size) for extending contour</td>
</tr>
<tr>
<td>zdir</td>
<td>The direction to use: x, y or z (default)</td>
</tr>
<tr>
<td>offset</td>
<td>If specified plot a projection of the contour lines on this position in plane normal to zdir</td>
</tr>
</tbody>
</table>

The positional and other keyword arguments are passed on to `contour()`

Returns a `contour`

**contour3D(X, Y, Z, *args, **kwargs)**
Create a 3D contour plot.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, Y, Z</td>
<td>Data values as numpy.arrays</td>
</tr>
<tr>
<td>extend3d</td>
<td>Whether to extend contour in 3D (default: False)</td>
</tr>
<tr>
<td>stride</td>
<td>Stride (step size) for extending contour</td>
</tr>
<tr>
<td>zdir</td>
<td>The direction to use: x, y or z (default)</td>
</tr>
<tr>
<td>offset</td>
<td>If specified plot a projection of the contour lines on this position in plane normal to zdir</td>
</tr>
</tbody>
</table>

The positional and other keyword arguments are passed on to `contour()`

Returns a `contour`

**contourf(X, Y, Z, *args, **kwargs)**
Create a 3D contourf plot.
The positional and keyword arguments are passed on to `contourf()`

Returns a `contourf`.

Changed in version 1.1.0: The `zdir` and `offset` kwargs were added.

### contourf3D(X, Y, Z, *args, **kwargs)

Create a 3D contourf plot.

The positional and keyword arguments are passed on to `contourf()`

Returns a `contourf`.

Changed in version 1.1.0: The `zdir` and `offset` kwargs were added.

### convert_zunits(z)

For artists in an axes, if the zaxis has units support, convert z using zaxis unit type

New in version 1.2.1.

### disable_mouse_rotation()

Disable mouse button callbacks.

### draw(renderer)

### format_coord(xd, yd)

Given the 2D view coordinates attempt to guess a 3D coordinate. Looks for the nearest edge to the point and then assumes that the point is at the same z location as the nearest point on the edge.

### format_zdata(z)

Return z string formatted. This function will use the `fmt_zdata` attribute if it is callable, else will fall back on the zaxis major formatter.

### get_autoscale_on()

Get whether autoscaling is applied for all axes on plot commands

New in version 1.1.0: This function was added, but not tested. Please report any bugs.
get_autoscalez_on()
Get whether autoscaling for the z-axis is applied on plot commands

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

get_axis_position()

get_axisbelow()
Get whether axis below is true or not.
For axes3d objects, this will always be True

New in version 1.1.0: This function was added for completeness.

get_children()

get_frame_on()
Get whether the 3D axes panels are drawn

New in version 1.1.0.

get_proj()
Create the projection matrix from the current viewing position.
elev stores the elevation angle in the z plane
azim stores the azimuth angle in the x,y plane
dist is the distance of the eye viewing point from the object point.

get_w_lims()
Get 3D world limits.

get_xlim()
Get the x-axis range

Returns xlimits : tuple
Returns the current x-axis limits as the tuple (left, right).

Notes

The x-axis may be inverted, in which case the left value will be greater than the right value.

Changed in version 1.1.0: This function now correctly refers to the 3D x-limits

get_xlim3d()
Get the x-axis range

Returns xlimits : tuple
Returns the current x-axis limits as the tuple (left, right).
Notes

The x-axis may be inverted, in which case the left value will be greater than the right value.

Changed in version 1.1.0: This function now correctly refers to the 3D x-limits

get_xlim()
Get the x-axis range

Returns ylimits : tuple
    Returns the current y-axis limits as the tuple (bottom, top).

Notes

The y-axis may be inverted, in which case the bottom value will be greater than the top value.

Changed in version 1.1.0: This function now correctly refers to the 3D y-limits.

get_ylim3d()
Get the y-axis range

Returns ylimits : tuple
    Returns the current y-axis limits as the tuple (bottom, top).

Notes

The y-axis may be inverted, in which case the bottom value will be greater than the top value.

Changed in version 1.1.0: This function now correctly refers to the 3D y-limits.

get_zbound()
Returns the z-axis numerical bounds where:

lowerBound < upperBound

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

get_zlabel()
Get the z-label text string.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

get_zlim()
Get 3D z limits.

get_zlim3d()
Get 3D z limits.

get_zmajorticklabels()
Get the ztick labels as a list of Text instances

New in version 1.1.0.
get_zminorticklabels()
Get the ztick labels as a list of Text instances

Note: Minor ticks are not supported. This function was added only for completeness.
New in version 1.1.0.

get_zscale()

get_zticklabels(minor=False)
Get ztick labels as a list of Text instances. See matplotlib.axes.Axes.get_yticklabels() for more details.

Note: Minor ticks are not supported.
New in version 1.1.0.

get_zticklines()
Get ztick lines as a list of Line2D instances. Note that this function is provided merely for completeness. These lines are re-calculated as the display changes.
New in version 1.1.0.

get_zticks(minor=False)
Return the z ticks as a list of locations See matplotlib.axes.Axes.get_yticks() for more details.

Note: Minor ticks are not supported.
New in version 1.1.0.

grid(b=True, **kwargs)
Set / unset 3D grid.

Note: Currently, this function does not behave the same as matplotlib.axes.Axes.grid(), but it is intended to eventually support that behavior.
Changed in version 1.1.0: This function was changed, but not tested. Please report any bugs.

have_units()
Return True if units are set on the x, y, or z axes

invert_zaxis()
Invert the z-axis.
New in version 1.1.0: This function was added, but not tested. Please report any bugs.
**locator_params***(axis='both', tight=None, **kwargs)***
Convenience method for controlling tick locators.

See matplotlib.axes.Axes.locator_params() for full documentation Note that this is for Axes3D objects, therefore, setting axis to ‘both’ will result in the parameters being set for all three axes. Also, axis can also take a value of ‘z’ to apply parameters to the z axis.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

**margins**(*args, **kw)*
Convenience method to set or retrieve autoscaling margins.

**signatures:** margins()
returns xmargin, ymargin, zmargin

margins(margin)
margins(xmargin, ymargin, zmargin)
margins(x=xmargin, y=ymargin, z=zmargin)
margins(..., tight=False)

All forms above set the xmargin, ymargin and zmargin parameters. All keyword parameters are optional. A single argument specifies xmargin, ymargin and zmargin. The tight parameter is passed to autoscale_view(), which is executed after a margin is changed; the default here is True, on the assumption that when margins are specified, no additional padding to match tick marks is usually desired. Setting tight to None will preserve the previous setting.

Specifying any margin changes only the autoscaling; for example, if xmargin is not None, then xmargin times the X data interval will be added to each end of that interval before it is used in autoscaling.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

**mouse_init**(rotate_btn=1, zoom_btn=3)
Initializes mouse button callbacks to enable 3D rotation of the axes. Also optionally sets the mouse buttons for 3D rotation and zooming.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rotate_btn</td>
<td>The integer or list of integers specifying which mouse button or buttons to use for 3D rotation of the axes. Default = 1.</td>
</tr>
<tr>
<td>zoom_btn</td>
<td>The integer or list of integers specifying which mouse button or buttons to use to zoom the 3D axes. Default = 3.</td>
</tr>
</tbody>
</table>

**name** = ‘3d’

**plot**(xs, ys, *args, **kwargs)
Plot 2D or 3D data.
<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xs, ys</td>
<td>x, y coordinates of vertices</td>
</tr>
<tr>
<td>zs</td>
<td>z value(s), either one for all points or one for each point.</td>
</tr>
<tr>
<td>zdir</td>
<td>Which direction to use as z (‘x’, ‘y’ or ‘z’) when plotting a 2D set.</td>
</tr>
</tbody>
</table>

Other arguments are passed on to plot()

**plot3D**(xs, ys, *args, **kwargs)

Plot 2D or 3D data.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xs, ys</td>
<td>x, y coordinates of vertices</td>
</tr>
<tr>
<td>zs</td>
<td>z value(s), either one for all points or one for each point.</td>
</tr>
<tr>
<td>zdir</td>
<td>Which direction to use as z (‘x’, ‘y’ or ‘z’) when plotting a 2D set.</td>
</tr>
</tbody>
</table>

Other arguments are passed on to plot()

**plot_surface**(X, Y, Z, *args, **kwargs)

Create a surface plot.

By default it will be colored in shades of a solid color, but it also supports color mapping by supplying the cmap argument.

The rstride and cstride kwargs set the stride used to sample the input data to generate the graph. If 1k by 1k arrays are passed in, the default values for the strides will result in a 100x100 grid being plotted. Defaults to 10. Raises a ValueError if both stride and count kwargs (see next section) are provided.

The rcount and ccount kwargs supersedes rstride and cstride for default sampling method for surface plotting. These arguments will determine at most how many evenly spaced samples will be taken from the input data to generate the graph. This is the default sampling method unless using the ‘classic’ style. Will raise ValueError if both stride and count are specified. Added in v2.0.0.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, Y, Z</td>
<td>Data values as 2D arrays</td>
</tr>
<tr>
<td>rstride</td>
<td>Array row stride (step size)</td>
</tr>
</tbody>
</table>
cstride  | Array column stride (step size) |
|rcount   | Use at most this many rows, defaults to 50 |
ccount   | Use at most this many columns, defaults to 50 |
color    | Color of the surface patches |
cmap     | A colormap for the surface patches. |
facecolors | Face colors for the individual patches |
norm     | An instance of Normalize to map values to colors |
vmin     | Minimum value to map |
vmax     | Maximum value to map |
shade    | Whether to shade the facecolors |

Other arguments are passed on to Poly3DCollection

**plot_trisurf**( *args, **kwargs)
<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, Y, Z</td>
<td>Data values as 1D arrays</td>
</tr>
<tr>
<td>color</td>
<td>Color of the surface patches</td>
</tr>
<tr>
<td>cmap</td>
<td>A colormap for the surface patches.</td>
</tr>
<tr>
<td>norm</td>
<td>An instance of Normalize to map values to colors</td>
</tr>
<tr>
<td>vmin</td>
<td>Minimum value to map</td>
</tr>
<tr>
<td>vmax</td>
<td>Maximum value to map</td>
</tr>
<tr>
<td>shade</td>
<td>Whether to shade the facecolors</td>
</tr>
</tbody>
</table>

The (optional) triangulation can be specified in one of two ways; either:

```python
plot_trisurf(triangulation, ...)
```

where triangulation is a `Triangulation` object, or:

```python
plot_trisurf(X, Y, ...)  
plot_trisurf(X, Y, triangles, ...)  
plot_trisurf(X, Y, triangles=triangles, ...)
```

in which case a Triangulation object will be created. See `Triangulation` for a explanation of these possibilities.

The remaining arguments are:

```python
plot_trisurf(..., Z)
```

where Z is the array of values to contour, one per point in the triangulation.

Other arguments are passed on to `Poly3DCollection`

**Examples:**
New in version 1.2.0: This plotting function was added for the v1.2.0 release.

```
plot_wireframe(X, Y, Z, *args, **kwargs)
```

Plot a 3D wireframe.

The rstride and cstride kwargs set the stride used to sample the input data to
generate the graph. If either is 0 the input data in not sampled along this direction producing a 3D line plot rather than a wireframe plot. The stride arguments are only used by default if in the ‘classic’ mode. They are now superseded by rcount and ccount. Will raise ValueError if both stride and count are used.

The rcount and ccount kwargs supersedes rstride and cstride for default sampling method for wireframe plotting. These arguments will determine at most how many evenly spaced samples will be taken from the input data to generate the graph. This is the default sampling method unless using the ‘classic’ style. Will raise ValueError if both stride and count are specified. If either is zero, then the input data is not sampled along this direction, producing a 3D line plot rather than a wireframe plot. Added in v2.0.0.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, Y, Z</td>
<td>Data values as 2D arrays</td>
</tr>
<tr>
<td>rstride</td>
<td>Array row stride (step size), defaults to 1</td>
</tr>
<tr>
<td>cstride</td>
<td>Array column stride (step size), defaults to 1</td>
</tr>
<tr>
<td>rcount</td>
<td>Use at most this many rows, defaults to 50</td>
</tr>
<tr>
<td>ccount</td>
<td>Use at most this many columns, defaults to 50</td>
</tr>
</tbody>
</table>

Keyword arguments are passed on to LineCollection.

Returns a Line3DCollection

quiver(*args, **kwargs)

Plot a 3D field of arrows.

call signatures:

| quiver(X, Y, Z, U, V, W, **kwargs) |

Arguments:

- X, Y, Z: The x, y and z coordinates of the arrow locations (default is tail of arrow; see pivot kwarg)
- U, V, W: The x, y and z components of the arrow vectors

The arguments could be array-like or scalars, so long as they they can be broadcast together. The arguments can also be masked arrays. If an element in any of argument is masked, then that corresponding quiver element will not be plotted.

Keyword arguments:

- length: [1.0 | float] The length of each quiver, default to 1.0, the unit is the same with the axes
- arrow_length_ratio: [0.3 | float] The ratio of the arrow head with respect to the quiver, default to 0.3
- pivot: [‘tail’ | ‘middle’ | ‘tip’] The part of the arrow that is at the grid point; the arrow rotates about this point, hence the name pivot. Default is ‘tail’
**normalize**: [**False** | **True**] When True, all of the arrows will be the same length. This defaults to False, where the arrows will be different lengths depending on the values of u,v,w.

Any additional keyword arguments are delegated to `LineCollection`

**quiver3D**(*args, **kwargs)
Plot a 3D field of arrows.

call signatures:

```
quiver(X, Y, Z, U, V, W, **kwargs)
```

Arguments:

- *X, Y, Z*: The x, y and z coordinates of the arrow locations (default is tail of arrow; see `pivot` kwarg)
- *U, V, W*: The x, y and z components of the arrow vectors

The arguments could be array-like or scalars, so long as they can be broadcast together. The arguments can also be masked arrays. If an element in any of argument is masked, then that corresponding quiver element will not be plotted.

Keyword arguments:

- **length**: [1.0 | float] The length of each quiver, default to 1.0, the unit is the same with the axes
- **arrow_length_ratio**: [0.3 | float] The ratio of the arrow head with respect to the quiver, default to 0.3
- **pivot**: [‘tail’ | ‘middle’ | ‘tip’] The part of the arrow that is at the grid point; the arrow rotates about this point, hence the name `pivot`. Default is ‘tail’
- **normalize**: [**False** | **True**] When True, all of the arrows will be the same length. This defaults to False, where the arrows will be different lengths depending on the values of u,v,w.

Any additional keyword arguments are delegated to `LineCollection`

**scatter**(*xs, ys, zs=0, zdir='z', s=20, c=None, depthshade=True, **kwargs)
Create a scatter plot.
<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>xs, ys</code></td>
<td>Positions of data points.</td>
</tr>
<tr>
<td><code>zs</code></td>
<td>Either an array of the same length as <code>xs</code> and <code>ys</code> or a single value to place all points in the same plane. Default is 0.</td>
</tr>
<tr>
<td><code>zdir</code></td>
<td>Which direction to use as z (‘x’, ‘y’ or ‘z’) when plotting a 2D set.</td>
</tr>
<tr>
<td><code>s</code></td>
<td>Size in points^2. It is a scalar or an array of the same length as <code>x</code> and <code>y</code>.</td>
</tr>
<tr>
<td><code>c</code></td>
<td>A color. <code>c</code> can be a single color format string, or a sequence of color specifications of length <code>N</code>, or a sequence of <code>N</code> numbers to be mapped to colors using the <code>cmap</code> and <code>norm</code> specified via kwargs (see below). Note that <code>c</code> should not be a single numeric RGB or RGBA sequence because that is indistinguishable from an array of values to be colormapped. <code>c</code> can be a 2-D array in which the rows are RGB or RGBA, however, including the case of a single row to specify the same color for all points.</td>
</tr>
<tr>
<td><code>depthshade</code></td>
<td>Whether or not to shade the scatter markers to give the appearance of depth. Default is <code>True</code>.</td>
</tr>
</tbody>
</table>

Keyword arguments are passed on to `scatter()`. Returns a `Patch3DCollection`

`scatter3D(xs, ys, zs=0, zdir='z', s=20, c=None, depthshade=True, *args, **kwargs)`
Create a scatter plot.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>xs, ys</code></td>
<td>Positions of data points.</td>
</tr>
<tr>
<td><code>zs</code></td>
<td>Either an array of the same length as <code>xs</code> and <code>ys</code> or a single value to place all points in the same plane. Default is 0.</td>
</tr>
<tr>
<td><code>zdir</code></td>
<td>Which direction to use as z (‘x’, ‘y’ or ‘z’) when plotting a 2D set.</td>
</tr>
<tr>
<td><code>s</code></td>
<td>Size in points^2. It is a scalar or an array of the same length as <code>x</code> and <code>y</code>.</td>
</tr>
<tr>
<td><code>c</code></td>
<td>A color. <code>c</code> can be a single color format string, or a sequence of color specifications of length <code>N</code>, or a sequence of <code>N</code> numbers to be mapped to colors using the <code>cmap</code> and <code>norm</code> specified via kwargs (see below). Note that <code>c</code> should not be a single numeric RGB or RGBA sequence because that is indistinguishable from an array of values to be colormapped. <code>c</code> can be a 2-D array in which the rows are RGB or RGBA, however, including the case of a single row to specify the same color for all points.</td>
</tr>
<tr>
<td><code>depthshade</code></td>
<td>Whether or not to shade the scatter markers to give the appearance of depth. Default is <code>True</code>.</td>
</tr>
</tbody>
</table>

Keyword arguments are passed on to `scatter()`. Returns a `Patch3DCollection`

`set_autoscale_on(b)`
Set whether autoscaling is applied on plot commands

accepts: [ `True` | `False` ]

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

`set_autoscalez_on(b)`
Set whether autoscaling for the z-axis is applied on plot commands
accepts: [True | False ]

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

```python
set_axis_off()
```

```python
set_axis_on()
```

```python
set_axisbelow(b)
```

Set whether the axis ticks and gridlines are above or below most artists

For axes3d objects, this will ignore any settings and just use True

ACCEPSTS: [True | False ]

New in version 1.1.0: This function was added for completeness.

```python
set_frame_on(b)
```

Set whether the 3D axes panels are drawn

ACCEPSTS: [True | False ]

New in version 1.1.0.

```python
set_title(label, fontdict=None, loc='center', **kwargs)
```

Set a title for the axes.

Set one of the three available axes titles. The available titles are positioned above the axes in the center, flush with the left edge, and flush with the right edge.

**Parameters**

- `label`: str
  - Text to use for the title
- `fontdict`: dict
  - A dictionary controlling the appearance of the title text, the default fontdict is:

```python
{'fontsize': rcParams['axes.titlesize'],
 'fontweight': rcParams['axes.titleweight'],
 'verticalalignment': 'baseline',
 'horizontalalignment': 'loc'}
```

- `loc`: {'center', 'left', 'right'}, str, optional
  - Which title to set, defaults to ‘center’

**Returns**

- `text`: Text
  - The matplotlib text instance representing the title

**Other Parameters**

- `kwargs`: text properties
  - Other keyword arguments are text properties, see Text for a list of valid text properties.
**set_top_view()**

**set_xlim**(left=None, right=None, emit=True, auto=False, **kw)
Set 3D x limits.

See `matplotlib.axes.Axes.set_xlim()` for full documentation.

**set_xlim3d**(left=None, right=None, emit=True, auto=False, **kw)
Set 3D x limits.

See `matplotlib.axes.Axes.set_xlim()` for full documentation.

**set_xscale**(value, **kwargs)
Set the x-axis scale

Set the scaling of the x-axis: ‘linear’ | ‘log’ | ‘logit’ | ‘symlog’

ACCEPTS: [‘linear’ | ‘log’ | ‘logit’ | ‘symlog’]

**Different kwargs are accepted, depending on the scale:**

‘linear’

‘log’

*basex/basey*: The base of the logarithm

*nonposx/nonposy*: [‘mask’ | ‘clip’] non-positive values in x or y can be masked as invalid, or clipped to a very small positive number

*subsx/subsy*: Where to place the subticks between each major tick. Should be a sequence of integers. For example, in a log10 scale: [2, 3, 4, 5, 6, 7, 8, 9]

will place 8 logarithmically spaced minor ticks between each major tick.

‘logit’

*nonpos*: [‘mask’ | ‘clip’] values beyond ]0, 1[ can be masked as invalid, or clipped to a number very close to 0 or 1

‘symlog’

*basex/basey*: The base of the logarithm

*linthreshx/linthreshy*: The range (-x, x) within which the plot is linear (to avoid having the plot go to infinity around zero).

*subsx/subsy*: Where to place the subticks between each major tick. Should be a sequence of integers. For example, in a log10 scale: [2, 3, 4, 5, 6, 7, 8, 9]

will place 8 logarithmically spaced minor ticks between each major tick.

*linscalex/linscaley*: This allows the linear range (-linthresh to linthresh) to be stretched relative to the logarithmic range. Its value is the number of decades to use for each half of the linear range. For example, when `linscale == 1.0` (the default), the space used for the positive and negative halves of the linear range will be equal to one decade in the logarithmic range.
New in version 1.1.0: This function was added, but not tested. Please report any bugs.

```
set_ylim(bottom=None, top=None, emit=True, auto=False, **kw)
```

Set 3D y limits.

See `matplotlib.axes.Axes.set_ylim()` for full documentation.

```
set_ylim3d(bottom=None, top=None, emit=True, auto=False, **kw)
```

Set 3D y limits.

See `matplotlib.axes.Axes.set_ylim()` for full documentation.

```
set_yscale(value, **kwargs)
```

Set the y-axis scale

Set the scaling of the y-axis: ‘linear’ | ‘log’ | ‘logit’ | ‘symlog’

ACCEPTS: [‘linear’ | ‘log’ | ‘logit’ | ‘symlog’]

Different kwargs are accepted, depending on the scale: ‘linear’

‘log’

`base`x/`base`y: The base of the logarithm

`nonpos`x/`nonpos`y: [‘mask’ | ‘clip’] non-positive values in x or y can be masked as invalid, or clipped to a very small positive number

`sub`x/`sub`y: Where to place the subticks between each major tick. Should be a sequence of integers. For example, in a log10 scale: [2, 3, 4, 5, 6, 7, 8, 9]

will place 8 logarithmically spaced minor ticks between each major tick.

‘logit’

`nonpos`: [‘mask’ | ‘clip’] values beyond [0, 1] can be masked as invalid, or clipped to a number very close to 0 or 1

‘symlog’

`base`x/`base`y: The base of the logarithm

`linthresh`x/`linthresh`y: The range (-x, x) within which the plot is linear (to avoid having the plot go to infinity around zero).

`sub`x/`sub`y: Where to place the subticks between each major tick. Should be a sequence of integers. For example, in a log10 scale: [2, 3, 4, 5, 6, 7, 8, 9]

will place 8 logarithmically spaced minor ticks between each major tick.

`linscale`x/`linscale`y: This allows the linear range (-linthresh to linthresh) to be stretched relative to the logarithmic range. Its value is the number of decades to use for each half of the linear range. For example, when `linscale` == 1.0 (the default), the space used for the positive and negative halves of the linear range will be equal to one decade in the logarithmic range.
New in version 1.1.0: This function was added, but not tested. Please report any bugs.

**set_zbound**(lower=None, upper=None)
Set the lower and upper numerical bounds of the z-axis. This method will honor axes inversion regardless of parameter order. It will not change the _autoscaleZon attribute.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

**set_zlabel**(zlabel, fontdict=None, labelpad=None, **kwargs)
Set zlabel. See doc for set_ylabel() for description.

**set_zlim**(bottom=None, top=None, emit=True, auto=False, **kw)
Set 3D z limits.
See matplotlib.axes.Axes.set_ylim() for full documentation

**set_zlim3d**(bottom=None, top=None, emit=True, auto=False, **kw)
Set 3D z limits.
See matplotlib.axes.Axes.set_ylim() for full documentation

**set_zmargin**(m)
Set padding of Z data limits prior to autoscaling.

\( m \) times the data interval will be added to each end of that interval before it is used in autoscaling.

accepts: float in range 0 to 1

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

**set_zscale**(value, **kwargs)
Set the scaling of the z-axis: 'linear' | 'log' | 'logit' | 'symlog'

ACCEPTS: ['linear' | 'log' | 'logit' | 'symlog']

**Different kwargs are accepted, depending on the scale:** 'linear'

'log'

\( basex/basey: \) The base of the logarithm

\( nonposx/nonposy: [\text{'mask'} | \text{'clip'}] \) non-positive values in x or y can be masked as invalid, or clipped to a very small positive number

\( subsx/subsy: \) Where to place the subticks between each major tick. Should be a sequence of integers. For example, in a log10 scale: [2, 3, 4, 5, 6, 7, 8, 9]

will place 8 logarithmically spaced minor ticks between each major tick.

'logit'

\( nonpos: [\text{'mask'} | \text{'clip'}] \) values beyond ]0, 1[ can be masked as invalid, or clipped to a number very close to 0 or 1

'symlog'

\( basex/basey: \) The base of the logarithm
**linthreshx/linthreshy**: The range \((-x, x)\) within which the plot is linear (to avoid having the plot go to infinity around zero).

**subsx/subsy**: Where to place the subticks between each major tick. Should be a sequence of integers. For example, in a log10 scale: [2, 3, 4, 5, 6, 7, 8, 9]

will place 8 logarithmically spaced minor ticks between each major tick.

**linscalex/linscaley**: This allows the linear range (\(-\text{linthresh} to \text{linthresh}\)) to be stretched relative to the logarithmic range. Its value is the number of decades to use for each half of the linear range. For example, when \(\text{linscale} == 1.0\) (the default), the space used for the positive and negative halves of the linear range will be equal to one decade in the logarithmic range.

---

**Note**: Currently, Axes3D objects only supports linear scales. Other scales may or may not work, and support for these is improving with each release.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

**set_zticklabels**(*args, **kwargs*)

Set z-axis tick labels. See `matplotlib.axes.Axes.set_yticklabels()` for more details.

---

**Note**: Minor ticks are not supported by Axes3D objects.

New in version 1.1.0.

**set_zticks**(*args, **kwargs*)

Set z-axis tick locations. See `matplotlib.axes.Axes.set_yticks()` for more details.

---

**Note**: Minor ticks are not supported.

New in version 1.1.0.

**text**(*x, y, z, s, zdir=None, **kwargs*)

Add text to the plot. kwargs will be passed on to Axes.text, except for the zdir keyword, which sets the direction to be used as the z direction.

**text2D**(*x, y, s, fontdict=None, withdash=False, **kwargs*)

Add text to the axes.

Add text in string `s` to axis at location `x`, `y`, data coordinates.

**Parameters**

- **x**, **y** : scalars
  data coordinates

- **s** : string
  text
fontdict : dictionary, optional, default: None

A dictionary to override the default text properties. If fontdict is None, the defaults are determined by your rc parameters.

withdash : boolean, optional, default: False

Creates a TextWithDash instance instead of a Text instance.

Other Parameters kwargs: Text properties.

Other miscellaneous text parameters.

Examples

Individual keyword arguments can be used to override any given parameter:

```python
>>> text(x, y, s, fontsize=12)
```

The default transform specifies that text is in data coords, alternatively, you can specify text in axis coords (0,0 is lower-left and 1,1 is upper-right). The example below places text in the center of the axes:

```python
>>> text(0.5, 0.5, 'matplotlib', horizontalalignment='center',
...       verticalalignment='center',
...       transform=ax.transAxes)
```

You can put a rectangular box around the text instance (e.g., to set a background color) by using the keyword bbox. bbox is a dictionary of Rectangle properties. For example:

```python
>>> text(x, y, s, bbox=dict(facecolor='red', alpha=0.5))
```

text3D(x, y, z, s, zdir=None, **kwargs)

Add text to the plot. kwargs will be passed on to Axes.text, except for the zdir keyword, which sets the direction to be used as the z direction.

tick_params(axis='both', **kwargs)

Convenience method for changing the appearance of ticks and tick labels.

See matplotlib.axes.Axes.tick_params() for more complete documentation.

The only difference is that setting axis to ‘both’ will mean that the settings are applied to all three axes. Also, the axis parameter also accepts a value of ‘z’, which would mean to apply to only the z-axis.

Also, because of how Axes3D objects are drawn very differently from regular 2D axes, some of these settings may have ambiguous meaning. For simplicity, the ‘z’ axis will accept settings as if it was like the ‘y’ axis.

**Note:** While this function is currently implemented, the core part of the Axes3D object may ignore some of these settings. Future releases will fix this. Priority will be given to those who
file bugs.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

**ticklabel_format(**kwargs)

Convenience method for manipulating the ScalarFormatter used by default for linear axes in Axes3D objects.

See `matplotlib.axes.Axes.ticklabel_format()` for full documentation. Note that this version applies to all three axes of the Axes3D object. Therefore, the `axis` argument will also accept a value of ‘z’ and the value of ‘both’ will apply to all three axes.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

**tricontour(**args, **kwargs)

Create a 3D contour plot.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, Y, Z</td>
<td>Data values as numpy.arrays</td>
</tr>
<tr>
<td>extend3d</td>
<td>Whether to extend contour in 3D (default: False)</td>
</tr>
<tr>
<td>stride</td>
<td>Stride (step size) for extending contour</td>
</tr>
<tr>
<td>zdir</td>
<td>The direction to use: x, y or z (default)</td>
</tr>
<tr>
<td>offset</td>
<td>If specified plot a projection of the contour lines on this position in plane normal to zdir</td>
</tr>
</tbody>
</table>

Other keyword arguments are passed on to `tricontour()`

Returns a `contour`

Changed in version 1.3.0: Added support for custom triangulations

EXPERIMENTAL: This method currently produces incorrect output due to a longstanding bug in 3D PolyCollection rendering.

**tricontourf(**args, **kwargs)

Create a 3D contourf plot.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, Y, Z</td>
<td>Data values as numpy.arrays</td>
</tr>
<tr>
<td>zdir</td>
<td>The direction to use: x, y or z (default)</td>
</tr>
<tr>
<td>offset</td>
<td>If specified plot a projection of the contour lines on this position in plane normal to zdir</td>
</tr>
</tbody>
</table>

Other keyword arguments are passed on to `tricontour()`

Returns a `contour`

Changed in version 1.3.0: Added support for custom triangulations
EXPERIMENTAL: This method currently produces incorrect output due to a longstanding bug in 3D PolyCollection rendering.

tunit_cube(vals=None, M=None)

tunit_edges(vals=None, M=None)

unit_cube(vals=None)

update_datalim(xys, **kwargs)

view_init(elev=None, azim=None)

Set the elevation and azimuth of the axes.

This can be used to rotate the axes programatically.

‘elev’ stores the elevation angle in the z plane. ‘azim’ stores the azimuth angle in the x,y plane.

if elev or azim are None (default), then the initial value is used which was specified in the Axes3D constructor.

zaxis_date(tz=None)

Sets up z-axis ticks and labels that treat the z data as dates.

tz is a timezone string or tzinfo instance. Defaults to rc value.

Note: This function is merely provided for completeness. Axes3D objects do not officially support dates for ticks, and so this may or may not work as expected.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

zaxis_inverted()

Returns True if the z-axis is inverted.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

mpl_toolkits.mplot3d.axes3d.get_test_data(delta=0.05)

Return a tuple X, Y, Z with a test data set.

mpl_toolkits.mplot3d.axes3d.unit_bbox()

70.2.2 axis3d

Note: Historically, axis3d has suffered from having hard-coded constants controlling the look and feel of the 3D plot. This precluded user level adjustments such as label spacing, font colors and panel colors.
For version 1.1.0, these constants have been consolidated into a single private member dictionary, self._axinfo, for the axis object. This is intended only as a stop-gap measure to allow user-level customization, but it is not intended to be permanent.

```python
class mpl_toolkits.mplot3d.axis3d.Axis(adir, v_intervalx, d_intervalx, axes, *args, **kwargs)
    Bases: matplotlib.axis.XAxis

draw(renderer)

draw_pane(renderer)

get_major_ticks(numticks=None)

get_rotate_label(text)

get_tick_positions()

get_tightbbox(renderer)

get_view_interval()
    return the Interval instance for this 3d axis view limits

init3d()

set_pane_color(color)
    Set pane color to a RGBA tuple

set_pane_pos(xys)

set_rotate_label(val)
    Whether to rotate the axis label: True, False or None. If set to None the label will be rotated if longer than 4 chars.

set_view_interval(vmin, vmax, ignore=False)
```

```python
class mpl_toolkits.mplot3d.axis3d.XAxis(adir, v_intervalx, d_intervalx, axes, *args, **kwargs)
    Bases: mpl_toolkits.mplot3d.axis3d.Axis

generate_data_interval()
    return the Interval instance for this axis data limits

class mpl_toolkits.mplot3d.axis3d.YAxis(adir, v_intervalx, d_intervalx, axes, *args, **kwargs)
    Bases: mpl_toolkits.mplot3d.axis3d.Axis
```
get_data_interval()
  return the Interval instance for this axis data limits

class mpl_toolkits.mplot3d.axis3d.ZAxis(adir, v_intervalx, d_intervalx, axes, *args, **kwargs)
  Bases: mpl_toolkits.mplot3d.axis3d.Axis

  get_data_interval()
  return the Interval instance for this axis data limits

mpl_toolkits.mplot3d.axis3d.get_flip_min_max(coord, index, mins, maxs)

mpl_toolkits.mplot3d.axis3d.move_from_center(coord, centers, deltas, axmask=(True, True, True))
  Return a coordinate that is moved by “deltas” away from the center.

mpl_toolkits.mplot3d.axis3d.tick_update_position(tick, tickxs, tickys, labelpos)
  Update tick line and label position and style.

70.2.3 art3d

Module containing 3D artist code and functions to convert 2D artists into 3D versions which can be added to an Axes3D.

class mpl_toolkits.mplot3d.art3d.Line3D(xs, ys, zs, *args, **kwargs)
  Bases: matplotlib.lines.Line2D

  3D line object.

  Keyword arguments are passed onto Line2D().

  draw(renderer)

set_3d_properties(zs=0, zdir='z')

class mpl_toolkits.mplot3d.art3d.Line3DCollection(segments, *args, **kwargs)
  Bases: matplotlib.collections.LineCollection

  A collection of 3D lines.

  Keyword arguments are passed onto LineCollection().

  do_3d_projection(renderer)
    Project the points according to renderer matrix.

  draw(renderer, project=False)

set_segments(segments)
  Set 3D segments

set_sort_zpos(val)
  Set the position to use for z-sorting.
class mpl_toolkits.mplot3d.art3d.Patch3D(*args, **kwargs)
    Bases: matplotlib.patches.Patch
    3D patch object.

do_3d_projection(renderer)

draw(renderer)

get_facecolor()

get_path()

set_3d_properties(verts, zs=0, zdir='z')

class mpl_toolkits.mplot3d.art3d.Patch3DCollection(*args, **kwargs)
    Bases: matplotlib.collections.PatchCollection
    A collection of 3D patches.

    Create a collection of flat 3D patches with its normal vector pointed in zdir direction, and located at zs on the zdir axis. 'zs' can be a scalar or an array-like of the same length as the number of patches in the collection.

    Constructor arguments are the same as for PatchCollection. In addition, keywords zs=0 and zdir='z' are available.

    Also, the keyword argument “depthshade” is available to indicate whether or not to shade the patches in order to give the appearance of depth (default is True). This is typically desired in scatter plots.

do_3d_projection(renderer)

set_3d_properties(zs, zdir)

set_sort_zpos(val)
    Set the position to use for z-sorting.

class mpl_toolkits.mplot3d.art3d.Path3DCollection(*args, **kwargs)
    Bases: matplotlib.collections.PathCollection
    A collection of 3D paths.

    Create a collection of flat 3D paths with its normal vector pointed in zdir direction, and located at zs on the zdir axis. 'zs' can be a scalar or an array-like of the same length as the number of paths in the collection.

    Constructor arguments are the same as for PathCollection. In addition, keywords zs=0 and zdir='z' are available.
Also, the keyword argument “depthshade” is available to indicate whether or not to shade the patches in order to give the appearance of depth (default is True). This is typically desired in scatter plots.

```python
do_3d_projection(renderer)
```

```python
set_3d_properties(zs, zdir)
```

```python
set_sort_zpos(val)
  Set the position to use for z-sorting.
```

```python
class mpl_toolkits.mplot3d.art3d.PathPatch3D(path, **kwargs)
  Bases: mpl_toolkits.mplot3d.art3d.Patch3D
  3D PathPatch object.
  do_3d_projection(renderer)
```

```python
set_3d_properties(path, zs=0, zdir='z')
```

```python
class mpl_toolkits.mplot3d.art3d.Poly3DCollection(verts, *args, **kwargs)
  Bases: matplotlib.collections.PolyCollection
  A collection of 3D polygons.
  Create a Poly3DCollection.
  verts should contain 3D coordinates.
  Keyword arguments: zsort, see set_zsort for options.
  Note that this class does a bit of magic with the _facecolors and _edgecolors properties.
  do_3d_projection(renderer)
    Perform the 3D projection for this object.
  draw(renderer)
```

```python
get_edgecolor()
```

```python
get_edgecolors()
```

```python
get_facecolor()
```

```python
get_facecolors()
```

```python
get_vector(segments3d)
  Optimize points for projection
```
```python
set_3d_properties()

set_alpha(alpha)
Set the alpha transparencies of the collection. alpha must be a float or None.
   ACCEPTS: float or None

set_edgecolor(colors)

set_edgecolors(colors)

set_facecolor(colors)

set_facecolors(colors)

set_sort_zpos(val)
Set the position to use for z-sorting.

set_verts(verts, closed=True)
Set 3D vertices.

set_verts_and_codes(verts, codes)
Sets 3D vertices with path codes

set_zsort(zsort)

    Set z-sorting behaviour: boolean: if True use default ‘average’ string: ‘average’, ‘min’ or ‘max’

class mpl_toolkits.mplot3d.art3d.Text3D(x=0, y=0, z=0, text='', zdir='z', **kwargs)
    Bases: matplotlib.text.Text
    Text object with 3D position and (in the future) direction.
    x, y, z Position of text text Text string to display zdir Direction of text
    Keyword arguments are passed onto Text().
    draw(renderer)

    set_3d_properties(z=0, zdir='z')

mpl_toolkits.mplot3d.art3d.get_colors(c, num)
    Stretch the color argument to provide the required number num

mpl_toolkits.mplot3d.art3d.get_dir_vector(zdir)

mpl_toolkits.mplot3d.art3d.get_patch_verts(patch)
    Return a list of vertices for the path of a patch.
```
mpl_toolkits.mplot3d.art3d.iscolor(c)

mpl_toolkits.mplot3d.art3d.juggle_axes(xs, ys, zs, zdir)
Reorder coordinates so that 2D xs, ys can be plotted in the plane orthogonal to zdir. zdir is normally
x, y or z. However, if zdir starts with a ‘-‘ it is interpreted as a compensation for rotate_axes.

mpl_toolkits.mplot3d.art3d.line_2d_to_3d(line, zs=0, zdir='z')
Convert a 2D line to 3D.

mpl_toolkits.mplot3d.art3d.line_collection_2d_to_3d(col, zs=0, zdir='z')
Convert a LineCollection to a Line3DCollection object.

mpl_toolkits.mplot3d.art3d.norm_angle(a)
Return angle between -180 and +180

mpl_toolkits.mplot3d.art3d.norm_text_angle(a)
Return angle between -90 and +90

mpl_toolkits.mplot3d.art3d.patch_2d_to_3d(patch, z=0, zdir='z')
Convert a Patch to a Patch3D object.

mpl_toolkits.mplot3d.art3d.patch_collection_2d_to_3d(col, zs=0, zdir='z',
depthshade=True)
Convert a PatchCollection into a Patch3DCollection object (or a PathCollection into a
Path3DCollection object).

Keywords:

za The location or locations to place the patches in the collection along the zdir axis. Defaults to
0.

zdir The axis in which to place the patches. Default is “z”.

depthshade Whether to shade the patches to give a sense of depth. Defaults to True.

mpl_toolkits.mplot3d.art3d.path_to_3d_segment(path, zs=0, zdir='z')
Convert a path to a 3D segment.

mpl_toolkits.mplot3d.art3d.path_to_3d_segment_with_codes(path, zs=0, zdir='z')
Convert a path to a 3D segment with path codes.

mpl_toolkits.mplot3d.art3d.pathpatch_2d_to_3d(pathpatch, z=0, zdir='z')
Convert a PathPatch to a PathPatch3D object.

mpl_toolkits.mplot3d.art3d.paths_to_3d_segments(paths, zs=0, zdir='z')
Convert paths from a collection object to 3D segments.

mpl_toolkits.mplot3d.art3d.paths_to_3d_segments_with_codes(paths, zs=0, zdir='z')
Convert paths from a collection object to 3D segments with path codes.

mpl_toolkits.mplot3d.art3d.poly_collection_2d_to_3d(col, zs=0, zdir='z')
Convert a PolyCollection to a Poly3DCollection object.

mpl_toolkits.mplot3d.art3d.rotate_axes(xs, ys, zs, zdir)
Reorder coordinates so that the axes are rotated with zdir along the original z axis. Prepending the
axis with a ‘-‘ does the inverse transform, so zdir can be x, -x, y, -y, z or -z
Convert a Text to a Text3D object.

Modify the alphas of the color list according to depth

Various transforms used for by the 3D code

Return 2D equation of line in the form $ax + by + c = 0$

Distance from line to point line is a tuple of coefficients $a, b, c$

distance(s) from line defined by $p1 - p2$ to point(s) $p0$

Intersection point $p = p1 + u*(p2-p1)$ and intersection point lies within segment if $u$ is between 0 and 1

3d vector length

Transform the points by the projection matrix

Transform the points by the projection matrix and return the clipping result returns $txs, tys, tzs, tis$

Distance from line to point line is a tuple of coefficients $a, b, c$

Intersection point $p = p1 + u*(p2-p1)$ and intersection point lies within segment if $u$ is between 0 and 1

3d vector length

Transform the points by the projection matrix

Transform the points by the projection matrix and return the clipping result returns $txs, tys, tzs, tis$
mpl_toolkits.mplot3d.proj3d.rot_x(\(V, \alpha\))

mpl_toolkits.mplot3d.proj3d.test_lines_dists()

mpl_toolkits.mplot3d.proj3d.test_proj()

mpl_toolkits.mplot3d.proj3d.test_proj_draw_axes(\(M, s=1\))

mpl_toolkits.mplot3d.proj3d.test_proj_make_M(\(E=\text{None}\))

mpl_toolkits.mplot3d.proj3d.test_rot()

mpl_toolkits.mplot3d.proj3d.test_world()

mpl_toolkits.mplot3d.proj3d.transform(xs, ys, zs, M)

Transform the points by the projection matrix

mpl_toolkits.mplot3d.proj3d.vec_pad_ones(xs, ys, zs)

mpl_toolkits.mplot3d.proj3d.view_transformation(E, R, V)

mpl_toolkits.mplot3d.proj3d.world_transformation(xmin, xmax, ymin, ymax, zmin, zmax)

### 70.3 mplot3d FAQ

#### 70.3.1 How is mplot3d different from MayaVi?

**MayaVi2** is a very powerful and featureful 3D graphing library. For advanced 3D scenes and excellent rendering capabilities, it is highly recommended to use MayaVi2.

mplot3d was intended to allow users to create simple 3D graphs with the same “look-and-feel” as matplotlib’s 2D plots. Furthermore, users can use the same toolkit that they are already familiar with to generate both their 2D and 3D plots.

#### 70.3.2 My 3D plot doesn’t look right at certain viewing angles

This is probably the most commonly reported issue with mplot3d. The problem is that – from some viewing angles – a 3D object would appear in front of another object, even though it is physically behind it. This can result in plots that do not look “physically correct.”
Unfortunately, while some work is being done to reduce the occurrence of this artifact, it is currently an intractable problem, and cannot be fully solved until matplotlib supports 3D graphics rendering at its core.

The problem occurs due to the reduction of 3D data down to 2D + z-order scalar. A single value represents the 3rd dimension for all parts of 3D objects in a collection. Therefore, when the bounding boxes of two collections intersect, it becomes possible for this artifact to occur. Furthermore, the intersection of two 3D objects (such as polygons or patches) cannot be rendered properly in matplotlib’s 2D rendering engine.

This problem will likely not be solved until OpenGL support is added to all of the backends (patches are greatly welcomed). Until then, if you need complex 3D scenes, we recommend using MayaVi.

### 70.3.3 I don’t like how the 3D plot is laid out, how do I change that?

Historically, mplot3d has suffered from a hard-coding of parameters used to control visuals such as label spacing, tick length, and grid line width. Work is being done to eliminate this issue. For matplotlib v1.1.0, there is a semi-official manner to modify these parameters. See the note in the `axis3d` section of the mplot3d API documentation for more information.
Part VI

Toolkits
Toolkits are collections of application-specific functions that extend matplotlib.
71.1 Basemap

(Not distributed with matplotlib)
Plots data on map projections, with continental and political boundaries, see basemap docs.

71.2 Cartopy

(Not distributed with matplotlib)
An alternative mapping library written for matplotlib v1.2 and beyond. Cartopy builds on top of matplotlib to provide object oriented map projection definitions and close integration with Shapely for powerful yet
easy-to-use vector data processing tools. An example plot from the Cartopy gallery:

US States which intersect the track of Hurricane Katrina (2005)
72.1 mplot3d

`mpl_toolkits.mplot3d` provides some basic 3D plotting (scatter, surf, line, mesh) tools. Not the fastest or feature complete 3D library out there, but ships with matplotlib and thus may be a lighter weight solution for some use cases.
72.2 AxesGrid

The matplotlib AxesGrid toolkit is a collection of helper classes to ease displaying multiple images in matplotlib. The AxesGrid toolkit is distributed with matplotlib source.

72.3 mplcursors

mplcursors provides interactive data cursors for matplotlib.

72.4 MplDataCursor

(Not distributed with matplotlib)

MplDataCursor is a toolkit written by Joe Kington to provide interactive “data cursors” (clickable annotation boxes) for matplotlib.

72.5 GTK Tools

mpl_toolkits.gtktools provides some utilities for working with GTK. This toolkit ships with matplotlib, but requires pygtk.

72.6 Excel Tools

mpl_toolkits.exceltools provides some utilities for working with Excel. This toolkit ships with matplotlib, but requires xlwt.
72.7 Natgrid

(Not distributed with matplotlib)

mpl_toolkits.natgrid is an interface to natgrid C library for gridding irregularly spaced data. This requires a separate installation of the natgrid toolkit.

72.8 Matplotlib-Venn

(Not distributed with matplotlib)

Matplotlib-Venn provides a set of functions for plotting 2- and 3-set area-weighted (or unweighted) Venn diagrams.

72.9 mplstereonet

(Not distributed with matplotlib)

mplstereonet provides stereonets for plotting and analyzing orientation data in Matplotlib.
Several projects have started to provide a higher-level interface to matplotlib. These are independent projects.

**73.1 seaborn**

*(Not distributed with matplotlib)*

seaborn is a high level interface for drawing statistical graphics with matplotlib. It aims to make visualization a central part of exploring and understanding complex datasets.

**73.2 holoviews**

*(Not distributed with matplotlib)*

holoviews makes it easier to visualize data interactively, especially in a Jupyter notebook, by providing a set of declarative plotting objects that store your data and associated metadata. Your data is then immediately visualizable alongside or overlaid with other data, either statically or with automatically provided widgets for parameter exploration.
import numpy as np
import holoviews as hv
hv.notebook_extension('matplotlib')
fractal = hv.Image(np.load('mandelbrot.npy'))
((fractal * hv.HLine(y=0)).hist() + fractal.sample(y=0))

### 73.3 ggplot

*(Not distributed with matplotlib)*

**ggplot** is a port of the R ggplot2 to python based on matplotlib.

### 73.4 prettyplotlib

*(Not distributed with matplotlib)*

**prettyplotlib** is an extension to matplotlib which changes many of the defaults to make plots some consider more attractive.
73.5 iTerm2 terminal backend

*(Not distributed with matplotlib)*

matplotlib_iTerm2 is an external matplotlib backend uses iTerm2 nightly build inline image display feature.
Part VII

External Resources
BOOKS, CHAPTERS AND ARTICLES

- Mastering matplotlib by Duncan M. McGregor
- Interactive Applications Using Matplotlib by Benjamin Root
- Matplotlib for Python Developers by Sandro Tosi
- Matplotlib chapter by John Hunter and Michael Droettboom in The Architecture of Open Source Applications
- Graphics with Matplotlib by David J. Raymond
- Ten Simple Rules for Better Figures by Nicolas P. Rougier, Michael Droettboom and Philip E. Bourne
- Learning Scientific Programming with Python chapter 7 by Christian Hill
• Plotting with matplotlib by Mike Müller
• Introduction to NumPy and Matplotlib by Eric Jones
• Anatomy of Matplotlib by Benjamin Root
• Data Visualization Basics with Python (O’Reilly) by Randal S. Olson
TUTORIALS

- Matplotlib tutorial by Nicolas P. Rougier
- Anatomy of Matplotlib - IPython Notebooks by Benjamin Root
Part VIII

The Matplotlib Developers’ Guide
This project is a community effort, and everyone is welcome to contribute. The project is hosted on https://github.com/matplotlib/matplotlib

### 77.1 Submitting a bug report

If you find a bug in the code or documentation, do not hesitate to submit a ticket to the Bug Tracker. You are also welcome to post feature requests or pull requests.

### 77.2 Retrieving and installing the latest version of the code

When working on the Matplotlib source, setting up a virtual environment or a conda environment is recommended.

**Warning:** If you already have a version of Matplotlib installed, use an virtual environment or uninstall using the same method you used to install it. Installing multiple versions of Matplotlib via different methods into the same environment may not always work as expected.

We use Git for version control and GitHub for hosting our main repository.

You can check out the latest sources with the command (see *Set up your fork* for more details):

```
git clone git@github.com:matplotlib/matplotlib.git
```

and navigate to the matplotlib directory.

To make sure the tests run locally you must build against the correct version of freetype. To configure the build system to fetch and build it either export the env MPLLOCALFREETYPE as:

```
export MPLLOCALFREETYPE=1
```

or copy setup.cfg.template to setup.cfg and edit it to contain
To install Matplotlib (and compile the c-extensions) run the following command from the top-level directory

```
$ pip install -v -e ./
```

This installs Matplotlib in ‘editable/develop mode’, i.e., builds everything and places the correct link entries in the install directory so that python will be able to import Matplotlib from the source directory. Thus, any changes to the *.py files will be reflected the next time you import the library. If you change the c-extension source (which might happen if you change branches) you will need to run:

```
$ python setup.py build
```

or re-run `pip install -v -e ./`

Alternatively, if you do

```
$ pip install -v ./
```

all of the files will be copied to the installation directory however, you will have to rerun this command every time the source is changed. Additionally you will need to copy `setup.cfg.template` to `setup.cfg` and edit it to contain

```
[test]
local_freetype = True
tests = True
```

In either case you can then run the tests to check your work environment is set up properly:

```
$ python tests.py
```

---

**Note:** Additional dependencies for testing: nose (version 1.0 or later), mock (if python < 3.3), Ghostscript, Inkscape

---

See also:

- *Developer’s tips for testing*

## 77.3 Contributing code

### 77.3.1 How to contribute

The preferred way to contribute to Matplotlib is to fork the main repository on GitHub, then submit a “pull request” (PR):

1. Create an account on GitHub if you do not already have one.

---
2. Fork the *project repository*: click on the ‘Fork’ button near the top of the page. This creates a copy of the code under your account on the GitHub server.

3. Clone this copy to your local disk:

```bash
$ git clone git@github.com:YourLogin/matplotlib.git
```

4. Create a branch to hold your changes:

```bash
$ git checkout -b my-feature origin/master
```

and start making changes. Never work in the `master` branch!

5. Work on this copy, on your computer, using Git to do the version control. When you’re done editing e.g., `lib/matplotlib/collections.py`, do:

```bash
$ git add lib/matplotlib/collections.py
$ git commit
```

to record your changes in Git, then push them to GitHub with:

```bash
$ git push -u origin my-feature
```

Finally, go to the web page of your fork of the Matplotlib repo, and click ‘Pull request’ to send your changes to the maintainers for review. You may want to consider sending an email to the mailing list for more visibility.

**See also:**
- Git documentation
- *Development workflow.*
- Working with Matplotlib source code

### 77.3.2 Contributing pull requests

It is recommended to check that your contribution complies with the following rules before submitting a pull request:

- If your pull request addresses an issue, please use the title to describe the issue and mention the issue number in the pull request description to ensure a link is created to the original issue.
- All public methods should have informative docstrings with sample usage when appropriate. Use the *numpy docstring standard*.
- Formatting should follow *PEP8 recommendation*. You should consider installing/enabling automatic PEP8 checking in your editor. Part of the test suite is checking PEP8 compliance, things go smoother if the code is mostly PEP8 compliant to begin with.
- Each high-level plotting function should have a simple example in the *Example* section of the docstring. This should be as simple as possible to demonstrate the method. More complex examples should go in the *examples* tree.
• Changes (both new features and bugfixes) should be tested. See *Developer’s tips for testing* for more details.

• Import the following modules using the standard scipy conventions:

```python
import numpy as np
import numpy.ma as ma
import matplotlib as mpl
import matplotlib.pyplot as plt
import matplotlib.cbook as cbook
import matplotlib.patches as mpatches
```

• If your change is a major new feature, add an entry to the *What's new* section by adding a new file in `doc/users/whats_new` (see `doc/users/whats_new/README` for more information).

• If you change the API in a backward-incompatible way, please document it in `doc/api/api_changes`, by adding a new file describing your changes (see `doc/api/api_changes/README` for more information).

• See below for additional points about *Keyword argument processing*, if code in your pull request does that.

In addition, you can check for common programming errors with the following tools:

• Code with a good unittest coverage (at least 70%, better 100%), check with:

```bash
pip install coverage
python tests.py --with-coverage
```

• No pyflakes warnings, check with:

```bash
pip install pyflakes
pyflakes path/to/module.py
```

**Note:** The current state of the Matplotlib code base is not compliant with all of those guidelines, but we expect that enforcing those constraints on all new contributions will move the overall code base quality in the right direction.

See also:

• *Coding guidelines*

• *Developer’s tips for testing*

• *Developer’s tips for documenting matplotlib*

### 77.3.3 Issues for New Contributors

New contributors should look for the following tags when looking for issues. We strongly recommend that new contributors tackle *new-contributor-friendly* issues (easy, well documented issues, that do not require
an understanding of the different submodules of Matplotlib) and Easy-fix issues. This helps the contributor become familiar with the contribution workflow, and for the core devs to become acquainted with the contributor; besides which, we frequently underestimate how easy an issue is to solve!

### 77.4 Other ways to contribute

Code is not the only way to contribute to Matplotlib. For instance, documentation is also a very important part of the project and often doesn’t get as much attention as it deserves. If you find a typo in the documentation, or have made improvements, do not hesitate to send an email to the mailing list or submit a GitHub pull request. Full documentation can be found under the doc/ directory.

It also helps us if you spread the word: reference the project from your blog and articles or link to it from your website!

### 77.5 Coding guidelines

#### 77.5.1 New modules and files: installation

- If you have added new files or directories, or reorganized existing ones, make sure the new files are included in the match patterns in MANIFEST.in, and/or in package_data in setup.py.

#### 77.5.2 C/C++ extensions

- Extensions may be written in C or C++.
- Code style should conform to PEP7 (understanding that PEP7 doesn’t address C++, but most of its admonitions still apply).
- Python/C interface code should be kept separate from the core C/C++ code. The interface code should be named F00_wrap.cpp or F00_wrapper.cpp.
- Header file documentation (aka docstrings) should be in Numpydoc format. We don’t plan on using automated tools for these docstrings, and the Numpydoc format is well understood in the scientific Python community.

#### 77.5.3 Keyword argument processing

Matplotlib makes extensive use of **kwargs for pass-through customizations from one function to another. A typical example is in `matplotlib.pyplot.text()`. The definition of the pylab text function is a simple pass-through to `matplotlib.axes.Axes.text()`:

```python
# in pylab.py
def text(*args, **kwargs):
    ret = gca().text(*args, **kwargs)
    draw_if_interactive()
    return ret
```
text() in simplified form looks like this, i.e., it just passes all args and kwargs on to matplotlib.text.

```
# in axes/_axes.py
def text(self, x, y, s, fontdict=None, withdash=False, **kwargs):
    t = Text(x=x, y=y, text=s, **kwargs)
```

and __init__() (again with liberties for illustration) just passes them on to the matplotlib.artist.Artist.update() method:

```
# in text.py
def __init__(self, x=0, y=0, text='', **kwargs):
    Artist.__init__(self)
    self.update(kwargs)
```

update does the work looking for methods named like set_property if property is a keyword argument. i.e., no one looks at the keywords, they just get passed through the API to the artist constructor which looks for suitably named methods and calls them with the value.

As a general rule, the use of **kwargs should be reserved for pass-through keyword arguments, as in the example above. If all the keyword args are to be used in the function, and not passed on, use the key/value keyword args in the function definition rather than the **kwargs idiom.

In some cases, you may want to consume some keys in the local function, and let others pass through. You can pop the ones to be used locally and pass on the rest. For example, in plot(), scalex and scaley are local arguments and the rest are passed on as Line2D() keyword arguments:

```
# in axes/_axes.py
def plot(self, *args, **kwargs):
    scalex = kwargs.pop('scalex', True)
    scaley = kwargs.pop('scaley', True)
    if not self._hold: self.cla()
    lines = []
    for line in self._get_lines(*args, **kwargs):
        self.add_line(line)
    lines.append(line)
```

Note: there is a use case when kwargs are meant to be used locally in the function (not passed on), but you still need the **kwargs idiom. That is when you want to use *args to allow variable numbers of non-keyword args. In this case, python will not allow you to use named keyword args after the *args usage, so you will be forced to use **kwargs. An example is matplotlib.contour.ContourLabeler.clabel():

```
# in contour.py
def clabel(self, *args, **kwargs):
    fontsize = kwargs.get('fontsize', None)
    inline = kwargs.get('inline', 1)
    self.fmt = kwargs.get('fmt', '%1.3f')
    colors = kwargs.get('colors', None)
    if len(args) == 0:
        levels = self.levels
        indices = range(len(self.levels))
    elif len(args) == 1:
        ...etc...
```
77.5.4 Developing a new backend

If you are working on a custom backend, the `backend` setting in `matplotlibrc` *(Customizing matplotlib)* supports an external backend via the `module` directive. If `my_backend.py` is a Matplotlib backend in your `PYTHONPATH`, you can set it on one of several ways:

- in `matplotlibrc`:
  
  ```plaintext
  backend : module://my_backend
  ```

- with the `MPLBACKEND` environment variable:
  
  ```plaintext
  > export MPLBACKEND="module://my_backend"
  > python simple_plot.py
  ```

- from the command shell with the `-d` flag:
  
  ```plaintext
  > python simple_plot.py -d module://my_backend
  ```

- with the use directive in your script:
  
  ```python
  import matplotlib
  matplotlib.use('module://my_backend')
  ```

77.5.5 Writing examples

We have hundreds of examples in subdirectories of `matplotlib/examples`, and these are automatically generated when the website is built to show up both in the examples and gallery sections of the website.

Any sample data that the example uses should be kept small and distributed with Matplotlib in the `lib/matplotlib/mpl-data/sample_data/` directory. Then in your example code you can load it into a file handle with:

```python
import matplotlib.cbook as cbook
fh = cbook.get_sample_data('mydata.dat')
```
DEVELOPER’S TIPS FOR TESTING

Matplotlib has a testing infrastructure based on nose, making it easy to write new tests. The tests are in matplotlib.tests, and customizations to the nose testing infrastructure are in matplotlib.testing. (There is other old testing cruft around, please ignore it while we consolidate our testing to these locations.)

### 78.1 Requirements

The following software is required to run the tests:

- **nose**, version 1.0 or later
- **mock**, when running python versions < 3.3
- **Ghostscript** (to render PDF files)
- **Inkscape** (to render SVG files)

Optionally you can install:

- **coverage** to collect coverage information
- **pep8** to test coding standards

### 78.2 Building matplotlib for image comparison tests

matplotlib’s test suite makes heavy use of image comparison tests, meaning the result of a plot is compared against a known good result. Unfortunately, different versions of FreeType produce differently formed characters, causing these image comparisons to fail. To make them reproducible, matplotlib can be built with a special local copy of FreeType. This is recommended for all matplotlib developers.

Add the following content to a `setup.cfg` file at the root of the matplotlib source directory:

```ini
[test]
local_freetype = True
tests = True
```

or by setting the `MPLLOCALFREETYPE` environmental variable to any true value.
78.3 Running the tests

Running the tests is simple. Make sure you have nose installed and run:

```python
tests.py
```

in the root directory of the distribution. The script takes a set of commands, such as:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--pep8</td>
<td>pep8 checks</td>
</tr>
<tr>
<td>--no-pep8</td>
<td>Do not perform pep8 checks</td>
</tr>
<tr>
<td>--no-network</td>
<td>Disable tests that require network access</td>
</tr>
</tbody>
</table>

Additional arguments are passed on to nosetests. See the nose documentation for supported arguments. Some of the more important ones are given here:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--verbose</td>
<td>Be more verbose</td>
</tr>
<tr>
<td>--processes=NUM</td>
<td>Run tests in parallel over NUM processes</td>
</tr>
<tr>
<td>--process-timeout=SECONDS</td>
<td>Set timeout for results from test runner process</td>
</tr>
<tr>
<td>--nocapture</td>
<td>Do not capture stdout</td>
</tr>
</tbody>
</table>

To run a single test from the command line, you can provide a dot-separated path to the module followed by the function separated by a colon, e.g., (this is assuming the test is installed):

```python
tests.py matplotlib.tests.test_simplification:test_clipping
```

If you want to run the full test suite, but want to save wall time try running the tests in parallel:

```python
tests.py --nocapture --verbose --processes=5 --process-timeout=300
```

An alternative implementation that does not look at command line arguments works from within Python is to run the tests from the matplotlib library function `matplotlib.test()`:

```python
import matplotlib
matplotlib.test()
```

**Hint:** To run the tests you need to install nose and mock if using python 2.7:

```bash
pip install nose  
pip install mock
```

78.4 Writing a simple test

Many elements of Matplotlib can be tested using standard tests. For example, here is a test from `matplotlib.tests.test_basic`:

```python
from nose.tools import assert_equal

def test_simple():
```
Nose determines which functions are tests by searching for functions beginning with “test” in their name.

If the test has side effects that need to be cleaned up, such as creating figures using the pyplot interface, use the @cleanup decorator:

```python
from matplotlib.testing.decorators import cleanup

@cleanup
def test_create_figure():
    """
    very simple example test that creates a figure using pyplot.
    """
    fig = figure()
    ...
```

### 78.5 Writing an image comparison test

Writing an image based test is only slightly more difficult than a simple test. The main consideration is that you must specify the “baseline”, or expected, images in the `image_comparison()` decorator. For example, this test generates a single image and automatically tests it:

```python
import numpy as np
import matplotlib
from matplotlib.testing.decorators import image_comparison
import matplotlib.pyplot as plt

@image_comparison(baseline_images=['spines_axes_positions'],
                  extensions=['png'])
def test_spines_axes_positions():
    # SF bug 2852168
    fig = plt.figure()
    x = np.linspace(0,2*np.pi,100)
    y = 2*np.sin(x)
    ax = fig.add_subplot(1,1,1)
    ax.set_title('centered spines')
    ax.plot(x,y)
    ax.spines['right'].set_position(('axes',0.1))
    ax.yaxis.set_ticks_position('right')
    ax.spines['top'].set_position(('axes',0.25))
    ax.xaxis.set_ticks_position('top')
    ax.spines['left'].set_color('none')
    ax.spines['bottom'].set_color('none')
```

The first time this test is run, there will be no baseline image to compare against, so the test will fail. Copy the output images (in this case result_images/test_category/spines_axes_positions.png) to the
correct subdirectory of baseline_images tree in the source directory (in this case lib/matplotlib/tests/baseline_images/test_category). Put this new file under source code revision control (with git add). When rerunning the tests, they should now pass.

The image_comparison() decorator defaults to generating png, pdf and svg output, but in interest of keeping the size of the library from ballooning we should only include the svg or pdf outputs if the test is explicitly exercising a feature dependent on that backend.

There are two optional keyword arguments to the image_comparison decorator:

- **extensions**: If you only wish to test additional image formats (rather than just png), pass any additional file types in the list of the extensions to test. When copying the new baseline files be sure to only copy the output files, not their conversions to png. For example only copy the files ending in pdf, not in _pdf.png.

- **tol**: This is the image matching tolerance, the default 1e-3. If some variation is expected in the image between runs, this value may be adjusted.

### 78.6 Known failing tests

If you’re writing a test, you may mark it as a known failing test with the knownfailureif() decorator. This allows the test to be added to the test suite and run on the buildbots without causing undue alarm. For example, although the following test will fail, it is an expected failure:

```python
from nose.tools import assert_equal
from matplotlib.testing.decorators import knownfailureif

@knownfailureif(True)
def test_simple_fail():
    "very simple example test that should fail"
    assert_equal(1+1,3)
```

Note that the first argument to the knownfailureif() decorator is a fail condition, which can be a value such as True, False, or ‘indeterminate’, or may be a dynamically evaluated expression.

### 78.7 Creating a new module in matplotlib.tests

We try to keep the tests categorized by the primary module they are testing. For example, the tests related to the mathtext.py module are in test_mathtext.py.

Let’s say you’ve added a new module named whizbang.py and you want to add tests for it in matplotlib.tests.test_whizbang. To add this module to the list of default tests, append its name to default_test_modules in lib/matplotlib/__init__.py.

### 78.8 Using Travis CI

Travis CI is a hosted CI system “in the cloud”.

---

1914 Chapter 78. Developer’s tips for testing
Travis is configured to receive notifications of new commits to GitHub repos (via GitHub “service hooks”) and to run builds or tests when it sees these new commits. It looks for a YAML file called `.travis.yml` in the root of the repository to see how to test the project.

Travis CI is already enabled for the main matplotlib GitHub repository – for example, see its Travis page. If you want to enable Travis CI for your personal matplotlib GitHub repo, simply enable the repo to use Travis CI in either the Travis CI UI or the GitHub UI (Admin | Service Hooks). For details, see the Travis CI Getting Started page. This generally isn’t necessary, since any pull request submitted against the main matplotlib repository will be tested.

Once this is configured, you can see the Travis CI results at https://travis-ci.org/your_GitHub_user_name/matplotlib – here’s an example.

### 78.9 Using tox

Tox is a tool for running tests against multiple Python environments, including multiple versions of Python (e.g., 2.7, 3.4, 3.5) and even different Python implementations altogether (e.g., CPython, PyPy, Jython, etc.)

Testing all versions of Python (2.6, 2.7, 3.*) requires having multiple versions of Python installed on your system and on the PATH. Depending on your operating system, you may want to use your package manager (such as apt-get, yum or MacPorts) to do this.

Tox makes it easy to determine if your working copy introduced any regressions before submitting a pull request. Here’s how to use it:

```bash
$ pip install tox
$ tox
```

You can also run tox on a subset of environments:

```bash
$ tox -e py26,py27
```

Tox processes everything serially so it can take a long time to test several environments. To speed it up, you might try using a new, parallelized version of tox called detox. Give this a try:

```bash
$ pip install -U -i http://pypi.testrun.org detox
$ detox
```

Tox is configured using a file called `tox.ini`. You may need to edit this file if you want to add new environments to test (e.g., py33) or if you want to tweak the dependencies or the way the tests are run. For more info on the `tox.ini` file, see the Tox Configuration Specification.
DEVELOPER’S TIPS FOR DOCUMENTING MATPLOTLIB

79.1 Getting started

The documentation for matplotlib is generated from ReStructured Text using the Sphinx documentation generation tool. The requirements are as follows (https://github.com/matplotlib/matplotlib/blob/master/doc-requirements.txt):

1. Sphinx-1.3 or later (Version 1.5.0 is not supported)
2. numpydoc 0.4 or later
3. IPython
4. Mock
5. colorspacious
6. pillow
7. graphviz

The documentation sources are found in the doc/ directory in the trunk. To build the users guide in html format, cd into doc/ and do:

```
python make.py html
```

or:

```
./make.py html
```

you can also pass a latex flag to make.py to build a pdf, or pass no arguments to build everything.

The output produced by Sphinx can be configured by editing the conf.py file located in the doc/.

79.2 Organization of matplotlib’s documentation

The actual ReStructured Text files are kept in doc/users, doc/devel, doc/api and doc/faq. The main entry point is doc/index.rst, which pulls in the index.rst file for the users guide, developers guide, api reference, and faqs. The documentation suite is built as a single document in order to make the most
effective use of cross referencing, we want to make navigating the Matplotlib documentation as easy as possible.

Additional files can be added to the various guides by including their base file name (the .rst extension is not necessary) in the table of contents. It is also possible to include other documents through the use of an include statement, such as:

```ini
.. include:: ../../TODO
```

## 79.2.1 docstrings

In addition to the “narrative” documentation described above, matplotlib also defines its API reference documentation in docstrings. For the most part, these are standard Python docstrings, but matplotlib also includes some features to better support documenting getters and setters.

Matplotlib uses artist introspection of docstrings to support properties. All properties that you want to support through `setp` and `getp` should have a `set_property` and `get_property` method in the `Artist` class. Yes, this is not ideal given python properties or enthought traits, but it is a historical legacy for now. The setter methods use the docstring with the `ACCEPTS` token to indicate the type of argument the method accepts. e.g., in `matplotlib.lines.Line2D`:

```py
# in lines.py
def set_linestyle(self, linestyle):
    """
    Set the linestyle of the line
    
    ACCEPTS: ['-']|
    """
```

Since matplotlib uses a lot of pass-through `kwargs`, e.g., in every function that creates a line (`plot()`, `semilogx()`, `semilogy()`, etc...), it can be difficult for the new user to know which `kwargs` are supported. Matplotlib uses a docstring interpolation scheme to support documentation of every function that takes a `**kwargs`. The requirements are:

1. single point of configuration so changes to the properties don’t require multiple docstring edits.
2. as automated as possible so that as properties change, the docs are updated automagically.

The function `matplotlib.artist.kwdoc()` and the decorator `matplotlib.docstring.dedent_interpd()` facilitate this. They combine python string interpolation in the docstring with the matplotlib artist introspection facility that underlies `setp` and `getp`. The `kwdoc` function gives the list of properties as a docstring. In order to use this in another docstring, first update the matplotlib. docstring.interpd object, as seen in this example from `matplotlib.lines`:

```py
# in lines.py
docstring.interpd.update(Line2D=artist.kwdoc(Line2D))
```

Then in any function accepting `Line2D` pass-through `kwargs`, e.g., `matplotlib.axes.Axes.plot()`:

```py
# in axes.py
@docstring.dedent_interpd
```
```python
def plot(self, *args, **kwargs):
    """
    Some stuff omitted

    The kwargs are Line2D properties:
    %(Line2D)s

    kwargs scalex and scaley, if defined, are passed on
to autoscale_view to determine whether the x and y axes are
autoscaled; default True. See Axes.autoscale_view for more
information
    """
    pass
```

Note there is a problem for Artist __init__ methods, e.g., matplotlib.patches.Patch.__init__(),
which supports Patch kwargs, since the artist inspector cannot work until the class is fully defined and we
can’t modify the Patch.__init__.__doc__ docstring outside the class definition. There are some some
manual hacks in this case, violating the “single entry point” requirement above – see the docstring.
interpd.update calls in matplotlib.patches.

## 79.3 Formatting

The Sphinx website contains plenty of documentation concerning ReST markup and working with Sphinx
in general. Here are a few additional things to keep in mind:

- Please familiarize yourself with the Sphinx directives for inline markup. Matplotlib’s documentation
  makes heavy use of cross-referencing and other semantic markup. For example, when referring to
  external files, use the :file: directive.

- Function arguments and keywords should be referred to using the emphasis role. This will keep
  matplotlib’s documentation consistent with Python’s documentation:

  Here is a description of *argument*

  Please do not use the default role:

  Please do not describe `argument` like this.

  nor the literal role:

  Please do not describe ``argument`` like this.

- Sphinx does not support tables with column- or row-spanning cells for latex output. Such tables can
  not be used when documenting matplotlib.

- Mathematical expressions can be rendered as png images in html, and in the usual way by latex. For
  example:

  :math:`\sin(x_n^2)` yields: \(\sin(x_n^2)\), and:
Interactive IPython sessions can be illustrated in the documentation using the following directive:

```ipython
In [69]: lines = plot([1,2,3])
```

which would yield:

```
In [69]: lines = plot([1,2,3])
```

Footnotes\(^1\) can be added using `[#]`, followed later by:

```
.. rubric:: Footnotes

.. [#]
```

Use the `note` and `warning` directives, sparingly, to draw attention to important comments:

```
.. note::

Here is a note
```

yields:

**Note:** here is a note

also:

```
Warning: here is a warning
```

Use the `deprecated` directive when appropriate:

```
.. deprecated:: 0.98

This feature is obsolete, use something else.
```

yields:

Deprecated since version 0.98: This feature is obsolete, use something else.

\(^1\) For example.
• Use the `versionadded` and `versionchanged` directives, which have similar syntax to the `deprecated` role:

```plaintext
.. versionadded:: 0.98
    The transforms have been completely revamped.
```

New in version 0.98: The transforms have been completely revamped.

• Use the `seealso` directive, for example:

```plaintext
.. seealso::

    Using ReST :ref:`emacs-helpers`:
        One example

    A bit about :ref:`referring-to-mpl-docs`:
        One more
```

yields:

**See also:**

**Using ReST Emacs helpers:** One example

**A bit about Referring to mpl documents:** One more

• Please keep the `Glossary` in mind when writing documentation. You can create a references to a term in the glossary with the `:term:` role.

• The autodoc extension will handle index entries for the API, but additional entries in the index need to be explicitly added.

• Please limit the text width of docstrings to 70 characters.

• Keyword arguments should be described using a definition list.

**Note:** matplotlib makes extensive use of keyword arguments as pass-through arguments, there are a many cases where a table is used in place of a definition list for autogenerated sections of docstrings.

### 79.4 Figures

#### 79.4.1 Dynamically generated figures

Figures can be automatically generated from scripts and included in the docs. It is not necessary to explicitly save the figure in the script, this will be done automatically at build time to ensure that the code that is included runs and produces the advertised figure.

The path should be relative to the doc directory. Any plots specific to the documentation should be added to the doc/mpl_examples/pyplots directory and committed to git. Plots from the examples directory may be referenced through the symlink mpl_examples in the doc directory, e.g.:
The :scale: directive rescales the image to some percentage of the original size, though we don’t recommend using this in most cases since it is probably better to choose the correct figure size and dpi in mpl and let it handle the scaling.

**Plot directive documentation**

A directive for including a matplotlib plot in a Sphinx document.

By default, in HTML output, plot will include a .png file with a link to a high-res .png and .pdf. In LaTeX output, it will include a .pdf.

The source code for the plot may be included in one of three ways:

1. **A path to a source file** as the argument to the directive:

   ```python
   .. plot:: path/to/plot.py
   ```

   When a path to a source file is given, the content of the directive may optionally contain a caption for the plot:

   ```python
   .. plot:: path/to/plot.py
   
   This is the caption for the plot
   ```

   Additionally, one may specify the name of a function to call (with no arguments) immediately after importing the module:

   ```python
   .. plot:: path/to/plot.py plot_function1
   ```

2. Included as **inline content** to the directive:

   ```python
   .. plot::
   
   import matplotlib.pyplot as plt
   import matplotlib.image as mpimg
   import numpy as np
   img = mpimg.imread('_static/stinkbug.png')
   imgplot = plt.imshow(img)
   ```

3. Using **doctest** syntax:

   ```python
   .. plot::
   
   A plotting example:
   >>> import matplotlib.pyplot as plt
   >>> plt.plot([1,2,3], [4,5,6])
   ```
Options

The plot directive supports the following options:

- **format** [{‘python’, ‘doctest’}] Specify the format of the input
- **include-source** [bool] Whether to display the source code. The default can be changed using the plot_include_source variable in conf.py
- **encoding** [str] If this source file is in a non-UTF8 or non-ASCII encoding, the encoding must be specified using the :encoding: option. The encoding will not be inferred using the -*- coding -*- metacomment.
- **context** [bool or str] If provided, the code will be run in the context of all previous plot directives for which the :context: option was specified. This only applies to inline code plot directives, not those run from files. If the :context: reset option is specified, the context is reset for this and future plots, and previous figures are closed prior to running the code. :context:close-figs keeps the context but closes previous figures before running the code.
- **nofigs** [bool] If specified, the code block will be run, but no figures will be inserted. This is usually useful with the :context: option.

Additionally, this directive supports all of the options of the image directive, except for target (since plot will add its own target). These include alt, height, width, scale, align and class.

Configuration options

The plot directive has the following configuration options:

- **plot_include_source** Default value for the include-source option
- **plot_html_show_source_link** Whether to show a link to the source in HTML.
- **plot_pre_code** Code that should be executed before each plot.
- **plot_basedir** Base directory, to which plot:: file names are relative to. (If None or empty, file names are relative to the directory where the file containing the directive is.)
- **plot_formats** File formats to generate. List of tuples or strings:

  ```python
  [(suffix, dpi), suffix, ...]
  ```

  that determine the file format and the DPI. For entries whose DPI was omitted, sensible defaults are chosen. When passing from the command line through sphinx_build the list should be passed as suffix:dpi,suffix:dpi, ....

- **plot_html_show_formats** Whether to show links to the files in HTML.
- **plot_rcparams** A dictionary containing any non-standard rcParams that should be applied before each plot.
plot_apply_rcparams  By default, rcParams are applied when context option is not used in a plot directive. This configuration option overrides this behavior and applies rcParams before each plot.

plot_working_directory  By default, the working directory will be changed to the directory of the example, so the code can get at its data files, if any. Also its path will be added to sys.path so it can import any helper modules sitting beside it. This configuration option can be used to specify a central directory (also added to sys.path) where data files and helper modules for all code are located.

plot_template  Provide a customized template for preparing restructured text.

79.4.2 Static figures

Any figures that rely on optional system configurations need to be handled a little differently. These figures are not to be generated during the documentation build, in order to keep the prerequisites to the documentation effort as low as possible. Please run the doc/pyplots/make.py script when adding such figures, and commit the script and the images to git. Please also add a line to the README in doc/pyplots for any additional requirements necessary to generate a new figure. Once these steps have been taken, these figures can be included in the usual way:

```python
.. plot:: mpl_examples/pyplots/tex_unicode_demo.py
   :include-source:
```

79.4.3 Examples

The source of the files in the examples directory are automatically included in the HTML docs. An image is generated and included for all examples in the api and pylab_examples directories. To exclude the example from having an image rendered, insert the following special comment anywhere in the script:

```
# -*- noplot -*-
```

79.4.4 Animations

We have a matplotlib google/gmail account with username mplgithub which we used to setup the github account but can be used for other purposes, like hosting google docs or youtube videos. You can embed a matplotlib animation in the docs by first saving the animation as a movie using `matplotlib.animation.Animation.save()`, and then uploading to matplotlib’s youtube channel and inserting the embedding string youtube provides like:

```html
.. raw:: html

   <iframe width="420" height="315"
      src="http://www.youtube.com/embed/32cjc6V0OZY"
      frameborder="0" allowfullscreen>

   </iframe>
```
An example save command to generate a movie looks like this

```python
ani = animation.FuncAnimation(fig, animate, np.arange(1, len(y)),
    interval=25, blit=True, init_func=init)
ani.save('double_pendulum.mp4', fps=15)
```

Contact Michael Droettboom for the login password to upload youtube videos of google docs to the mplgithub account.

### 79.5 Referring to mpl documents

In the documentation, you may want to include a document in the matplotlib src, e.g., a license file or an image file from mpl-data, refer to it via a relative path from the document where the rst file resides, e.g., in `users/navigation_toolbar.rst`, we refer to the image icons with:

```
.. image:: ../../lib/matplotlib/mpl-data/images/subplots.png
```

In the `users` subdirectory, if I want to refer to a file in the mpl-data directory, I use the symlink directory. For example, from `customizing.rst`:

```
.. literalinclude:: ../../lib/matplotlib/mpl-data/matplotlibrc
```

One exception to this is when referring to the examples dir. Relative paths are extremely confusing in the sphinx plot extensions, so without getting into the dirty details, it is easier to simply include a symlink to the files at the top doc level directory. This way, API documents like `matplotlib.pyplot.plot()` can refer to the examples in a known location.

In the top level doc directory we have symlinks pointing to the mpl examples:

```
home:/mpl/doc> ls -l mpl_*
ml_examples -> ../examples
```

So we can include plots from the examples dir using the symlink:

```
.. plot:: mpl_examples/pylab_examples/simple_plot.py
```

We used to use a symlink for mpl-data too, but the distro becomes very large on platforms that do not support links (e.g., the font files are duplicated and large)

### 79.6 Internal section references

To maximize internal consistency in section labeling and references, use hyphen separated, descriptive labels for section references, e.g.:

```
.. _howto-webapp:
```

and refer to it using the standard reference syntax:
Keep in mind that we may want to reorganize the contents later, so let’s avoid top level names in references like user or devel or faq unless necessary, because for example the FAQ “what is a backend?” could later become part of the users guide, so the label:

.. _what-is-a-backend

is better than:

.. _faq-backend

In addition, since underscores are widely used by Sphinx itself, let’s prefer hyphens to separate words.

### 79.7 Section names, etc

For everything but top level chapters, please use Upper lower for section titles, e.g., Possible hangups rather than Possible Hangups

### 79.8 Inheritance diagrams

Class inheritance diagrams can be generated with the inheritance-diagram directive. To use it, you provide the directive with a number of class or module names (separated by whitespace). If a module name is provided, all classes in that module will be used. All of the ancestors of these classes will be included in the inheritance diagram.

A single option is available: parts controls how many of parts in the path to the class are shown. For example, if parts == 1, the class matplotlib.patches.Patch is shown as Patch. If parts == 2, it is shown as patches.Patch. If parts == 0, the full path is shown.

Example:

```
.. inheritance-diagram:: matplotlib.patches matplotlib.lines matplotlib.text
   :parts: 2
```
79.9 Emacs helpers

There is an emacs mode `rst.el` which automates many important ReST tasks like building and updating table-of-contents, and promoting or demoting section headings. Here is the basic `.emacs` configuration:

```
(require 'rst)
(setq auto-mode-alist
     (append '(("\.txt\$" . rst-mode)
              ('"\.rst\$" . rst-mode)
              ('"\.rest\$" . rst-mode)) auto-mode-alist))
```

Some helpful functions:

- **C-c TAB** - `rst-toc-insert`
  
  Insert table of contents at point

- **C-c C-u** - `rst-toc-update`
Update the table of contents at point

C-c C-l rst-shift-region-left

Shift region to the left

C-c C-r rst-shift-region-right

Shift region to the right
Matplotlib supports the addition of custom procedures that transform the data before it is displayed.

There is an important distinction between two kinds of transformations. Separable transformations, working on a single dimension, are called “scales”, and non-separable transformations, that handle data in two or more dimensions at a time, are called “projections”.

From the user’s perspective, the scale of a plot can be set with \texttt{set_xscale()} and \texttt{set_yscale()}. Projections can be chosen using the \texttt{projection} keyword argument to the \texttt{plot()} or \texttt{subplot()} functions, e.g.:

\begin{lstlisting}[language=python]
plot(x, y, projection="custom")
\end{lstlisting}

This document is intended for developers and advanced users who need to create new scales and projections for matplotlib. The necessary code for scales and projections can be included anywhere: directly within a plot script, in third-party code, or in the matplotlib source tree itself.

\section*{80.1 Creating a new scale}

Adding a new scale consists of defining a subclass of \texttt{matplotlib.scale.ScaleBase}, that includes the following elements:

- A transformation from data coordinates into display coordinates.
- An inverse of that transformation. This is used, for example, to convert mouse positions from screen space back into data space.
- A function to limit the range of the axis to acceptable values (\texttt{limit_range_for_scale()}). A log scale, for instance, would prevent the range from including values less than or equal to zero.
- Locators (major and minor) that determine where to place ticks in the plot, and optionally, how to adjust the limits of the plot to some “good” values. Unlike \texttt{limit_range_for_scale()}, which is always enforced, the range setting here is only used when automatically setting the range of the plot.
- Formatters (major and minor) that specify how the tick labels should be drawn.

Once the class is defined, it must be registered with matplotlib so that the user can select it.
A full-fledged and heavily annotated example is in examples/api/custom_scale_example.py. There are also some classes in matplotlib.scale that may be used as starting points.

## 80.2 Creating a new projection

Adding a new projection consists of defining a projection axes which subclasses matplotlib.axes.Axes and includes the following elements:

- A transformation from data coordinates into display coordinates.
- An inverse of that transformation. This is used, for example, to convert mouse positions from screen space back into data space.
- Transformations for the gridlines, ticks and ticklabels. Custom projections will often need to place these elements in special locations, and matplotlib has a facility to help with doing so.
- Setting up default values (overriding cla()), since the defaults for a rectilinear axes may not be appropriate.
- Defining the shape of the axes, for example, an elliptical axes, that will be used to draw the background of the plot and for clipping any data elements.
- Defining custom locators and formatters for the projection. For example, in a geographic projection, it may be more convenient to display the grid in degrees, even if the data is in radians.
- Set up interactive panning and zooming. This is left as an “advanced” feature left to the reader, but there is an example of this for polar plots in matplotlib.projections.polar.
- Any additional methods for additional convenience or features.

Once the projection axes is defined, it can be used in one of two ways:

- By defining the class attribute `name`, the projection axes can be registered with matplotlib.projections.register_projection() and subsequently simply invoked by name:

  ```python
  plt.axes(projection='my_proj_name')
  ```

- For more complex, parameterisable projections, a generic “projection” object may be defined which includes the method `_as_mpl_axes`. `_as_mpl_axes` should take no arguments and return the projection’s axes subclass and a dictionary of additional arguments to pass to the subclass’ `__init__` method. Subsequently a parameterised projection can be initialised with:

  ```python
  plt.axes(projection=MyProjection(param1=param1_value))
  ```

  where MyProjection is an object which implements a `_as_mpl_axes` method.

A full-fledged and heavily annotated example is in examples/api/custom_projection_example.py. The polar plot functionality in matplotlib.projections.polar may also be of interest.
80.3 API documentation

- matplotlib.scale
- matplotlib.projections
- matplotlib.projections.polar
As of matplotlib 1.4, the six library is used to support Python 2 and 3 from a single code base. The 2to3 tool is no longer used.

This document describes some of the issues with that approach and some recommended solutions. It is not a complete guide to Python 2 and 3 compatibility.

81.1 Welcome to the __future__

The top of every py file should include the following:

```python
from __future__ import (absolute_import, division,
                        print_function, unicode_literals)
import six
```

This will make the Python 2 interpreter behave as close to Python 3 as possible.

All matplotlib files should also import six, whether they are using it or not, just to make moving code between modules easier, as six gets used a lot.

81.2 Finding places to use six

The only way to make sure code works on both Python 2 and 3 is to make sure it is covered by unit tests. However, the 2to3 commandline tool can also be used to locate places that require special handling with six.

(The modernize tool may also be handy, though I’ve never used it personally).

The six documentation serves as a good reference for the sorts of things that need to be updated.

81.3 The dreaded \u escapes

When from __future__ import unicode_literals is used, all string literals (not preceded with a b) will become unicode literals.
Normally, one would use “raw” string literals to encode strings that contain a lot of slashes that we don’t want Python to interpret as special characters. A common example in matplotlib is when it deals with TeX and has to represent things like `r"\usepackage{foo}"`. Unfortunately, on Python 2 there is no way to represent \ in a raw unicode string literal, since it will always be interpreted as the start of a unicode character escape, such as \u20af. The only solution is to use a regular (non-raw) string literal and repeat all slashes, e.g. `"\usepackage{foo}"`.

The following shows the problem on Python 2:

```python
g>>> ur'\u'
  File "<stdin>", line 1
SyntaxError: (unicode error) 'rawunicodeescape' codec can't decode bytes in position 0-1: truncated \uXXXX
>>> ur'\\u'
u'\\\u'
>>> u'\\u'
  File "<stdin>", line 1
SyntaxError: (unicode error) 'unicodeescape' codec can't decode bytes in position 0-1: truncated \uXXXX escape

>>> u'\\u'
u'\\u'
```

This bug has been fixed in Python 3, however, we can’t take advantage of that and still support Python 2:

```python
g>>> r'\u'
'\\u'
>>> r'\\u'
'\\\u'
>>> '\\u'
  File "<stdin>", line 1
SyntaxError: (unicode error) 'unicodeescape' codec can't decode bytes in position 0-1: truncated \uXXXX escape
>>> '\\u'
'\\u'
```

### 81.4 Iteration

The behavior of the methods for iterating over the items, values and keys of a dictionary has changed in Python 3. Additionally, other built-in functions such as `zip`, `range` and `map` have changed to return iterators rather than temporary lists.

In many cases, the performance implications of iterating vs. creating a temporary list won’t matter, so it’s tempting to use the form that is simplest to read. However, that results in code that behaves differently on Python 2 and 3, leading to subtle bugs that may not be detected by the regression tests. Therefore, unless the loop in question is provably simple and doesn’t call into other code, the `six` versions that ensure the same behavior on both Python 2 and 3 should be used. The following table shows the mapping of equivalent semantics between Python 2, 3 and six for `dict.items()`:
<table>
<thead>
<tr>
<th>Python 2</th>
<th>Python 3</th>
<th>six</th>
</tr>
</thead>
<tbody>
<tr>
<td>d.items()</td>
<td>list(d.items())</td>
<td>list(six.iteritems(d))</td>
</tr>
<tr>
<td>d.iteritems()</td>
<td>d.items()</td>
<td>six.iteritems(d)</td>
</tr>
</tbody>
</table>

### 81.5 Numpy-specific things

When specifying dtypes, all strings must be byte strings on Python 2 and unicode strings on Python 3. The best way to handle this is to force cast them using `str()`. The same is true of structure specifiers in the `struct` built-in module.
82.1 Introduction

These pages describe a git and github workflow for the Matplotlib project. There are several different workflows here, for different ways of working with Matplotlib.

This is not a comprehensive git reference, it’s just a workflow for our own project. It’s tailored to the github hosting service. You may well find better or quicker ways of getting stuff done with git, but these should get you started.

For general resources for learning git, see git resources.

82.2 Install git

82.2.1 Overview

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debian / Ubuntu</td>
<td><code>sudo apt-get install git</code></td>
</tr>
<tr>
<td>Fedora</td>
<td><code>sudo yum install git</code></td>
</tr>
<tr>
<td>Windows</td>
<td>Download and install msysGit</td>
</tr>
<tr>
<td>OS X</td>
<td>Use the <code>git-osx-installer</code></td>
</tr>
</tbody>
</table>

82.2.2 In detail

See the git page for the most recent information.

Have a look at the github install help pages available from github help

There are good instructions here: https://git-scm.com/book/en/v2/Getting-Started-Installing-Git
82.3 Following the latest source

These are the instructions if you just want to follow the latest Matplotlib source, but you don’t need to do any development for now.

The steps are:

- Install git
- get local copy of the Matplotlib github git repository
- update local copy from time to time

82.3.1 Get the local copy of the code

From the command line:

```bash
git clone git://github.com/matplotlib/matplotlib.git
```

You now have a copy of the code tree in the new matplotlib directory.

82.3.2 Updating the code

From time to time you may want to pull down the latest code. Do this with:

```bash
cd matplotlib
git pull
```

The tree in matplotlib will now have the latest changes from the initial repository.

82.4 Making a patch

You’ve discovered a bug or something else you want to change in Matplotlib .. — excellent!

You’ve worked out a way to fix it — even better!

You want to tell us about it — best of all!

The easiest way is to make a patch or set of patches. Here we explain how. Making a patch is the simplest and quickest, but if you’re going to be doing anything more than simple quick things, please consider following the Git for development model instead.

82.4.1 Making patches

Overview
Then, send the generated patch files to the Matplotlib mailing list — where we will thank you warmly.

**In detail**

1. Tell git who you are so it can label the commits you’ve made:

   ```
   git config --global user.email you@yourdomain.example.com
   git config --global user.name "Your Name Comes Here"
   ```

2. If you don’t already have one, clone a copy of the Matplotlib repository:

   ```
   git clone git://github.com/matplotlib/matplotlib.git
   cd matplotlib
   ```

3. Make a ‘feature branch’. This will be where you work on your bug fix. It’s nice and safe and leaves you with access to an unmodified copy of the code in the main branch:

   ```
   git branch the-fix-im-thinking-of
   git checkout the-fix-im-thinking-of
   ```

4. Do some edits, and commit them as you go:

   ```
   # hack, hack, hack
   # Tell git about any new files you’ve made
   git add somewhere/tests/test_my_bug.py
   # commit work in progress as you go
   git commit -am 'BF - added tests for Funny bug'
   # hack hack, hack
   git commit -am 'BF - added fix for Funny bug'
   # make the patch files
   git format-patch -M -C master
   ```

   Note the -am options to `commit`. The m flag just signals that you’re going to type a message on the command line. The a flag — you can just take on faith — or see why the -a flag?.
5. When you have finished, check you have committed all your changes:

```bash
git status
```

6. Finally, make your commits into patches. You want all the commits since you branched from the `master` branch:

```bash
git format-patch -M -C master
```

You will now have several files named for the commits:

```
0001-BF-added-tests-for-Funny-bug.patch
0002-BF-added-fix-for-Funny-bug.patch
```

Send these files to the Matplotlib mailing list.

When you are done, to switch back to the main copy of the code, just return to the `master` branch:

```bash
git checkout master
```

### 82.4.2 Moving from patching to development

If you find you have done some patches, and you have one or more feature branches, you will probably want to switch to development mode. You can do this with the repository you have.

Fork the Matplotlib repository on github — *Making your own copy (fork) of Matplotlib*. Then:

```bash
# checkout and refresh master branch from main repo
git checkout master

git pull origin master
# rename pointer to main repository to 'upstream'
git remote rename origin upstream
# point your repo to default read / write to your fork on github
git remote add origin git@github.com:your-user-name/matplotlib.git
# push up any branches you've made and want to keep
git push origin the-fix-im-thinking-of
```

Then you can, if you want, follow the *Development workflow*.

### 82.5 Git for development

Contents:

#### 82.5.1 Making your own copy (fork) of Matplotlib

You need to do this only once. The instructions here are very similar to the instructions at https://help.github.com/forking/ — please see that page for more detail. We’re repeating some of it here just to give the specifics for the Matplotlib project, and to suggest some default names.
Set up and configure a github account

If you don’t have a github account, go to the github page, and make one.
You then need to configure your account to allow write access — see the Generating SSH keys help on github help.

Create your own forked copy of Matplotlib

1. Log into your github account.
2. Go to the Matplotlib github home at Matplotlib github.
3. Click on the fork button:

Now, after a short pause, you should find yourself at the home page for your own forked copy of Matplotlib.

82.5.2 Set up your fork

First you follow the instructions for Making your own copy (fork) of Matplotlib.

Overview

```
git clone git@github.com:your-user-name/matplotlib.git
cd matplotlib
git remote add upstream git://github.com/matplotlib/matplotlib.git
```

In detail

Clone your fork

1. Clone your fork to the local computer with `git clone git@github.com:your-user-name/matplotlib.git`
2. Investigate. Change directory to your new repo: `cd matplotlib`. Then `git branch -a` to show you all branches. You’ll get something like:

```
* master
remotes/origin/master
```
This tells you that you are currently on the master branch, and that you also have a remote connection to origin/master. What remote repository is remote/origin? Try git remote -v to see the URLs for the remote. They will point to your github fork.

Now you want to connect to the upstream Matplotlib github repository, so you can merge in changes from trunk.

### Linking your repository to the upstream repo

```bash
cd matplotlib
git remote add upstream git://github.com/matplotlib/matplotlib.git
```

upstream here is just the arbitrary name we’re using to refer to the main Matplotlib repository at Matplotlib github.

Note that we’ve used git:// for the URL rather than git@. The git:// URL is read only. This means we that we can’t accidentally (or deliberately) write to the upstream repo, and we are only going to use it to merge into our own code.

Just for your own satisfaction, show yourself that you now have a new ‘remote’, with git remote -v show, giving you something like:

```
upstream    git://github.com/matplotlib/matplotlib.git (fetch)
upstream    git://github.com/matplotlib/matplotlib.git (push)
origin      git@github.com:your-user-name/matplotlib.git (fetch)
origin      git@github.com:your-user-name/matplotlib.git (push)
```

### 82.5.3 Configure git

#### Overview

Your personal git configurations are saved in the .gitconfig file in your home directory.

Here is an example .gitconfig file:

```ini
[user]
    name = Your Name
    email = you@yourdomain.example.com

[alias]
    ci = commit -a
    co = checkout
    st = status
    stat = status
    br = branch
    wdiff = diff --color-words

[core]
    editor = vim
```
### [merge]

| Summary = true |

You can edit this file directly or you can use the `git config --global` command:

```bash
git config --global user.name "Your Name"
git config --global user.email you@yourdomain.example.com
git config --global alias.ci "commit -a"
git config --global alias.co checkout
git config --global alias.st "status -a"
git config --global alias.stat "status -a"
git config --global alias.br branch
git config --global alias.wdiff "diff --color-words"
git config --global core.editor vim
git config --global merge.summary true
```

To set up on another computer, you can copy your `~/.gitconfig` file, or run the commands above.

#### In detail

**user.name and user.email**

It is good practice to tell `git` who you are, for labeling any changes you make to the code. The simplest way to do this is from the command line:

```bash
git config --global user.name "Your Name"
git config --global user.email you@yourdomain.example.com
```

This will write the settings into your git configuration file, which should now contain a user section with your name and email:

```plaintext
[User]
name = Your Name
email = you@yourdomain.example.com
```

Of course you’ll need to replace `Your Name` and `you@yourdomain.example.com` with your actual name and email address.

**Aliases**

You might well benefit from some aliases to common commands.

For example, you might well want to be able to shorten `git checkout` to `git co`. Or you may want to alias `git diff --color-words` (which gives a nicely formatted output of the diff) to `git wdiff`

The following `git config --global` commands:

```bash
git config --global alias.ci "commit -a"
git config --global alias.co checkout
```

---

82.5. Git for development 1943
will create an alias section in your .gitconfig file with contents like this:

```
[alias]
    ci = commit -a
    co = checkout
    st = status -a
    stat = status -a
    br = branch
    wdiff = diff --color-words
```

### Editor

You may also want to make sure that your editor of choice is used:

```
git config --global core.editor vim
```

### Merging

To enforce summaries when doing merges (~/.gitconfig file again):

```
[merge]
    log = true
```

Or from the command line:

```
git config --global merge.log true
```

### Fancy log output

This is a very nice alias to get a fancy log output; it should go in the alias section of your .gitconfig file:

```
lg = log --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr) %d %C(bold blue)%an%Creset' --abbrev-commit --date=relative
```

You use the alias with:

```
git lg
```

and it gives graph / text output something like this (but with color!):
82.5.4 Development workflow

You already have your own forked copy of the Matplotlib repository, by following Making your own copy (fork) of Matplotlib. You have Set up your fork. You have configured git by following Configure git. Now you are ready for some real work.

Workflow summary

In what follows we’ll refer to the upstream Matplotlib master branch, as “trunk”.

- Don’t use your master branch for anything. Consider deleting it.
- When you are starting a new set of changes, fetch any changes from trunk, and start a new feature branch from that.
- Make a new branch for each separable set of changes — “one task, one branch” (ipython git workflow).
- Name your branch for the purpose of the changes - e.g. bugfix-for-issue-14 or refactor-database-code.
- If you can possibly avoid it, avoid merging trunk or any other branches into your feature branch while you are working.
- If you do find yourself merging from trunk, consider Rebasing on trunk.
• Ask on the Matplotlib mailing list if you get stuck.
• Ask for code review!

This way of working helps to keep work well organized, with readable history. This in turn makes it easier for project maintainers (that might be you) to see what you’ve done, and why you did it.

See linux git workflow and ipython git workflow for some explanation.

Consider deleting your master branch

It may sound strange, but deleting your own master branch can help reduce confusion about which branch you are on. See deleting master on github for details.

Update the mirror of trunk

First make sure you have done Linking your repository to the upstream repo.

From time to time you should fetch the upstream (trunk) changes from github:

```
git fetch upstream
```

This will pull down any commits you don’t have, and set the remote branches to point to the right commit. For example, ‘trunk’ is the branch referred to by (remote/branchname) upstream/master - and if there have been commits since you last checked, upstream/master will change after you do the fetch.

Make a new feature branch

When you are ready to make some changes to the code, you should start a new branch. Branches that are for a collection of related edits are often called ‘feature branches’.

Making an new branch for each set of related changes will make it easier for someone reviewing your branch to see what you are doing.

Choose an informative name for the branch to remind yourself and the rest of us what the changes in the branch are for. For example add-ability-to-fly, or buxfix-for-issue-42.

```
# Update the mirror of trunk
git fetch upstream
# Make new feature branch starting at current trunk
git branch my-new-feature upstream/master
git checkout my-new-feature
```

Generally, you will want to keep your feature branches on your public github fork of Matplotlib. To do this, you git push this new branch up to your github repo. Generally (if you followed the instructions in these pages, and by default), git will have a link to your github repo, called origin. You push up to your own repo on github with:

```
git push origin my-new-feature
```
In git >= 1.7 you can ensure that the link is correctly set by using the `--set-upstream` option:

```
git push --set-upstream origin my-new-feature
```

From now on git will know that `my-new-feature` is related to the `my-new-feature` branch in the github repo.

### The editing workflow

#### Overview

```
# hack hack
git add my_new_file
git commit -am 'NF - some message'
git push
```

#### In more detail

1. Make some changes
2. See which files have changed with `git status` (see `git status`). You’ll see a listing like this one:

```
# On branch my-new-feature
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
# modified:  README
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
# INSTALL
no changes added to commit (use "git add" and/or "git commit -a")
```
3. Check what the actual changes are with `git diff` (`git diff`).
4. Add any new files to version control `git add new_file_name` (see `git add`).
5. To commit all modified files into the local copy of your repo, do `git commit -am 'A commit message'`. Note the `-am` options to commit. The `m` flag just signals that you’re going to type a message on the command line. The a flag — you can just take on faith — or see why the `-a` flag? — and the helpful use-case description in the tangled working copy problem. The `git commit` manual page might also be useful.
6. To push the changes up to your forked repo on github, do a `git push` (see `git push`).

82.5. Git for development 1947
Ask for your changes to be reviewed or merged

When you are ready to ask for someone to review your code and consider a merge:

1. Go to the URL of your forked repo, say https://github.com/your-user-name/matplotlib.

2. Use the ‘Switch Branches’ dropdown menu near the top left of the page to select the branch with your changes:

3. Click on the ‘Pull request’ button:

Enter a title for the set of changes, and some explanation of what you’ve done. Say if there is anything you’d like particular attention for - like a complicated change or some code you are not happy with.

If you don’t think your request is ready to be merged, just say so in your pull request message. This is still a good way of getting some preliminary code review.

Some other things you might want to do

Delete a branch on github

```
git checkout master
# delete branch locally
git branch -D my-unwanted-branch
# delete branch on github
git push origin :my-unwanted-branch
```

Note the colon : before my-unwanted-branch. See also: https://help.github.com/articles/pushing-to-a-remote/#deleting-a-remote-branch-or-tag
Several people sharing a single repository

If you want to work on some stuff with other people, where you are all committing into the same repository, or even the same branch, then just share it via github.

First fork Matplotlib into your account, as from Making your own copy (fork) of Matplotlib. Then, go to your forked repository github page, say https://github.com/your-user-name/matplotlib

Click on the ‘Admin’ button, and add anyone else to the repo as a collaborator:

Now all those people can do:

git clone git@github.com:your-user-name/matplotlib.git

Remember that links starting with git@ use the ssh protocol and are read-write; links starting with git:// are read-only.

Your collaborators can then commit directly into that repo with the usual:

git commit -am 'ENH - much better code'
git push origin master # pushes directly into your repo

Explore your repository

To see a graphical representation of the repository branches and commits:

gitk --all

To see a linear list of commits for this branch:

git log

You can also look at the network graph visualizer for your github repo.

Finally the Fancy log output lg alias will give you a reasonable text-based graph of the repository.

Rebasing on trunk

Let’s say you thought of some work you’d like to do. You Update the mirror of trunk and Make a new feature branch called cool-feature. At this stage trunk is at some commit, let’s call it E. Now you make
some new commits on your cool-feature branch, let’s call them A, B, C. Maybe your changes take a while, or you come back to them after a while. In the meantime, trunk has progressed from commit E to commit (say) G:

```
A---B---C cool-feature
/     
D---E---F---G trunk
```

At this stage you consider merging trunk into your feature branch, and you remember that this here page sternly advises you not to do that, because the history will get messy. Most of the time you can just ask for a review, and not worry that trunk has got a little ahead. But sometimes, the changes in trunk might affect your changes, and you need to harmonize them. In this situation you may prefer to do a rebase.

rebase takes your changes (A, B, C) and replays them as if they had been made to the current state of trunk. In other words, in this case, it takes the changes represented by A, B, C and replays them on top of G. After the rebase, your history will look like this:

```
A'--B'--C' cool-feature
/     
D---E---F---G trunk
```

See rebase without tears for more detail.

To do a rebase on trunk:

```
# Update the mirror of trunk
git fetch upstream
# go to the feature branch
git checkout cool-feature
# make a backup in case you mess up
git branch tmp cool-feature
# rebase cool-feature onto trunk
git rebase --onto upstream/master upstream/master cool-feature
```

In this situation, where you are already on branch cool-feature, the last command can be written more succinctly as:

```
git rebase upstream/master
```

When all looks good you can delete your backup branch:

```
git branch -D tmp
```

If it doesn’t look good you may need to have a look at Recovering from mess-ups.

If you have made changes to files that have also changed in trunk, this may generate merge conflicts that you need to resolve - see the git rebase man page for some instructions at the end of the “Description” section. There is some related help on merging in the git user manual - see resolving a merge.
Recovering from mess-ups

Sometimes, you mess up merges or rebases. Luckily, in git it is relatively straightforward to recover from such mistakes.

If you mess up during a rebase:

```
git rebase --abort
```

If you notice you messed up after the rebase:

```
# reset branch back to the saved point
git reset --hard tmp
```

If you forgot to make a backup branch:

```
# look at the reflog of the branch
git reflog show cool-feature

8630830 cool-feature@{0}: commit: BUG: io: close file handles immediately
278dd2a cool-feature@{1}: rebase finished: refs/heads/my-feature-branch onto..11ee694744f2552d
26aa21a cool-feature@{2}: commit: BUG: lib: make seek_gzip_factory not leak gzip obj...

# reset the branch to where it was before the botched rebase
git reset --hard cool-feature@{2}
```

Rewriting commit history

**Note:** Do this only for your own feature branches.

There’s an embarrassing typo in a commit you made? Or perhaps the you made several false starts you would like the posterity not to see.

This can be done via *interactive rebasing*.

Suppose that the commit history looks like this:

```
git log --oneline
eadc391 Fix some remaining bugs
a815645 Modify it so that it works
2dec1ac Fix a few bugs + disable
13d7934 First implementation
6ad92e5 * masked is now an instance of a new object, MaskedConstant
2901ed Add pre-nep for a couple of structured_array_extensions.
...
```

and 6ad92e5 is the last commit in the *cool-feature* branch. Suppose we want to make the following changes:
• Rewrite the commit message for 13d7934 to something more sensible.
• Combine the commits 2dec1ac, a815645, eadc391 into a single one.

We do as follows:

```
# make a backup of the current state
git branch tmp HEAD
# interactive rebase
git rebase -i 6ad92e5
```

This will open an editor with the following text in it:

```
pick 13d7934 First implementation
pick 2dec1ac Fix a few bugs + disable
pick a815645 Modify it so that it works
pick eadc391 Fix some remaining bugs

# Rebase 6ad92e5..eadc391 onto 6ad92e5
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
#
# If you remove a line here THAT COMMIT WILL BE LOST.
# However, if you remove everything, the rebase will be aborted.
#
```

To achieve what we want, we will make the following changes to it:

```
r 13d7934 First implementation
pick 2dec1ac Fix a few bugs + disable
f a815645 Modify it so that it works
f eadc391 Fix some remaining bugs
```

This means that (i) we want to edit the commit message for 13d7934, and (ii) collapse the last three commits into one. Now we save and quit the editor.

Git will then immediately bring up an editor for editing the commit message. After revising it, we get the output:

```
[detached HEAD 721fc64] FOO: First implementation
  2 files changed, 199 insertions(+), 66 deletions(-)
[detached HEAD 0f22701] Fix a few bugs + disable
  1 files changed, 79 insertions(+), 61 deletions(-)
Successfully rebased and updated refs/heads/my-feature-branch.
```

and the history looks now like this:
0f22701 Fix a few bugs + disable
721fc64 ENH: Sophisticated feature
6ad92e5 * masked is now an instance of a new object, MaskedConstant

If it went wrong, recovery is again possible as explained above.

### 82.5.5 Maintainer workflow

This page is for maintainers — those of us who merge our own or other peoples’ changes into the upstream repository.

Being as how you’re a maintainer, you are completely on top of the basic stuff in Development workflow.

The instructions in Linking your repository to the upstream repo add a remote that has read-only access to the upstream repo. Being a maintainer, you’ve got read-write access.

It’s good to have your upstream remote have a scary name, to remind you that it’s a read-write remote:

```
git remote add upstream-rw git@github.com:matplotlib/matplotlib.git

# Fetch upstream changes
git fetch upstream-rw
```

### Integrating changes

Let’s say you have some changes that need to go into trunk (upstream-rw/master).

The changes are in some branch that you are currently on. For example, you are looking at someone’s changes like this:

```
git remote add someone git://github.com/someone/matplotlib.git

# Fetch upstream changes
git fetch upstream-rw
```

```
git branch cool-feature --track someone/cool-feature

# Checkout cool-feature
```

So now you are on the branch with the changes to be incorporated upstream. The rest of this section assumes you are on this branch.

### A few commits

If there are only a few commits, consider rebasing to upstream:

```
# Fetch upstream changes
git fetch upstream-rw

# rebase
git rebase upstream-rw/master
```

Remember that, if you do a rebase, and push that, you’ll have to close any github pull requests manually, because github will not be able to detect the changes have already been merged.
A long series of commits

If there are a longer series of related commits, consider a merge instead:

```
git fetch upstream-rw
git merge --no-ff upstream-rw/master
```

The merge will be detected by github, and should close any related pull requests automatically.

Note the `--no-ff` above. This forces git to make a merge commit, rather than doing a fast-forward, so that these set of commits branch off trunk then rejoin the main history with a merge, rather than appearing to have been made directly on top of trunk.

Check the history

Now, in either case, you should check that the history is sensible and you have the right commits:

```
git log --oneline --graph
 git log -p upstream-rw/master..
```

The first line above just shows the history in a compact way, with a text representation of the history graph. The second line shows the log of commits excluding those that can be reached from trunk (`upstream-rw/master`), and including those that can be reached from current HEAD (implied with the `..` at the end). So, it shows the commits unique to this branch compared to trunk. The `-p` option shows the diff for these commits in patch form.

Push to trunk

```
git push upstream-rw my-new-feature:master
```

This pushes the `my-new-feature` branch in this repository to the `master` branch in the `upstream-rw` repository.

82.6 git resources

82.6.1 Tutorials and summaries

- `github help` has an excellent series of how-to guides.
- The `pro git book` is a good in-depth book on git.
- A `git cheat sheet` is a page giving summaries of common commands.
- The `git user manual`
- The `git tutorial`
- The `git community book`
• **git ready** — a nice series of tutorials
• **git magic** — extended introduction with intermediate detail
• The **git parable** is an easy read explaining the concepts behind git.
• **git foundation** expands on the **git parable**.
• Fernando Perez’ **git page** — Fernando’s **git page** — many links and tips
• A good but technical page on **git concepts**
• **git svn crash course**: git for those of us used to **subversion**

### 82.6.2 Advanced git workflow

There are many ways of working with git; here are some posts on the rules of thumb that other projects have come up with:

• Linus Torvalds on **git management**
  
  Linus Torvalds on **linux git workflow**. Summary; use the git tools to make the history of your edits as clean as possible; merge from upstream edits as little as possible in branches where you are doing active development.

### 82.6.3 Manual pages online

You can get these on your own machine with (e.g) **git help push** or (same thing) **git push --help**, but, for convenience, here are the online manual pages for some common commands:

• **git add**
• **git branch**
• **git checkout**
• **git clone**
• **git commit**
• **git config**
• **git diff**
• **git log**
• **git pull**
• **git push**
• **git remote**
• **git status**
82.7 Two and three dots in difference specs

Thanks to Yarik Halchenko for this explanation.

Imagine a series of commits A, B, C, D... Imagine that there are two branches, topic and master. You branched topic off master when master was at commit ‘E’. The graph of the commits looks like this:

```
A---B---C topic
/    
D---E---F---G master
```

Then:

```
git diff master..topic
```

will output the difference from G to C (i.e. with effects of F and G), while:

```
git diff master...topic
```

would output just differences in the topic branch (i.e. only A, B, and C).
83.1 Pull request checklist

83.1.1 Branch selection

- In general, simple bugfixes that are unlikely to introduce new bugs of their own should be merged onto the maintenance branch. New features, or anything that changes the API, should be made against master. The rules are fuzzy here – when in doubt, target master.

- Once changes are merged into the maintenance branch, they should be merged into master.

83.1.2 Documentation

- Every new feature should be documented. If it’s a new module, don’t forget to add a new rst file to the API docs.

- Each high-level plotting function should have a simple example in the Example section of the docstring. This should be as simple as possible to demonstrate the method. More complex examples should go in the examples tree.

- Build the docs and make sure all formatting warnings are addressed.

- See Developer’s tips for documenting matplotlib for our documentation style guide.

- If your change is a major new feature, add an entry to doc/users/whats_new.rst.

- If you change the API in a backward-incompatible way, please document it in doc/api/api_changes.rst.

83.2 PR Review guidelines

- If you have a commit bit, then you are trusted to use it. Please help review and merge PRs!

- Two developers (those with commit rights) should review all pull requests. If you are the first to review a PR please and approve of the changes please edit the title to include ‘[MRG+1]’ and use the github ‘approve review’ tool to mark it as such. If you are a subsequent reviewer and you approve
either merge (and backport if needed) or increment the number in the title to ask for further review (and trigger the gh ‘approve review’). If you do the merge please removed the ' [MRG+N'] prefix.

- Make sure the Travis tests are passing before merging.
  - The Travis tests automatically test on all of the Python versions Matplotlib supports whenever a pull request is created or updated. The tox support in Matplotlib may be useful for testing locally.

- Do not self merge, except for ‘small’ patches to un-break the CI.

- Squashing is case-by-case. The balance is between burden on the contributor, keeping a relatively clean history, and keeping a history usable for bisecting. The only time we are really strict about it is to eliminate binary files (ex multiple test image re-generations) and to remove upstream merges.

- Be patient with new contributors.

- Do not let perfect be the enemy of the good, particularly for documentation or example PRs. If you find yourself making many small suggestions, either open a PR against the original branch or merge the PR and then open a new PR against upstream.

### 83.3 Backports

When doing backports please include the branch you backported the commit to along with the SHA in a comment on the original PR.

Assuming we have matplotlib as a read-only remote to the matplotlib/matplotlib repo and DANGER as a read/write remote to the matplotlib/matplotlib repo, we do a backport from master to 2.x. The TARGET_SHA is the hash of the merge commit you would like to backport. This can be read off of the github PR page (in the UI with the merge notification) or through the git CLI tools:

```bash
git fetch matplotlib
git checkout v2.x
git merge --ff-only matplotlib/v2.x
git cherry-pick -m 1 TARGET_SHA
gitk  # to look at it
# local tests? (use your judgment)
git push DANGER v2.x
# leave a comment on PR noting sha of the resulting commit
# from the cherry-pick + branch it was moved to
```

These commands work on git 2.7.1.
A guide for developers who are doing a matplotlib release.

## 84.1 All Releases

### 84.1.1 Testing

We use travis-ci for continuous integration. When preparing for a release, the final tagged commit should be tested locally before it is uploaded:

```bash
python tests.py --processes=8 --process-timeout=300
```

In addition the following two tests should be run and manually inspected:

```bash
python unit/memleak_hawaii3.py
pushd examples/tests/
python backend_driver.py
popd
```

### 84.1.2 GitHub Stats

We automatically extract GitHub issue, PRs, and authors from GitHub via the API:

```bash
python tools/github_stats.py --since-tag $TAG --project 'matplotlib/matplotlib' --links > doc/users/github_stats.rst
```

Review and commit changes. Some issue/PR titles may not be valid rst (the most common issue is * which is interpreted as unclosed markup).

### 84.1.3 Check Docs

Before tagging, make sure that the docs build cleanly
pushd doc
python make.py html latex -n 16
popd

After the docs are built, check that all of the links, internal and external, are still valid. We use linkchecker for this, which has not been ported to python3 yet. You will need to create a python2 environment with requests==2.9.0 and linkchecker

conda create -p /tmp/lnkchk python=2 requests==2.9.0
source activate /tmp/lnkchk
pip install linkchecker
pushd doc/build/html
linkchecker index.html --check-extern

Address any issues which may arise. The internal links are checked on travis, this should only flag failed external links.

84.1.4 Create release commit and tag

To create the tag, first create an empty commit with a very terse set of the release notes in the commit message

```bash
git commit --allow-empty
```

and then create a signed, annotated tag with the same text in the body message

```bash
git tag -a -s v2.0.0
```

which will prompt you for your gpg key password and an annotation. For pre releases it is important to follow PEP 440 so that the build artifacts will sort correctly in pypi. Finally, push the tag to GitHub

```bash
git push -t DANGER v2.0.0
```

Congratulations, the scariest part is done!

To prevent issues with any down-stream builders which download the tarball from GitHub it is important to move all branches away from the commit with the tag\(^1\):

```bash
git commit --allow-empty
git push DANGER master
```

If this is a final release, also create a ‘doc’ branch (this is not done for pre-releases):

\(^1\) The tarball that is provided by GitHub is produced using git archive. We use versioneer which uses a format string in `lib/matplotlib/_version.py` to have git insert a list of references to exported commit (see `.gitattributes` for the configuration). This string is then used by versioneer to produce the correct version, based on the git tag, when users install from the tarball. However, if there is a branch pointed at the tagged commit, then the branch name will also be included in the tarball. When the branch eventually moves, anyone who checked the hash of the tarball before the branch moved will have an incorrect hash.

To generate the file that GitHub does use

```bash
git archive v2.0.0 -o matplotlib-2.0.0.tar.gz --prefix=matplotlib-2.0.0/
```
and if this is a major or minor release, also create a bug-fix branch (a micro release will be cut off of this branch):

```
git branch v2.0.x
git push DANGER v2.0.x
```

### 84.1.5 Release Management / DOI

Via the GitHub UI (chase down link), turn the newly pushed tag into a release. If this is a pre-release remember to mark it as such.

For final releases also get a DOI from [zenodo](https://zenodo.org) and edit `doc/_templates/citing.html` with DOI link and commit to the VER-doc branch and push to GitHub

```
git checkout v2.0.0-doc
emacs doc/_templates/citing.html
git push DANGER v2.0.0-doc:v2.0.0-doc
```

### 84.1.6 Building binaries

We distribute mac, windows, and many linux wheels as well as a source tarball via pypi. Before uploading anything, contact the various builders. Mac and manylinux wheels are built on travis. You need to edit the `.travis.yml` file and push to master of the build project.

Update the master branch (for pre-releases the `devel` branch) of the [conda-forge feedstock](https://github.com/conda-forge/mpl-feedstock) via pull request.

If this is a final release the following downstream packagers should be contacted:

- Debian
- Fedora
- Arch
- Gentoo
- Macports
- Homebrew
- Christoph Gohlke
- Continuum
- Enthought

This can be done ahead of collecting all of the binaries and uploading to pypi.
84.1.7 make distribution and upload to pypi / SF

Once you have collected all of the wheels, generate the tarball

```
    git checkout v2.0.0
    git clean -xfd
    python setup.py sdist
```

and copy all of the wheels into `dist` directory. You should use `twine` to upload all of the files to pypi

```
    twine upload -s dist/matplotlib*tar.gz
    twine upload dist/*whl
```

Congratulations, you have now done the second scariest part!

Additionally, for a final release, upload all of the files to sourceforge.

84.1.8 Build and Deploy Documentation

To build the documentation you must have the tagged version installed, but build the docs from the `ver-doc` branch. An easy way to arrange this is:

```
    pip install matplotlib
    pip install -r doc-requirements.txt
    git checkout v2.0.0-doc
    git clean -xfd
    cd doc
    python make.py html latex -n 16
```

which will build both the html and pdf version of the documentation.

The built documentation exists in the `matplotlib.github.com` repository. Pushing changes to master automatically updates the website.

The documentation is organized by version. At the root of the tree is always the documentation for the latest stable release. Under that, there are directories containing the documentation for older versions. The documentation for current master are built on travis and push to the `devdocs` repository. These are available at `matplotlib.org/devdocs`.

Assuming you have this repository checked out in the same directory as matplotlib

```
    cd ../matplotlib.github.com
    mkdir 2.0.0
    rsync -a ../matplotlib/doc/build/html/* 2.0.0
    cp ../matplotlib/doc/build/latex/Matplotlib.pdf 2.0.0
```

which will copy the built docs over. If this is a final release, also replace the top-level docs

```
    rsync -a 2.0.0/* ./
```

You will need to manually edit `versions.html` to show the last 3 tagged versions. Now commit and push everything to GitHub
git add *
git commit -a -m 'Updating docs for v2.0.0'
git push DANGER master

Congratulations you have now done the third scariest part!

It typically takes about 5-10 minutes for GitHub to process the push and update the live web page (remember to clear your browser cache).

84.1.9 Announcing

The final step is to announce the release to the world. A short version of the release notes along with acknowledgments should be sent to

- matplotlib-user@python.org
- matplotlib-devel@python.org
- matplotlib-announce@python.org

For final releases announcements should also be sent to the numpy/scipy/jupyter mailing lists and python-announce.

In addition, announcements should be made on social networks (twitter, g+, FB). For major release, NumFOCUS should be contacted for inclusion in their newsletter and maybe to have something posted on their blog.
This MEP template is a guideline of the sections that a MEP should contain. Extra sections may be added if appropriate, and unnecessary sections may be noted as such.

### 85.1.1 Status

MEPs go through a number of phases in their lifetime:

- **Discussion**: The MEP is being actively discussed on the mailing list and it is being improved by its author. The mailing list discussion of the MEP should include the MEP number (MEPxxx) in the subject line so they can be easily related to the MEP.

- **Progress**: Consensus was reached on the mailing list and implementation work has begun.

- **Completed**: The implementation has been merged into master.

- **Superseded**: This MEP has been abandoned in favor of another approach.

### 85.1.2 Branches and Pull requests

All development branches containing work on this MEP should be linked to from here.
All pull requests submitted relating to this MEP should be linked to from here. (A MEP does not need to be implemented in a single pull request if it makes sense to implement it in discrete phases).

85.1.3 Abstract

The abstract should be a short description of what the MEP will achieve.

85.1.4 Detailed description

This section describes the need for the MEP. It should describe the existing problem that it is trying to solve and why this MEP makes the situation better. It should include examples of how the new functionality would be used and perhaps some use cases.

85.1.5 Implementation

This section lists the major steps required to implement the MEP. Where possible, it should be noted where one step is dependent on another, and which steps may be optionally omitted. Where it makes sense, each step should include a link related pull requests as the implementation progresses.

85.1.6 Backward compatibility

This section describes the ways in which the MEP breaks backward incompatibility.

85.1.7 Alternatives

If there were any alternative solutions to solving the same problem, they should be discussed here, along with a justification for the chosen approach.

85.2 MEP8: PEP8
85.2.1 Status

Discussion

85.2.2 Branches and Pull requests

None so far.

85.2.3 Abstract

The matplotlib codebase predates PEP8, and therefore is less than consistent style-wise in some areas. Bringing the codebase into compliance with PEP8 would go a long way to improving its legibility.

85.2.4 Detailed description

Some files use four space indentation, some use three. Some use different levels in the same file.

For the most part, class/function/variable naming follows PEP8, but it wouldn’t hurt to fix where necessary.

85.2.5 Implementation

The implementation should be fairly mechanical: running the pep8 tool over the code and fixing where appropriate.

This should be merged in after the 2.0 release, since the changes will likely make merging any pending pull requests more difficult.

Additionally, and optionally, PEP8 compliance could be tracked by an automated build system.

85.2.6 Backward compatibility

Public names of classes and functions that require change (there shouldn’t be many of these) should first be deprecated and then removed in the next release cycle.

85.2.7 Alternatives

PEP8 is a popular standard for Python code style, blessed by the Python core developers, making any alternatives less desirable.

85.3 MEP9: Global interaction manager
Add a global manager for all user interactivity with artists; make any artist resizeable, moveable, highlightable, and selectable as desired by the user.

85.3.1 Status

Discussion

85.3.2 Branches and Pull requests

https://github.com/dhyams/matplotlib/tree/MEP9

85.3.3 Abstract

The goal is to be able to interact with matplotlib artists in a very similar way as drawing programs do. When appropriate, the user should be able to move, resize, or select an artist that is already on the canvas. Of course, the script writer is ultimately in control of whether an artist is able to be interacted with, or whether it is static.

This code to do this has already been privately implemented and tested, and would need to be migrated from its current “mixin” implementation, to a bona-fide part of matplotlib.

The end result would be to have four new keywords available to matplotlib.artist.Artist: _moveable_, _resizeable_, _selectable_, and _highlightable_. Setting any one of these keywords to True would activate interactivity for that artist.

In effect, this MEP is a logical extension of event handling in matplotlib; matplotlib already supports “low level” interactions like left mouse presses, a key press, or similar. The MEP extends the support to the logical level, where callbacks are performed on the artists when certain interactive gestures from the user are detected.
85.3.4 Detailed description

This new functionality would be used to allow the end-user to better interact with the graph. Many times, a graph is almost what the user wants, but a small repositioning and/or resizing of components is necessary. Rather than force the user to go back to the script to trial-and-error the location, and simple drag and drop would be appropriate.

Also, this would better support applications that use matplotlib; here, the end-user has no reasonable access or desire to edit the underlying source in order to fine-tune a plot. Here, if matplotlib offered the capability, one could move or resize artists on the canvas to suit their needs. Also, the user should be able to highlight (with a mouse over) an artist, and select it with a double-click, if the application supports that sort of thing. In this MEP, we also want to support the highlighting and selection natively; it is up to application to handle what happens when the artist is selected. A typical handling would be to display a dialog to edit the properties of the artist.

In the future, as well (this is not part of this MEP), matplotlib could offer backend-specific property dialogs for each artist, which are raised on artist selection. This MEP would be a necessary stepping stone for that sort of capability.

There are currently a few interactive capabilities in matplotlib (e.g. legend.draggable()), but they tend to be scattered and are not available for all artists. This MEP seeks to unify the interactive interface and make it work for all artists.

The current MEP also includes grab handles for resizing artists, and appropriate boxes drawn when artists are moved or resized.

85.3.5 Implementation

- Add appropriate methods to the “tree” of artists so that the interactivity manager has a consistent interface for the interactivity manager to deal with. The proposed methods to add to the artists, if they are to support interactivity, are:
  - get_pixel_position_ll(self): get the pixel position of the lower left corner of the artist’s bounding box
  - get_pixel_size(self): get the size of the artist’s bounding box, in pixels
  - set_pixel_position_and_size(self,x,y,dx,dy): set the new size of the artist, such that it fits within the specified bounding box.

- add capability to the backends to 1) provide cursors, since these are needed for visual indication of moving/resizing, and 2) provide a function that gets the current mouse position

- Implement the manager. This has already been done privately (by dhyams) as a mixin, and has been tested quite a bit. The goal would be to move the functionality of the manager into the artists so that it is in matplotlib properly, and not as a “monkey patch” as I currently have it coded.

85.3.6 Current summary of the mixin

(Note that this mixin is for now just private code, but can be added to a branch obviously)
InteractiveArtistMixin:

Mixin class to make any generic object that is drawn on a matplotlib canvas moveable and possibly resizeable. The Powerpoint model is followed as closely as possible; not because I’m enamoured with Powerpoint, but because that’s what most people understand. An artist can also be selectable, which means that the artist will receive the on_activated() callback when double clicked. Finally, an artist can be highlightable, which means that a highlight is drawn on the artist whenever the mouse passes over. Typically, highlightable artists will also be selectable, but that is left up to the user. So, basically there are four attributes that can be set by the user on a per-artist basis:

- highlightable
- selectable
- moveable
- resizeable

To be moveable (draggable) or resizeable, the object that is the target of the mixin must support the following protocols:

- get_pixel_position_ll(self)
- get_pixel_size(self)
- set_pixel_position_and_size(self,x,y,sx,sy)

Note that nonresizeable objects are free to ignore the sx and sy parameters. To be highlightable, the object that is the target of the mixin must also support the following protocol:

- get_highlight(self)

Which returns a list of artists that will be used to draw the highlight.

If the object that is the target of the mixin is not an matplotlib artist, the following protocols must also be implemented. Doing so is usually fairly trivial, as there has to be an artist somewhere that is being drawn. Typically your object would just route these calls to that artist.

- get_figure(self)
- get_axes(self)
- contains(self,event)
- set_animated(self,flag)
- draw(self,renderer)
- get_visible(self)

The following notifications are called on the artist, and the artist can optionally implement these.

- on_select_begin(self)
- on_select_end(self)
- on_drag_begin(self)
- on_drag_end(self)
- on_activated(self)
- on_highlight(self)
- on_right_click(self, event)
- on_left_click(self, event)
- on_middle_click(self, event)
- on_context_click(self, event)
- on_key_up(self, event)
- on_key_down(self, event)

The following notifications are called on the canvas, if no interactive artist handles the event:

- on_press(self, event)
- on_left_click(self, event)
- on_middle_click(self, event)
- on_right_click(self, event)
- on_context_click(self, event)
- on_key_up(self, event)
- on_key_down(self, event)

The following functions, if present, can be used to modify the behavior of the interactive object:

- press_filter(self, event) # determines if the object wants to have the press event routed to it
- handle_unpicked_cursor() # can be used by the object to set a cursor as the cursor passes over the object when it is unpicked.

Supports multiple canvases, maintaining a drag lock, motion notifier, and a global “enabled” flag per canvas. Supports fixed aspect ratio resizings by holding the shift key during the resize.

Known problems:

- Zorder is not obeyed during the selection/drag operations. Because of the blit technique used, I do not believe this can be fixed. The only way I can think of is to search for all artists that have a zorder greater then me, set them all to animated, and then redraw them all on top during each drag refresh. This might be very slow; need to try.

- the mixin only works for wx backends because of two things: 1) the cursors are hardcoded, and 2) there is a call to wx.GetMousePosition() Both of these shortcomings are reasonably fixed by having each backend supply these things.

**85.3.7 Backward compatibility**

No problems with backward compatibility, although once this is in place, it would be appropriate to obsolete some of the existing interactive functions (like legend.draggable())
85.3.8 Alternatives

None that I know of.

85.4 MEP10: Docstring consistency

- **Status**
- **Branches and Pull requests**
- **Abstract**
- **Detailed description**
  - Numpy docstring format
  - Cross references
  - Overriding signatures
  - Linking rather than duplicating
  - autosummary extension
  - Examples linking to relevant documentation
  - Documentation using help() vs a browser
- **Implementation**
- **Backward compatibility**
- **Alternatives**

85.4.1 Status

Progress

Targeted for 1.3

85.4.2 Branches and Pull requests

#1665 #1757 #1795

85.4.3 Abstract

matplotlib has a great deal of inconsistency between docstrings. This not only makes the docs harder to read, but it is harder on contributors, because they don’t know which specifications to follow. There should be a clear docstring convention that is followed consistently.
The organization of the API documentation is difficult to follow. Some pages, such as pyplot and axes, are enormous and hard to browse. There should instead be short summary tables that link to detailed documentation. In addition, some of the docstrings themselves are quite long and contain redundant information.

Building the documentation takes a long time and uses a make.py script rather than a Makefile.

85.4.4 Detailed description

There are number of new tools and conventions available since matplotlib started using Sphinx that make life easier. The following is a list of proposed changes to docstrings, most of which involve these new features.

Numpy docstring format

Numpy docstring format: This format divides the docstring into clear sections, each having different parsing rules that make the docstring easy to read both as raw text and as HTML. We could consider alternatives, or invent our own, but this is a strong choice, as it’s well used and understood in the Numpy/Scipy community.

Cross references

Most of the docstrings in matplotlib use explicit “roles” when linking to other items, for example: :func:`myfunction`. As of Sphinx 0.4, there is a “default_role” that can be set to “obj”, which will polymorphically link to a Python object of any type. This allows one to write `myfunction` instead. This makes docstrings much easier to read and edit as raw text. Additionally, Sphinx allows for setting a current module, so links like `~matplotlib.axes.Axes.set_xlim` could be written as `~axes.Axes.set_xlim`.

Overriding signatures

Many methods in matplotlib use the *args and **kwargs syntax to dynamically handle the keyword arguments that are accepted by the function, or to delegate on to another function. This, however, is often not useful as a signature in the documentation. For this reason, many matplotlib methods include something like:

```
def annotate(self, *args, **kwargs):
    """
    Create an annotation: a piece of text referring to a data point.

    Call signature:

    annotate(s, xy, xytext=None, xycoords='data',
             textcoords='data', arrowprops=None, **kwargs)
    """
```

This can’t be parsed by Sphinx, and is rather verbose in raw text. As of Sphinx 1.1, if the autodoc_docstring_signature config value is set to True, Sphinx will extract a replacement signature from the first line of the docstring, allowing this:
def annotate(self, *args, **kwargs):
    
    """
    annotate(s, xy, xytext=None, xycoords='data',
             textcoords='data', arrowprops=None, **kwargs)
    """

    Create an annotation: a piece of text referring to a data point.
    """

The explicit signature will replace the actual Python one in the generated documentation.

## Linking rather than duplicating

Many of the docstrings include long lists of accepted keywords by interpolating things into the docstring at load time. This makes the docstrings very long. Also, since these tables are the same across many docstrings, it inserts a lot of redundant information in the docs – particularly a problem in the printed version.

These tables should be moved to docstrings on functions whose only purpose is for help. The docstrings that refer to these tables should link to them, rather than including them verbatim.

## autosummary extension

The Sphinx autosummary extension should be used to generate summary tables, that link to separate pages of documentation. Some classes that have many methods (e.g. Axes.axes) should be documented with one method per page, whereas smaller classes should have all of their methods together.

## Examples linking to relevant documentation

The examples, while helpful at illustrating how to use a feature, do not link back to the relevant docstrings. This could be addressed by adding module-level docstrings to the examples, and then including that docstring in the parsed content on the example page. These docstrings could easily include references to any other part of the documentation.

## Documentation using help() vs a browser

Using Sphinx markup in the source allows for good-looking docs in your browser, but the markup also makes the raw text returned using help() look terrible. One of the aims of improving the docstrings should be to make both methods of accessing the docs look good.

### 85.4.5 Implementation

1. The numpydoc extensions should be turned on for matplotlib. There is an important question as to whether these should be included in the matplotlib source tree, or used as a dependency. Installing Numpy is not sufficient to get the numpydoc extensions – it’s a separate install procedure. In any
case, to the extent that they require customization for our needs, we should endeavor to submit those changes upstream and not fork them.

2. Manually go through all of the docstrings and update them to the new format and conventions. Updating the cross references (from `:func:` `myfunc` to `func`) may be able to be semi-automated. This is a lot of busy work, and perhaps this labor should be divided on a per-module basis so no single developer is over-burdened by it.

3. Reorganize the API docs using autosummary and sphinx-autogen. This should hopefully have minimal impact on the narrative documentation.

4. Modify the example page generator (gen_rst.py) so that it extracts the module docstring from the example and includes it in a non-literal part of the example page.

5. Use sphinx-quickstart to generate a new-style Sphinx Makefile. The following features in the current `make.py` will have to be addressed in some other way:
   
   - Copying of some static content
   - Specifying a “small” build (only low-resolution PNG files for examples)

Steps 1, 2, and 3 are interdependent. 4 and 5 may be done independently, though 5 has some dependency on 3.

85.4.6 Backward compatibility

As this mainly involves docstrings, there should be minimal impact on backward compatibility.

85.4.7 Alternatives

None yet discussed.

85.5 MEP11: Third-party dependencies

- Status
- Branches and Pull requests
- Abstract
- Detailed description
  - Current behavior
  - Desired behavior
- Implementation
- Backward compatibility


This MEP attempts to improve the way in which third-party dependencies in matplotlib are handled.

### 85.5.1 Status

**Completed** – needs to be merged

### 85.5.2 Branches and Pull requests

- #1157: Use automatic dependency resolution
- #1290: Debundle pyparsing
- #1261: Update six to 1.2

### 85.5.3 Abstract

One of the goals of matplotlib has been to keep it as easy to install as possible. To that end, some third-party dependencies are included in the source tree and, under certain circumstances, installed alongside matplotlib. This MEP aims to resolve some problems with that approach, bring some consistency, while continuing to make installation convenient.

At the time that was initially done, setuptools, easy_install and PyPI were not mature enough to be relied on. However, at present, we should be able to safely leverage the “modern” versions of those tools, distribute and pip.

While matplotlib has dependencies on both Python libraries and C/C++ libraries, this MEP addresses only the Python libraries so as to not confuse the issue. C libraries represent a larger and mostly orthogonal set of problems.

### 85.5.4 Detailed description

matplotlib depends on the following third-party Python libraries:

- Numpy
- dateutil (pure Python)
- pytz (pure Python)
- six – required by dateutil (pure Python)
- pyparsing (pure Python)
- PIL (optional)
- GUI frameworks: pygtk, gobject, tkinter, PySide, PyQt4, wx (all optional, but one is required for an interactive GUI)
Current behavior

When installing from source, a git checkout or pip:

- setup.py attempts to import numpy. If this fails, the installation fails.
- For each of dateutil, pytz and six, setup.py attempts to import them (from the top-level namespace). If that fails, matplotlib installs its local copy of the library into the top-level namespace.
- pyparsing is always installed inside of the matplotlib namespace.

This behavior is most surprising when used with pip, because no pip dependency resolution is performed, even though it is likely to work for all of these packages.

The fact that pyparsing is installed in the matplotlib namespace has reportedly (#1290) confused some users into thinking it is a matplotlib-related module and import it from there rather than the top-level.

When installing using the Windows installer, dateutil, pytz and six are installed at the top-level always, potentially overwriting already installed copies of those libraries.

TODO: Describe behavior with the OS-X installer.

When installing using a package manager (Debian, RedHat, MacPorts etc.), this behavior actually does the right thing, and there are no special patches in the matplotlib packages to deal with the fact that we handle dateutil, pytz and six in this way. However, care should be taken that whatever approach we move to continues to work in that context.

Maintaining these packages in the matplotlib tree and making sure they are up-to-date is a maintenance burden. Advanced new features that may require a third-party pure Python library have a higher barrier to inclusion because of this burden.

Desired behavior

Third-party dependencies are downloaded and installed from their canonical locations by leveraging pip, distribute and PyPI.

dateutil, pytz, and pyparsing should be made into optional dependencies – though obviously some features would fail if they aren’t installed. This will allow the user to decide whether they want to bother installing a particular feature.

85.5.5 Implementation

For installing from source, and assuming the user has all of the C-level compilers and dependencies, this can be accomplished fairly easily using distribute and following the instructions here. The only anticipated change to the matplotlib library code will be to import pyparsing from the top-level namespace rather than from within matplotlib. Note that distribute will also allow us to remove the direct dependency on six, since it is, strictly speaking, only a direct dependency of dateutil.

For binary installations, there are a number of alternatives (here ordered from best/hardest to worst/easiest):
1. The distutils wininst installer allows a post-install script to run. It might be possible to get this script to run pip to install the other dependencies. (See this thread for someone who has trod that ground before).

2. Continue to ship dateutil, pytz, six and pyparsing in our installer, but use the post-install-script to install them only if they can not already be found.

3. Move all of these packages inside a (new) matplotlib.extern namespace so it is clear for outside users that these are external packages. Add some conditional imports in the core matplotlib codebase so dateutil (at the top-level) is tried first, and failing that matplotlib.extern.dateutil is used.

2 and 3 are undesirable as they still require maintaining copies of these packages in our tree – and this is exacerbated by the fact that they are used less – only in the binary installers. None of these 3 approaches address Numpy, which will still have to be manually installed using an installer.

TODO: How does this relate to the Mac OS-X installer?

85.5.6 Backward compatibility

At present, matplotlib can be installed from source on a machine without the third party dependencies and without an internet connection. After this change, an internet connection (and a working PyPI) will be required to install matplotlib for the first time. (Subsequent matplotlib updates or development work will run without accessing the network).

85.5.7 Alternatives

Distributing binary eggs doesn’t feel like a usable solution. That requires getting easy_install installed first, and Windows users generally prefer the well known exe or msi installer that works out of the box.

85.6 MEP12: Improve Gallery and Examples

- Status
- Branches and Pull requests
- Abstract
- Detailed description
- Implementation
  - Gallery sections
  - Clean up guidelines
    * Additional suggestions
- Backward compatibility
85.6.1 Status

Progress

Initial changes added in 1.3. Conversion of the gallery is on-going. 29 September 2015 - The last `pylab_examples` where `pylab` is imported has been converted over to use `matplotlib pyplot` and `numpy`.

85.6.2 Branches and Pull requests

#1623, #1924, #2181

PR #2474 <https://github.com/matplotlib/matplotlib/pull/2474> demonstrates a single example being cleaned up and moved to the appropriate section.

85.6.3 Abstract

Reorganizing the matplotlib plot gallery would greatly simplify navigation of the gallery. In addition, examples should be cleaned-up and simplified for clarity.

85.6.4 Detailed description

The matplotlib gallery was recently set up to split examples up into sections. As discussed in that PR\(^1\), the current example sections (`api`, `pylab_examples`) aren’t terribly useful to users: New sections in the gallery would help users find relevant examples.

These sections would also guide a cleanup of the examples: Initially, all the current examples would remain and be listed under their current directories. Over time, these examples could be cleaned up and moved into one of the new sections.

This process allows users to easily identify examples that need to be cleaned up; i.e. anything in the `api` and `pylab_examples` directories.

85.6.5 Implementation

1. Create new gallery sections. [Done]
2. Clean up examples and move them to the new gallery sections (over the course of many PRs and with the help of many users/developers). [In progress]

\(^1\) https://github.com/matplotlib/matplotlib/pull/714
Gallery sections

The naming of sections is critical and will guide the clean-up effort. The current sections are:

- Lines, bars, and markers (more-or-less 1D data)
- Shapes and collections
- Statistical plots
- Images, contours, and fields
- Pie and polar charts: Round things
- Color
- Text, labels, and annotations
- Ticks and spines
- Subplots, axes, and figures
- Specialty plots (e.g., sankey, radar, tornado)
- Showcase (plots with tweaks to make them publication-quality)

These names are certainly up for debate. As these sections grow, we should reevaluate them and split them up as necessary.

Clean up guidelines

The current examples in the api and pylab_examples sections of the gallery would remain in those directories until they are cleaned up. After clean-up, they would be moved to one of the new gallery sections described above. “Clean-up” should involve:

- **PEP8** clean-ups (running flake8, or a similar checker, is highly recommended)
- Commented-out code should be removed.
- Add introductory sentence or paragraph in the main docstring. See 6d1b8a2.
- Replace uses of pylab interface with pyplot (+ numpy, etc.). See c25ef1e
- Remove shebang line, e.g.:
  ```
  #!/usr/bin/env python
  ```
- Use consistent imports. In particular:
  ```
  import numpy as np
  import matplotlib.pyplot as plt
  ```
  Avoid importing specific functions from these modules (e.g. from numpy import sin)
• Each example should focus on a specific feature (excluding showcase examples, which will show more “polished” plots). Tweaking unrelated to that feature should be removed. See f7b2217, e57b5fc, and 1458aa8

Use of *pylab* should be demonstrated/discussed on a dedicated help page instead of the gallery examples.

**Note:** When moving an existing example, you should search for references to that example. For example, the API documentation for *axes.py* and *pyplot.py* may use these examples to generate plots. Use your favorite search tool (e.g., grep, ack, grin, pss) to search the matplotlib package. See 2dc9a46 and aa6b410

### Additional suggestions

• Provide links (both ways) between examples and API docs for the methods/objects used. (issue #2222)

• Use `plt.subplots` (note trailing “s”) in preference over `plt.subplot`.

• Rename the example to clarify it’s purpose. For example, the most basic demo of `imshow` might be `imshow_demo.py`, and one demonstrating different interpolation settings would be `imshow_demo_interpolation.py` (*not imshow_demo2.py*).

• Split up examples that try to do too much. See 5099675 and fc2ab07

• Delete examples that don’t show anything new.

• Some examples exercise esoteric features for unit testing. These tweaks should be moved out of the gallery to an example in the *unit* directory located in the root directory of the package.

• Add plot titles to clarify intent of the example. See bd2b13c

### 85.6.6 Backward compatibility

The website for each Matplotlib version is readily accessible, so users who want to refer to old examples can still do so.

### 85.6.7 Alternatives

**Tags**

Tagging examples will also help users search the example gallery. Although tags would be a big win for users with specific goals, the plot gallery will remain the entry point to these examples, and sections could really help users navigate the gallery. Thus, tags are complementary to this reorganization.

### 85.7 MEP13: Use properties for Artists
85.7.1 Status

• Discussion

85.7.2 Branches and Pull requests

None

85.7.3 Abstract

Wrap all of the matplotlib getter and setter methods with python properties, allowing them to be read and written like class attributes.

85.7.4 Detailed description

Currently matplotlib uses getter and setter functions (usually prefixed with get_ and set_, respectively) for reading and writing data related to classes. However, since 2.6 python supports properties, which allow such setter and getter functions to be accessed as though they were attributes. This proposal would implement all existing setter and getter methods as properties.

85.7.5 Implementation

1. All existing getter and setter methods will need to have two aliases, one with the get_ or set_ prefix and one without. Getter methods that currently lack prefixes should be recording in a text file.
2. Classes should be reorganized so setter and getter methods are sequential in the code, with getter methods first.

3. Getter and setter methods the provide additional optional optional arguments should have those arguments accessible in another manner, either as additional getter or setter methods or attributes of other classes. If those classes are not accessible, getters for them should be added.

4. Property decorators will be added to the setter and getter methods without the prefix. Those with the prefix will be marked as deprecated.

5. Docstrings will need to be rewritten so the getter with the prefix has the current docstring and the getter without the prefix has a generic docstring appropriate for an attribute.

6. Automatic alias generation will need to be modified so it will also create aliases for the properties.

7. All instances of getter and setter method calls will need to be changed to attribute access.

8. All setter and getter aliases with prefixes will be removed

The following steps can be done simultaneously: 1, 2, and 3; 4 and 5; 6 and 7.

Only the following steps must be done in the same release: 4, 5, and 6. All other changes can be done in separate releases. 8 should be done several major releases after everything else.

**85.7.6 Backward compatibility**

All existing getter methods that do not have a prefix (such as get_) will need to be changed from function calls to attribute access. In most cases this will only require removing the parenthesis.

setter and getter methods that have additional optional arguments will need to have those arguments implemented in another way, either as a separate property in the same class or as attributes or properties of another class.

Cases where the setter returns a value will need to be changed to using the setter followed by the getter.

Cases where there are set_ATTR_on() and set_ATTR_off() methods will be changed to ATTR_on properties.

**85.7.7 Examples**

`axes.Axes.set_axis_off/set_axis_on`

Current implementation:

```python
axes.Axes.set_axis_off()
axes.Axes.set_axis_on()
```

New implementation:

```python
True = axes.Axes.axis_on
False = axes.Axes.axis_on
axes.Axes.axis_on = True
axes.Axes.axis_on = False
```
axes.Axes.get_xlim/set_xlim and get_autoscalex_on/set_autoscalex_on

Current implementation:

```python
[left, right] = axes.Axes.get_xlim()
auto = axes.Axes.get_autoscalex_on()

[left, right] = axes.Axes.set_xlim(left=left, right=right, emit=emit, auto=auto)
[left, right] = axes.Axes.set_xlim(left=left, right=None, emit=emit, auto=auto)
[left, right] = axes.Axes.set_xlim(left=None, right=right, emit=emit, auto=auto)
[left, right] = axes.Axes.set_xlim(left=left, emit=emit, auto=auto)
[left, right] = axes.Axes.set_xlim(right=right, emit=emit, auto=auto)

axes.Axes.set_autoscalex_on(auto)
```

New implementation:

```python
[left, right] = axes.Axes.axes_xlim
auto = axes.Axes.autoscalex_on

axes.Axes.axes_xlim = [left, right]
axes.Axes.axes_xlim = [left, None]
axes.Axes.axes_xlim = [None, right]
axes.Axes.axes_xlim[0] = left
axes.Axes.axes_xlim[1] = right

axes.Axes.autoscalex_on = auto

axes.Axes.emit_xlim = emit
```

axes.Axes.get_title/set_title

Current implementation:

```python
string = axes.Axes.get_title()
axes.Axes.set_title(string, fontdict=fontdict, **kwargs)
```

New implementation:

```python
string = axes.Axes.title
string = axes.Axes.title_text.text

text.Text = axes.Axes.title_text
text.Text.<attribute> = attribute
text.Text.fontdict = fontdict

axes.Axes.title = string
axes.Axes.title = text.Text
axes.Axes.title_text = string
axes.Axes.title_text = text.Text
```
axes.Axes.get_xticklabels/set_xticklabels

Current implementation:

```
[text.Text] = axes.Axes.get_xticklabels()
[text.Text] = axes.Axes.get_xticklabels(minor=False)
[text.Text] = axes.Axes.get_xticklabels(minor=True)
[text.Text] = axes.Axes([string], fontdict=None, **kwargs)
[text.Text] = axes.Axes([string], fontdict=None, minor=False, **kwargs)
[text.Text] = axes.Axes([string], fontdict=None, minor=True, **kwargs)
```

New implementation:

```
[text.Text] = axes.Axes.xticklabels
[text.Text] = axes.Axes.xminorticklabels
axes.Axes.xticklabels = [string]
axes.Axes.xminorticklabels = [string]
axes.Axes.xticklabels = [text.Text]
axes.Axes.xminorticklabels = [text.Text]
```

85.7.8 Alternatives

Instead of using decorators, it is also possible to use the property function. This would change the procedure so that all getter methods that lack a prefix will need to be renamed or removed. This makes handling docstrings more difficult and harder to read.

It is not necessary to deprecate the setter and getter methods, but leaving them in will complicate the code.

This could also serve as an opportunity to rewrite or even remove automatic alias generation.

Another alternate proposal:

Convert set_xlim, set_xlabel, set_title, etc. to xlim, xlabel, title,... to make the transition from plt functions to axes methods significantly simpler. These would still be methods, not properties, but it’s still a great usability enhancement while retaining the interface.

85.8 MEP14: Text handling
85.8.1 Status

Discussion

85.8.2 Branches and Pull requests

Issue #253 demonstrates a bug where using the bounding box rather than the advance width of text results in misaligned text. This is a minor point in the grand scheme of things, but it should be addressed as part of this MEP.

85.8.3 Abstract

By reorganizing how text is handled, this MEP aims to:

- improve support for Unicode and non-ltr languages
- improve text layout (especially multi-line text)
- allow support for more fonts, especially non-Apple-format TrueType fonts and OpenType fonts.
- make the font configuration easier and more transparent

85.8.4 Detailed description

Text layout

At present, matplotlib has two different ways to render text: “built-in” (based on FreeType and our own Python code), and “usetex” (based on calling out to a TeX installation). Adjunct to the “built-in” renderer there is also the Python-based “mathtext” system for rendering mathematical equations using a subset of the TeX language without having a TeX installation available. Support for these two engines in strewn about many source files, including every backend, where one finds clauses like

```python
if rcParams['text.usetex']:
    # do one thing else:
    # do another
```

Adding a third text rendering approach (more on that later) would require editing all of these places as well, and therefore doesn’t scale.

Instead, this MEP proposes adding a concept of “text engines”, where the user could select one of many different approaches for rendering text. The implementations of each of these would be localized to their own set of modules, and not have little pieces around the whole source tree.

Why add more text rendering engines? The “built-in” text rendering has a number of shortcomings.

- It only handles right-to-left languages, and doesn’t handle many special features of Unicode, such as combining diacriticals.
• The multiline support is imperfect and only supports manual line-breaking – it can not break up a paragraph into lines of a certain length.

• It also does not handle inline formatting changes in order to support something like Markdown, reStructuredText or HTML. (Though rich-text formatting is contemplated in this MEP, since we want to make sure this design allows it, the specifics of a rich-text formatting implementation is outside of the scope of this MEP.)

Supporting these things is difficult, and is the “full-time job” of a number of other projects:

• pango/harfbuzz
• QtTextLayout
• Microsoft DirectWrite
• Apple Core Text

Of the above options, it should be noted that harfbuzz is designed from the start as a cross platform option with minimal dependencies, so therefore is a good candidate for a single option to support.

Additionally, for supporting rich text, we could consider using WebKit, and possibly whether than represents a good single cross-platform option. Again, however, rich text formatting is outside of the scope of this project.

Rather than trying to reinvent the wheel and add these features to matplotlib’s “built-in” text renderer, we should provide a way to leverage these projects to get more powerful text layout. The “built-in” renderer will still need to exist for reasons of ease of installation, but its feature set will be more limited compared to the others. [TODO: This MEP should clearly decide what those limited features are, and fix any bugs to bring the implementation into a state of working correctly in all cases that we want it to work. I know @leejjoon has some thoughts on this.]

Font selection

Going from an abstract description of a font to a file on disk is the task of the font selection algorithm – it turns out to be much more complicated than it seems at first.

The “built-in” and “usetex” renderers have very different ways of handling font selection, given their different technologies. TeX requires the installation of TeX-specific font packages, for example, and can not use TrueType fonts directly. Unfortunately, despite the different semantics for font selection, the same set of font properties are used for each. This is true of both the FontProperties class and the font-related rcParams (which basically share the same code underneath). Instead, we should define a core set of font selection parameters that will work across all text engines, and have engine-specific configuration to allow the user to do engine-specific things when required. For example, it is possible to directly select a font by name in the “built-in” using font.family, but the same is not possible with “usetex”. It may be possible to make it easier to use TrueType fonts by using XeTeX, but users will still want to use the traditional metafonts through TeX font packages. So the issue still stands that different text engines will need engine-specific configuration, and it should be more obvious to the user which configuration will work across text engines and which are engine-specific.

Note that even excluding “usetex”, there are different ways to find fonts. The default is to use the font list cache in font_manager.py which matches fonts using our own algorithm based on the CSS font matching algorithm. It doesn’t always do the same thing as the native font selection algorithms on Linux (fontconfig), Mac and Windows, and it doesn’t always find all of the fonts on the system that the OS would normally
pick up. However, it is cross-platform, and always finds the fonts that ship with matplotlib. The Cairo and MacOSX backends (and presumably a future HTML5-based backend) currently bypass this mechanism and use the OS-native ones. The same is true when not embedding fonts in SVG, PS or PDF files and opening them in a third-party viewer. A downside there is that (at least with Cairo, need to confirm with MacOSX) they don’t always find the fonts we ship with matplotlib. (It may be possible to add the fonts to their search path, though, or we may need to find a way to install our fonts to a location the OS expects to find them).

There are also special modes in the PS and PDF to only use the core fonts that are always available to those formats. There, the font lookup mechanism must only match against those fonts. It is unclear whether the OS-native font lookup systems can handle this case.

There is also experimental support for using fontconfig for font selection in matplotlib, turned off by default. fontconfig is the native font selection algorithm on Linux, but is also cross platform and works well on the other platforms (though obviously is an additional dependency there).

Many of the text layout libraries proposed above (pango, QtTextLayout, DirectWrite and CoreText etc.) insist on using the font selection library from their own ecosystem.

All of the above seems to suggest that we should move away from our self-written font selection algorithm and use the native APIs where possible. That’s what Cairo and MacOSX backends already want to use, and it will be a requirement of any complex text layout library. On Linux, we already have the bones of a fontconfig implementation (which could also be accessed through pango). On Windows and Mac we may need to write custom wrappers. The nice thing is that the API for font lookup is relatively small, and essentially consist of “given a dictionary of font properties, give me a matching font file”.

Font subsetting

Font subsetting is currently handled using ttconv. ttconv was a standalone commandline utility for converting TrueType fonts to subsetted Type 3 fonts (among other features) written in 1995, which matplotlib (well, I) forked in order to make it work as a library. It only handles Apple-style TrueType fonts, not ones with the Microsoft (or other vendor) encodings. It doesn’t handle OpenType fonts at all. This means that even though the STIX fonts come as .otf files, we have to convert them to .ttf files to ship them with matplotlib. The Linux packagers hate this — they’d rather just depend on the upstream STIX fonts. ttconv has also been shown to have a few bugs that have been difficult to fix over time.

Instead, we should be able to use FreeType to get the font outlines and write our own code (probably in Python) to output subsetted fonts (Type 3 on PS and PDF and SVGFonts or paths on SVG). Freetype, as a popular and well-maintained project, handles a wide variety of fonts in the wild. This would remove a lot of custom C code, and remove some code duplication between backends.

Note that subsetting fonts this way, while the easiest route, does lose the hinting in the font, so we will need to continue, as we do now, provide a way to embed the entire font in the file where possible.

Alternative font subsetting options include using the subsetting built-in to Cairo (not clear if it can be used without the rest of Cairo), or using fontforge (which is a heavy and not terribly cross-platform dependency).

Freetype wrappers

Our FreeType wrapper could really use a reworking. It defines its own image buffer class (when a Numpy array would be easier). While FreeType can handle a huge diversity of font files, there are limitations to our wrapper that make it much harder to support non-Apple-vendor TrueType files, and certain features of
OpenType files. (See #2088 for a terrible result of this, just to support the fonts that ship with Windows 7 and 8). I think a fresh rewrite of this wrapper would go a long way.

**Text anchoring and alignment and rotation**

The handling of baselines was changed in 1.3.0 such that the backends are now given the location of the baseline of the text, not the bottom of the text. This is probably the correct behavior, and the MEP refactoring should also follow this convention.

In order to support alignment on multi-line text, it should be the responsibility of the (proposed) text engine to handle text alignment. For a given chunk of text, each engine calculates a bounding box for that text and the offset of the anchor point within that box. Therefore, if the va of a block was “top”, the anchor point would be at the top of the box.

Rotating of text should always be around the anchor point. I’m not sure that lines up with current behavior in matplotlib, but it seems like the sanest/least surprising choice. [This could be revisited once we have something working]. Rotation of text should not be handled by the text engine – that should be handled by a layer between the text engine and the rendering backend so it can be handled in a uniform way. [I don’t see any advantage to rotation being handled by the text engines individually...]

There are other problems with text alignment and anchoring that should be resolved as part of this work. [TODO: enumerate these].

**Other minor problems to fix**

The mathtext code has backend-specific code – it should instead provide its output as just another text engine. However, it’s still desirable to have mathtext layout inserted as part of a larger layout performed by another text engine, so it should be possible to do this. It’s an open question whether embedding the text layout of an arbitrary text engine in another should be possible.

The text mode is currently set by a global rcParam (“text.usetex”) so it’s either all on or all off. We should continue to have a global rcParam to choose the text engine (“text.layout_engine”), but it should under the hood be an overridable property on the Text object, so the same figure can combine the results of multiple text layout engines if necessary.

### 85.8.5 Implementation

A concept of a “text engine” will be introduced. Each text engine will implement a number of abstract classes. The TextFont interface will represent text for a given set of font properties. It isn’t necessarily limited to a single font file – if the layout engine supports rich text, it may handle a number of font files in a family. Given a TextFont instance, the user can get a TextLayout instance, which represents the layout for a given string of text in a given font. From a TextLayout, an iterator over TextSpans is returned so the engine can output raw editable text using as few spans as possible. If the engine would rather get individual characters, they can be obtained from the TextSpan instance:

```python
class TextFont(TextFontBase):
    def __init__(self, font_properties):
        """
        Create a new object for rendering text using the given font properties.
        """
        pass
```
def get_layout(self, s, ha, va):
    """
    Get the TextLayout for the given string in the given font and
    the horizontal (left, center, right) and verticalalignment (top,
    center, baseline, bottom)
    """
    pass

class TextLayout(TextLayoutBase):
    def get_metrics(self):
        """
        Return the bounding box of the layout, anchored at (0, 0).
        """
        pass

    def get_spans(self):
        """
        Returns an iterator over the spans of different in the layout.
        This is useful for backends that want to editable raw text as
        individual lines. For rich text where the font may change,
        each span of different font type will have its own span.
        """
        pass

    def get_image(self):
        """
        Returns a rasterized image of the text. Useful for raster backends,
        like Agg.

        In all likelihood, this will be overridden in the backend, as it can
        be created from get_layout(), but certain backends may want to
        override it if their library provides it (as freetype does).
        """
        pass

    def get_rectangles(self):
        """
        Returns an iterator over the filled black rectangles in the layout.
        Used by TeX and mathtext for drawing, for example, fraction lines.
        """
        pass

    def get_path(self):
        """
        Returns a single Path object of the entire layed out text.

        [Not strictly necessary, but might be useful for textpath
        functionality]
        """
        pass

class TextSpan(TextSpanBase):
    x, y  # Position of the span -- relative to the text layout as a whole
Graphic backends that want to output subset of fonts would likely build up a file-global dictionary of characters where the keys are (fontname, glyph_id) and the values are the paths so that only one copy of the path for each character will be stored in the file.

Special casing: The “usetex” functionality currently is able to get Postscript directly from TeX to insert directly in a Postscript file, but for other backends, parses a DVI file and generates something more abstract. For a case like this, TextLayout would implement get_spans for most backends, but add get_ps for the Postscript backend, which would look for the presence of this method and use it if available, or fall back to get_spans. This kind of special casing may also be necessary, for example, when the graphics backend and text engine belong to the same ecosystem, e.g. Cairo and Pango, or MacOSX and CoreText.

There are three main pieces to the implementation:

1. Rewriting the freetype wrapper, and removing ttconv.
2. Once (1) is done, as a proof of concept, we can move to the upstream STIX .otf fonts
3. Add support for web fonts loaded from a remote URL. (Enabled by using freetype for font subsetting).
4. Refactoring the existing “builtin” and “usetex” code into separate text engines and to follow the API outlined above.
5. Implementing support for advanced text layout libraries.
(1) and (2) are fairly independent, though having (1) done first will allow (2) to be simpler. (3) is dependent on (1) and (2), but even if it doesn’t get done (or is postponed), completing (1) and (2) will make it easier to move forward with improving the “builtin” text engine.

85.8.6 Backward compatibility

The layout of text with respect to its anchor and rotation will change in hopefully small, but improved, ways. The layout of multiline text will be much better, as it will respect horizontal alignment. The layout of bidirectional text or other advanced Unicode features will now work inherently, which may break some things if users are currently using their own workarounds.

Fonts will be selected differently. Hacks that used to sort of work between the “builtin” and “usetex” text rendering engines may no longer work. Fonts found by the OS that weren’t previously found by matplotlib may be selected.

85.8.7 Alternatives

TBD

85.9 MEP15 - Fix axis autoscaling when limits are specified for one axis only

85.9.1 Status

Discussion

85.9.2 Branches and Pull requests

None so far.
85.9.3 Abstract

When one axis of a 2-dimensional plot if overridden via xlim or ylim, automatic scaling of the remaining axis should be based on the data that falls within the specified limits of the first axis.

85.9.4 Detailed description

When axis limits for a 2-D plot are specified for one axis only (via xlim or ylim), matplotlib currently does not currently rescale the other axis. The result is that the displayed curves or symbols may be compressed into a tiny portion of the available area, so that the final plot conveys much less information than it would with appropriate axis scaling.

The proposed change of behavior would make matplotlib choose the scale for the remaining axis using only the data that falls within the limits for the axis where limits were specified.

85.9.5 Implementation

I don’t know enough about the internals of matplotlib to be able to suggest an implementation.

85.9.6 Backward compatibility

From the standpoint of software interfaces, there would be no break in backward compatibility. Some outputs would be different, but if the user truly desires the previous behavior, he/she can achieve this by overriding the axis scaling for both axes.

85.9.7 Alternatives

The only alternative that I can see is to maintain the status quo.

85.10 MEP19: Continuous Integration

85.10.1 Status

Discussion

85.10.2 Branches and Pull requests

85.10.3 Abstract

matplotlib could benefit from better and more reliable continuous integration, both for testing and building installers and documentation.
85.10.4 Detailed description

Current state-of-the-art

Testing

matplotlib currently uses Travis-CI for automated tests. While Travis-CI should be praised for how much it does as a free service, it has a number of shortcomings:

- It often fails due to network timeouts when installing dependencies.
- It often fails for inexplicable reasons.
- Build or test products can only be saved from build off of branches on the main repo, not pull requests, so it is often difficult to “post mortem” analyse what went wrong. This is particularly frustrating when the failure can not be subsequently reproduced locally.
- It is not extremely fast. matplotlib’s cpu and memory requirements for testing are much higher than the average Python project.
- It only tests on Ubuntu Linux, and we have only minimal control over the specifics of the platform. It can be upgraded at any time outside of our control, causing unexpected delays at times that may not be convenient in our release schedule.

On the plus side, Travis-CI’s integration with github – automatically testing all pending pull requests – is exceptional.

Builds

There is no centralized effort for automated binary builds for matplotlib. However, the following disparate things are being done [If the authors mentioned here could fill in detail, that would be great!]:

- @sandrotosi: builds Debian packages
- @takluyver: Has automated Ubuntu builds on Launchpad
- @cgothke: Makes Windows builds (don’t know how automated that is)
- @r-owen: Makes OS-X builds (don’t know how automated that is)

Documentation

Documentation of master is now built by travis and uploaded to http://matplotlib.org/devdocs/index.html @NelleV, I believe, generates the docs automatically and posts them on the web to chart MEP10 progress.

Peculiarities of matplotlib

matplotlib has complex requirements that make testing and building more taxing than many other Python projects.

- The CPU time to run the tests is quite high. It puts us beyond the free accounts of many CI services (e.g. ShiningPanda)
- It has a large number of dependencies, and testing the full matrix of all combinations is impractical. We need to be clever about what space we test and guarantee to support.
Requirements

This section outlines the requirements that we would like to have.

1. Testing all pull requests by hooking into the Github API, as Travis-CI does

2. Testing on all major platforms: Linux, Mac OS-X, MS Windows (in that order of priority, based on user survey)

3. Retain the last n days worth of build and test products, to aid in post-mortem debugging.

4. Automated nightly binary builds, so that users can test the bleeding edge without installing a complete compilation environment.

5. Automated benchmarking. It would be nice to have a standard benchmark suite (separate from the tests) whose performance could be tracked over time, in different backends and platforms. While this is separate from building and testing, ideally it would run on the same infrastructure.

6. Automated nightly building and publishing of documentation (or as part of testing, to ensure PRs don’t introduce documentation bugs). (This would not replace the static documentation for stable releases as a default).

7. The test systems should be manageable by multiple developers, so that no single person becomes a bottleneck. (Travis-CI’s design does this well – storing build configuration in the git repository, rather than elsewhere, is a very good design.)

8. Make it easy to test a large but sparse matrix of different versions of matplotlib’s dependencies. The matplotlib user survey provides some good data as to where to focus our efforts: https://docs.google.com/spreadsheet/ccc?key=0AjrPjTMRTwTdHpQS25pcTZIRWdqX0pNckNSU01sMHc#gid=0

9. Nice to have: A decentralized design so that those with more obscure platforms can publish build results to a central dashboard.

**85.10.5 Implementation**

This part is yet-to-be-written.

However, ideally, the implementation would be a third-party service, to avoid adding system administration to our already stretched time. As we have some donated funds, this service may be a paid one if it offers significant time-saving advantages over free offerings.

**85.10.6 Backward compatibility**

Backward compatibility is not a major concern for this MEP. We will replace current tools and procedures with something better and throw out the old.

**85.10.7 Alternatives**

**85.10.8 Hangout Notes**
**CI Infrastructure**

- We like Travis and it will probably remain part of our arsenal in any event. The reliability issues are being looked into.

- Enable Amazon S3 uploads of testing products on Travis. This will help with post-mortem of failures (@mdboom is looking into this now).

- We want Mac coverage. The best bet is probably to push Travis to enable it for our project by paying them for a Pro account (since they don’t otherwise allow testing on both Linux and Mac).

- We want Windows coverage. Shining Panda is an option there.

- Investigate finding or building a tool that would collect and synthesize test results from a number of sources and post it to Github using the Github API. This may be of general use to the Scipy community.

- For both Windows and Mac, we should document (or better yet, script) the process of setting up the machine for a build, and how to build binaries and installers. This may require getting information from Russel Owen and Christoph Gohlke. This is a necessary step for doing automated builds, but would also be valuable for a number of other reasons.

**The test framework itself**

- We should investigate ways to make it take less time
  - Eliminating redundant tests, if possible
  - General performance improvements to matplotlib will help

- We should be covering more things, particularly more backends

- We should have more unit tests, fewer integration tests, if possible

**85.11 MEP21: color and cm refactor**

- **Status**
- **Branches and Pull requests**
- **Abstract**
- **Detailed description**
- **Implementation**
- **Backward compatibility**
- **Alternatives**
85.11.1 Status

- **Discussion:** This MEP has not commenced yet, but here are some ongoing ideas which may become a part of this MEP:

85.11.2 Branches and Pull requests

85.11.3 Abstract

- color
  - tidy up the namespace
  - Define a “Color” class
  - make it easy to convert from one color type to another `hex -> RGB`, `RGB -> hex`, `HSV -> RGB` etc.
  - improve the construction of a colormap - the dictionary approach is archaic and overly complex (though incredibly powerful)
  - make it possible to interpolate between two or more color types in different modes, especially useful for construction of colormaps in HSV space for instance

- cm
  - rename the module to something more descriptive - mappables?

Overall, there are a lot of improvements that can be made with matplotlib color handling - managing backwards compatibility will be difficult as there are some badly named variables/modules which really shouldn’t exist - but a clear path and message for migration should be available, with a large amount of focus on this in the API changes documentation.

85.11.4 Detailed description

85.11.5 Implementation

85.11.6 Backward compatibility

85.11.7 Alternatives

85.12 MEP22: Toolbar rewrite

85.12. MEP22: Toolbar rewrite
• Detailed description

• Implementation
  – ToolBase(object)
  – ToolToggleBase(ToolBase)
  – NavigationBase
  – ToolbarBase

• Backward compatibility

85.12.1 Status

Progress

85.12.2 Branches and Pull requests

Previous work

• https://github.com/matplotlib/matplotlib/pull/1849
• https://github.com/matplotlib/matplotlib/pull/2557
• https://github.com/matplotlib/matplotlib/pull/2465

Pull Requests:

• Removing the NavigationToolbar classes https://github.com/matplotlib/matplotlib/pull/2740
  CLOSED

• Keeping the NavigationToolbar classes https://github.com/matplotlib/matplotlib/pull/2759
  CLOSED

• Navigation by events: https://github.com/matplotlib/matplotlib/pull/3652

85.12.3 Abstract

The main goal of this MEP is to make it easier to modify (add, change, remove) the way the user interacts with the figures.

The user interaction with the figure is deeply integrated within the Canvas and Toolbar. Making extremely difficult to do any modification.

This MEP proposes the separation of this interaction into Toolbar, Navigation and Tools to provide independent access and reconfiguration.

This approach will make easier to create and share tools among users. In the far future, we can even foresee a kind of Marketplace for Tools where the most popular can be added into the main distribution.
85.12.4 Detailed description

The reconfiguration of the Toolbar is complex, most of the time it requires a custom backend.

The creation of custom Tools sometimes interferes with the Toolbar, as example see https://github.com/matplotlib/matplotlib/issues/2694 also the shortcuts are hardcoded and again not easily modifiable https://github.com/matplotlib/matplotlib/issues/2699

The proposed solution is to take the actions out of the Toolbar and the shortcuts out of the Canvas. This actions and shortcuts will be in the form of Tools.

A new class Navigation will be the bridge between the events from the Canvas and Toolbar and redirect them to the appropriate Tool.

At the end the user interaction will be divided into three classes:

- NavigationBase: This class is instantiated for each FigureManager and connect the all user interactions with the Tools
  - ToolbarBase: This existing class is relegated only as a GUI access to Tools.
  - ToolBase: Is the basic definition of Tools.

85.12.5 Implementation

ToolBase(object)

Tools can have a graphical representation as the SubplotTool or not even be present in the Toolbar as Quit

The ToolBase has the following class attributes for configuration at definition time

- keymap = None: Key(s) to be used to trigger the tool
- description = ‘’: Small description of the tool
- image = None: Image that is used in the toolbar

The following instance attributes are set at instantiation:

- name
- navigation

Methods

- trigger(self, event): This is the main method of the Tool, it is called when the Tool is triggered by: * Toolbar button click * keypress associated with the Tool Keymap * Call to navigation.trigger_tool(name)
- set_figure(self, figure): Set the figure and navigation attributes
- destroy(self, *args): Destroy the Tool graphical interface (if exists)

Available Tools

- ToolQuit
Matplotlib, Release 2.0.2

- ToolEnableAllNavigation
- ToolEnableNavigation
- ToolToggleGrid
- ToolToggleFullScreen
- ToolToggleYScale
- ToolToggleXScale
- ToolHome
- ToolBack
- ToolForward
- SaveFigureBase
- ConfigureSubplotsBase

**ToolToggleBase(ToolBase)**

The **ToolToggleBase** has the following class attributes for configuration at definition time

- radio_group = None: Attribute to group ‘radio’ like tools (mutually exclusive)
- cursor = None: Cursor to use when the tool is active

The **Toggleable** Tools, can capture keypress, mouse moves, and mouse button press

It defines the following methods

- enable(self, event): Called by **ToolToggleBase.trigger** method
- disable(self, event): Called when the tool is untoggled
- toggled : Property True or False

**Available Tools**

- ToolZoom
- ToolPan

**NavigationBase**

Defines the following attributes

- canvas:
- keypresslock: Lock to know if the canvas key_press_event is available and process it
- messagelock: Lock to know if the message is available to write

Public methods for User use:

- nav_connect(self, s, func): Connect to navigation for events
• nav_disconnect(self, cid): Disconnect from navigation event
• message_event(self, message, sender=None): Emit a tool_message_event event
• active_toggle(self): Property The currently toggled tools or None
• get_tool_keymap(self, name): Return a list of keys that are associated with the tool
• set_tool_keymap(self, name, *keys): Set the keys for the given tool
• remove_tool(self, name): Removes tool from the navigation control.
• add_tools(self, tools): Add multiple tools to Navigation
• add_tool(self, name, tool, group=None, position=None): Add a tool to the Navigation
• tool_trigger_event(self, name, sender=None, canvasevent=None, data=None): Trigger a tool and fire the event
• tools(self) Property: Return a dict with available tools with corresponding keymaps, descriptions and objects
• get_tool(self, name): Return the tool object

**ToolbarBase**

**Methods for Backend implementation**

• add_toolitem(self, name, group, position, image, description, toggle): Add a toolitem to the toolbar. This method is a callback from tool_added_event (emited by navigation)
• set_message(self, s): Display a message on toolbar or in status bar
• toggle_toolitem(self, name): Toggle the toolitem without firing event.
• remove_toolitem(self, name): Remove a toolitem from the Toolbar

**85.12.6 Backward compatibility**

For backward compatibility added a ‘navigation’ key to rcsetup.validate_toolbar, that is used for Navigation classes instantiation instead of the NavigationToolbar classes

With this parameter, it makes it transparent to anyone using the existing backends.

[@pelson comment: This also gives us an opportunity to avoid needing to implement all of this in the same PR - some backends can potentially exist without the new functionality for a short while (but it must be done at some point).]

**85.13 MEP23: Multiple Figures per GUI window**
### 85.13.1 Status

**Discussion**

### 85.13.2 Branches and Pull requests

**Previous work** - https://github.com/matplotlib/matplotlib/pull/2465 To-delete

### 85.13.3 Abstract

Add the possibility to have multiple figures grouped under the same `FigureManager`

### 85.13.4 Detailed description

Under the current structure, every canvas has its own window.

This is and may continue to be the desired method of operation for most use cases.

Sometimes when there are too many figures open at the same time, it is desirable to be able to group these under the same window [see](https://github.com/matplotlib/matplotlib/issues/2194).

The proposed solution modifies `FigureManagerBase` to contain and manage more than one canvas. The settings parameter `rcParams['backend.multifigure']` control when the `MultiFigure` behaviour is desired.

**Note**
It is important to note, that the proposed solution, assumes that the [MEP22](https://github.com/matplotlib/matplotlib/wiki/Mep22) is already in place. This is simply because the actual implementation of the Toolbar makes it pretty hard to switch between canvases.

**85.13.5 Implementation**

The first implementation will be done in GTK3 using a Notebook as canvas container.

**FigureManagerBase**

will add the following new methods

- `add_canvas`: To add a canvas to an existing FigureManager object
- `remove_canvas`: To remove a canvas from a FigureManager object, if it is the last one, it will be destroyed
- `move_canvas`: To move a canvas from one FigureManager to another.
- `set_canvas_title`: To change the title associated with a specific canvas container
- `get_canvas_title`: To get the title associated with a specific canvas container
- `get_active_canvas`: To get the canvas that is in the foreground and is subject to the gui events. There is no `set_active_canvas` because the active canvas, is defined when `show` is called on a Canvas object.

**new_figure_manager**

To control which FigureManager will contain the new figures, an extra optional parameter `figuremanager` will be added, this parameter value will be passed to `new_figure_manager_given_figure`

**new_figure_manager_given_figure**

- If `figuremanager` parameter is give, this FigureManager object will be used instead of creating a new one.
- If `rcParams['backend.multifigure'] == True`: The last FigureManager object will be used instead of creating a new one.

**NavigationBase**

Modifies the NavigationBase to keep a list of canvases, directing the actions to the active one
85.13.6 Backward compatibility

For the MultiFigure properties to be visible, the user has to activate them directly setting `rcParams['backend.multifigure'] = True`

It should be backwards compatible for backends that adhere to the current FigureManagerBase structure even if they have not implemented the MultiFigure magic yet.

85.13.7 Alternatives

Instead of modifying the FigureManagerBase it could be possible to add a parallel class, that handles the cases where `rcParams['backend.multifigure'] = True`. This will warranty that there won’t be any problems with custom made backends, but also makes bigger the code, and more things to maintain.

85.14 MEP24: negative radius in polar plots

- Status
- Branches and Pull requests
- Abstract
- Detailed description
- Implementation
- Related Issues
- Backward compatibility
- Alternatives

85.14.1 Status

Discussion

85.14.2 Branches and Pull requests

None

85.14.3 Abstract

It is clear that polar plots need to be able to gracefully handle negative r values (not by clipping or reflection).
85.14.4 Detailed description

One obvious application that we should support is bB plots (see https://github.com/matplotlib/matplotlib/issues/1730#issuecomment-40815837), but this seems more generally useful (for example growth rate as a function of angle). The assumption in the current code (as I understand it) is that the center of the graph is r==0, however it would be good to be able to set the center to be at any r (with any value less than the off set clipped).

85.14.5 Implementation

85.14.6 Related Issues

#1730, #1603, #2203, #2133

85.14.7 Backward compatibility

85.14.8 Alternatives

85.15 MEP25: Serialization

- Status
- Branches and Pull requests
- Abstract
- Detailed description
- Examples
- Implementation
- Backward compatibility
- Alternatives

85.15.1 Status

Discussion

85.15.2 Branches and Pull requests

- development branches:
- related pull requests:
85.15.3 Abstract

This MEP aims at adding a serializable Controller objects to act as an Artist managers. Users would then communicate changes to an Artist via a Controller. In this way, functionality of the Controller objects may be added incrementally since each Artist is still responsible for drawing everything. The goal is to create an API that is usable both by graphing libraries requiring high-level descriptions of figures and libraries requiring low-level interpretations.

85.15.4 Detailed description

Matplotlib is a core plotting engine with an API that many users already understand. It’s difficult/impossible for other graphing libraries to (1) get a complete figure description, (2) output raw data from the figure object as the user has provided it, (3) understand the semantics of the figure objects without heuristics, and (4) give matplotlib a complete figure description to visualize. In addition, because an Artist has no conception of its own semantics within the figure, it’s difficult to interact with them in a natural way.

In this sense, matplotlib will adopt a standard Model-View-Controller (MVC) framework. The Model will be the user defined data, style, and semantics. The Views are the ensemble of each individual Artist, which are responsible for producing the final image based on the model. The Controller will be the Controller object managing its set of Artist objects.

The Controller must be able to export the information that it’s carrying about the figure on command, perhaps via a to_json method or similar. Because it would be extremely extraneous to duplicate all of the information in the model with the controller, only user-specified information (data + style) are explicitly kept. If a user wants more information (defaults) from the view/model, it should be able to query for it.

- This might be annoying to do, non-specified kwargs are pulled from the rcParams object which is in turn created from reading a user specified file and can be dynamically changed at run time. I suppose we could keep a dict of default defaults and compare against that. Not clear how this will interact with the style sheet [[MEP26]] - @tacaswell

Additional Notes:

- The raw data does not necessarily need to be a list, ndarray, etc. Rather, it can more abstractly just have a method to yield data when needed.

- Because the Controller will contain extra information that users may not want to keep around, it should not be created by default. You should be able to both (a) instantiate a Controller with a figure and (b) build a figure with a Controller.

Use Cases:

- Export all necessary informat

- Serializing a matplotlib figure, saving it, and being able to rerun later.

- Any other source sending an appropriately formatted representation to matplotlib to open

85.15.5 Examples

Here are some examples of what the controllers should be able to do.
1. Instantiate a matplotlib figure from a serialized representation (e.g., JSON):

```python
import json
from matplotlib.controllers import Controller
with open('my_figure') as f:
    o = json.load(f)
    c = Controller(o)
    fig = c.figure
```

2. Manage artists from the controller (e.g., Line2D):

```python
# not really sure how this should look
    c.axes[0].lines[0].color = 'b'
# ?
```

3. Export serializable figure representation:

```python
    o = c.to_json()
    # or... we should be able to throw a figure object in there too
    o = Controller.to_json(mpl_fig)
```

### 8.5.15.6 Implementation

1. Create base Controller objects that are able to manage Artist objects (e.g., Hist)

   Comments:
   - initialization should happen via unpacking **, so we need a copy of call signature parameter for the Artist we’re ultimately trying to control. Unfortunate hard-coded repetition...
   - should the additional **kwargs accepted by each Artist be tracked at the Controller
   - how does a Controller know which artist belongs where? E.g., do we need to pass axes references?

   Progress:
   - A simple NB demonstrating some functionality for Line2DController objects: https://nbviewer.jupyter.org/gist/theengineear/f0aa8d79f64325e767c0

2. Write in protocols for the Controller to update the model.

   Comments:
   - how should containers be dealt with? E.g., what happens to old patches when we re-bin a histogram?
   - in the link from (1), the old line is completely destroyed and redrawn, what if something is referencing it?

3. Create method by which a json object can be assembled from the Controllers

4. Deal with serializing the unserializable aspects of a figure (e.g., non-affine transforms?)
5. Be able to instantiate from a serialized representation
6. Reimplement the existing pyplot and Axes method, e.g. `pyplot.hist` and `Axes.hist` in terms of the new controller class.

> @theengineer: in #2 above, what do you mean by *get updates* from each *Artist*?

^ Yup. The Controller *shouldn’t* need to get updated. This just happens in #3. Delete comments when you see this.

### 85.15.7 Backward compatibility

- pickling will change
- non-affine transformations will require a defined pickling method

### 85.15.8 Alternatives

PR #3150 suggested adding semantics by parasitically attaching extra containers to axes objects. This is a more complete solution with what should be a more developed/flexible/powerful framework.

### 85.16 MEP26: Artist styling

- **Status**
- **Branches and Pull requests**
- **Abstract**
- **Detailed description**
- **Implementation**
  - BNF Grammar
  - Syntax
    * Selectors
    * GID selector
    * Attributes and values
  - Parsing
  - Visitor pattern for matplotlib figure
- **Backward compatibility**
- **Alternatives**
- **Appendix**
85.16.1 Status

Proposed

85.16.2 Branches and Pull requests

85.16.3 Abstract

This MEP proposes a new stylesheet implementation to allow more comprehensive and dynamic styling of artists.

The current version of matplotlib (1.4.0) allows stylesheets based on the rcParams syntax to be applied before creation of a plot. The methodology below proposes a new syntax, based on CSS, which would allow styling of individual artists and properties, which can be applied dynamically to existing objects.

This is related to (and makes steps toward) the overall goal of moving to a DOM/tree-like architecture.

85.16.4 Detailed description

Currently, the look and appearance of existing artist objects (figure, axes, Line2D etc...) can only be updated via set_ and get_ methods on the artist object, which is quite laborious, especially if no reference to the artist(s) has been stored. The new style sheets introduced in 1.4 allow styling before a plot is created, but do not offer any means to dynamically update plots or distinguish between artists of the same type (i.e. to specify the line color and line style separately for differing Line2D objects).

The initial development should concentrate on allowing styling of artist primitives (those artists that do not contain other artists), and further development could expand the CSS syntax rules and parser to allow more complex styling. See the appendix for a list of primitives.

The new methodology would require development of a number of steps:

- A new stylesheet syntax (likely based on CSS) to allow selection of artists by type, class, id etc...
- A mechanism by which to parse a stylesheet into a tree
- A mechanism by which to translate the parse-tree into something which can be used to update the properties of relevant artists. Ideally this would implement a method by which to traverse the artists in a tree-like structure.
- A mechanism by which to generate a stylesheet from existing artist properties. This would be useful to allow a user to export a stylesheet from an existing figure (where the appearance may have been set using the matplotlib API)...

85.16. MEP26: Artist styling 2009
85.16.5 Implementation

It will be easiest to allow a ‘3rd party’ to modify/set the style of an artist if the ‘style’ is created as a separate class and store against the artist as a property. The GraphicsContext class already provides a basis for a Style class and an artist’s draw method can be refactored to use the Style class rather than setting up it’s own GraphicsContext and transferring it’s style-related properties to it. A minimal example of how this could be implemented is shown here: https://github.com/JamesRamm/mpl_experiment

IMO, this will also make the API and code base much neater as individual get/set methods for artist style properties are now redundant... Indirectly related would be a general drive to replace get/set methods with properties. Implementing the style class with properties would be a big stride toward this...

For initial development, I suggest developing a syntax based on a much (much much) simplified version of CSS. I am in favour of dubbing this Artist Style Sheets :+1: :

BNF Grammar

I propose a very simple syntax to implement initially (like a proof of concept), which can be expanded upon in the future. The BNF form of the syntax is given below and then explained

| RuleSet ::= SelectorSequence "{"Declaration"}" |
| SelectorSequence ::= = Selector {"," Selector} |
| Declaration ::= propName":" propValue";" |
| Selector ::= ArtistIdent "#"Ident |
| propName ::= Ident |
| propValue ::= Ident | Number | Colour | "None" |

ArtistIdent, Ident, Number and Colour are tokens (the basic building blocks of the expression) which are defined by regular expressions.

Syntax

A CSS stylesheet consists of a series of rule sets in hierarchical order (rules are applied from top to bottom). Each rule follows the syntax

selector {attribute: value;}

Each rule can have any number of attribute: value pairs, and a stylesheet can have any number of rules.

The initial syntax is designed only for artist primitives. It does not address the question of how to set properties on container types (whose properties may themselves be artists with settable properties), however, a future solution to this could simply be nested RuleSet s
Selectors

Selectors define the object to which the attribute updates should be applied. As a starting point, I propose just 2 selectors to use in initial development:

Artist Type Selector
Select an artist by it’s type. E.g Line2D or Text:

```
Line2D {attribute: value}
```

The regex for matching the artist type selector (ArtistIdent in the BNF grammar) would be:

```
ArtistIdent = r'(?P<ArtistIdent>Line2D|bText|bAxesImage|bFigureImage|bPatch)'
```

GID selector

Select an artist by its gid:

```
Line2D#myGID {attribute: value}
```

A gid can be any string, so the regex could be as follows:

```
Ident = r'(?P<Ident>[a-zA-Z_]+[a-zA-Z_0-9]*)'
```

The above selectors roughly correspond to their CSS counterparts (http://www.w3.org/TR/CSS21/selector.html)

Attributes and values

- Attributes are any valid (settable) property for the artist in question.
- Values are any valid value for the property (Usually a string, or number).

Parsing

Parsing would consist of breaking the stylesheet into tokens (the python cookbook gives a nice tokenizing recipe on page 66), applying the syntax rules and constructing a Tree. This requires defining the grammar of the stylesheet (again, we can borrow from CSS) and writing a parser. Happily, there is a recipe for this in the python cookbook aswell.

Visitor pattern for matplotlib figure

In order to apply the stylesheet rules to the relevant artists, we need to ‘visit’ each artist in a figure and apply the relevant rule. Here is a visitor class (again, thanks to python cookbook), where each node would be
an artist in the figure. A visit_ method would need to be implemented for each mpl artist, to handle the different properties for each

```python
class Visitor:
    def visit(self, node):
        name = 'visit_' + type(node).__name__
        meth = getattr(self, name, None)
        if meth is None:
            raise NotImplementedError
        return meth(node)
```

An evaluator class would then take the stylesheet rules and implement the visitor on each one of them.

**85.16.6 Backward compatibility**

Implementing a separate Style class would break backward compatibility as many get/set methods on an artist would become redundant. While it would be possible to alter these methods to hook into the Style class (stored as a property against the artist), I would be in favor of simply removing them to both neaten/simplify the codebase and to provide a simple, uncluttered API...

**85.16.7 Alternatives**

No alternatives, but some of the ground covered here overlaps with MEP25, which may assist in this development

**85.16.8 Appendix**

Matplotlib primitives

This will form the initial selectors which stylesheets can use.

- Line2D
- Text
- AxesImage
- FigureImage
- Patch

**85.17 MEP27: decouple pyplot from backends**

- Status
- Branches and Pull requests
85.17.1 Status

Discussion

85.17.2 Branches and Pull requests

Main PR (including GTK3): + https://github.com/matplotlib/matplotlib/pull/4143

Backend specific branch diffs: + https://github.com/OceanWolf/matplotlib/compare/backend-refactor...
OceanWolf:backend-refactor-qt + https://github.com/OceanWolf/matplotlib/compare/backend-refactor...
backend-refactor-wx

85.17.3 Abstract

This MEP refactors the backends to give a more structured and consistent API, removing generic code and consolidate existing code. To do this we propose splitting:

1. FigureManagerBase and its derived classes into the core functionality class FigureManager and a backend specific class WindowBase and

2. ShowBase and its derived classes into Gcf.show_all and MainLoopBase.

85.17.4 Detailed description

This MEP aims to consolidate the backends API into one single uniform API, removing generic code out of the backend (which includes _pylab_helpers and Gcf), and push code to a more appropriate level in matplotlib. With this we automatically remove inconsistencies that appear in the backends, such as FigureManagerBase.resize(w, h) which sometimes sets the canvas, and other times set the entire window to the dimensions given, depending on the backend.

Two main places for generic code appear in the classes derived from FigureManagerBase and ShowBase.

1. FigureManagerBase has **three** jobs at the moment:
(a) The documentation describes it as a ‘‘Helper class for pyplot mode, wraps everything up into a neat bundle’’

(b) But it doesn’t just wrap the canvas and toolbar, it also does all of the windowing tasks itself. The conflation of these two tasks gets seen the best in the following line: `python self.
set_window_title("Figure %d" % num) ` This combines backend specific code `self.
set_window_title(title)` with matplotlib generic code `title = "Figure %d" % num.`

(c) Currently the backend specific subclass of `FigureManager` decides when to end the mainloop. This also seems very wrong as the figure should have no control over the other figures.

2. `ShowBase` has two jobs:

   (a) It has the job of going through all figure managers registered in `_pylab_helpers.Gcf` and telling them to show themselves.

   (b) And secondly it has the job of performing the backend specific `mainloop` to block the main programme and thus keep the figures from dying.

### 85.17.5 Implementation

The description of this MEP gives us most of the solution:

1. To remove the windowing aspect out of `FigureManagerBase` letting it simply wrap this new class along with the other backend classes. Create a new `WindowBase` class that can handle this functionality, with pass-through methods (`:arrow_right:`) to `WindowBase`. Classes that subclass `WindowBase` should also subclass the GUI specific window class to ensure backward compatibility (`manager.
window == manager.window`).

2. Refactor the mainloop of `ShowBase` into `MainLoopBase`, which encapsulates the end of the loop as well. We give an instance of `MainLoop` to `FigureManager` as a key unlock the exit method (requiring all keys returned before the loop can die). Note this opens the possibility for multiple backends to run concurrently.

3. Now that `FigureManagerBase` has no backend specifics in it, to rename it to `FigureManager`, and move to a new file `backend_managers.py` noting that:

   (a) This allows us to break up the conversion of backends into separate PRs as we can keep the existing `FigureManagerBase` class and its dependencies intact.

   (b) and this also anticipates MEP22 where the new `NavigationBase` has morphed into a backend independent `ToolManager`.
<table>
<thead>
<tr>
<th>FigureManager-Base(canvas, num)</th>
<th>FigureManager(figure, num)</th>
<th>WindowBase(title)</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>show</td>
<td>show</td>
<td></td>
<td></td>
</tr>
<tr>
<td>destroy</td>
<td>calls destroy on all components</td>
<td>destroy</td>
<td></td>
</tr>
<tr>
<td>full_screen_toggle</td>
<td>handles logic</td>
<td>set_fullscreen</td>
<td></td>
</tr>
<tr>
<td>resize</td>
<td>resize</td>
<td></td>
<td></td>
</tr>
<tr>
<td>key_press</td>
<td>key_press</td>
<td></td>
<td></td>
</tr>
<tr>
<td>show_popup</td>
<td>show_poup</td>
<td></td>
<td></td>
</tr>
<tr>
<td>get_window_title</td>
<td>get_window_title</td>
<td>set_window_title</td>
<td></td>
</tr>
<tr>
<td>set_window_title</td>
<td>set_window_title</td>
<td>_get_toolbar</td>
<td>A common method to all subclasses of FigureManagerBase</td>
</tr>
<tr>
<td></td>
<td></td>
<td>set_default_size</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>add_element_to_window</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ShowBase</th>
<th>MainLoopBase</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>mainloop</td>
<td>begin</td>
<td></td>
</tr>
<tr>
<td></td>
<td>end</td>
<td>Gets called automagically when no more instances of the subclass exist</td>
</tr>
<tr>
<td><strong>call</strong></td>
<td></td>
<td>Method moved to Gcf.show_all</td>
</tr>
</tbody>
</table>

**85.17.6 Future compatibility**

As eluded to above when discussing MEP 22, this refactor makes it easy to add in new generic features. At the moment, MEP 22 has to make ugly hacks to each class extending from FigureManagerBase. With this code, this only needs to get made in the single FigureManager class. This also makes the later deprecation of NavigationToolbar2 very straightforward, only needing to touch the single FigureManager class.

MEP 23 makes for another use case where this refactored code will come in very handy.

**85.17.7 Backward compatibility**

As we leave all backend code intact, only adding missing methods to existing classes, this should work seamlessly for all use cases. The only difference will lie for backends that used FigureManager.resize to resize the canvas and not the window, due to the standardisation of the API.

I would envision that the classes made obsolete by this refactor get deprecated and removed on the same timetable as NavigationToolbar2, also note that the change in call signature to the FigureCanvasWx constructor, while backward compatible, I think the old (imho ugly style) signature should get deprecated and removed in the same manner as everything else.

---

85.17. MEP27: decouple pyplot from backends 2015
<table>
<thead>
<tr>
<th>backend</th>
<th>manager.resize(w,h)</th>
<th>Extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>gtk3</td>
<td>window</td>
<td></td>
</tr>
<tr>
<td>Tk</td>
<td>canvas</td>
<td></td>
</tr>
<tr>
<td>Qt</td>
<td>window</td>
<td></td>
</tr>
<tr>
<td>Wx</td>
<td>canvas</td>
<td>FigureManagerWx had frame as an alias to window, so this also breaks BC.</td>
</tr>
</tbody>
</table>

### 85.17.8 Alternatives

If there were any alternative solutions to solving the same problem, they should be discussed here, along with a justification for the chosen approach.

### 85.17.9 Questions

Mdehoon: Can you elaborate on how to run multiple backends concurrently?

OceanWolf: @mdehoon, as I say, not for this MEP, but I see this MEP opens it up as a future possibility. Basically the MainLoopBase class acts a per backend Gcf, in this MEP it tracks the number of figures open per backend, and manages the mainloops for those backends. It closes the backend specific mainloop when it detects that no figures remain open for that backend. Because of this I imagine that with only a small amount of tweaking that we can do full-multi-backend matplotlib. No idea yet why one would want to, but I leave the possibility there in MainLoopBase. With all the backend-code specifics refactored out of FigureManager also aids in this, one manager to rule them (the backends) all.

Mdehoon: @OceanWolf, OK, thanks for the explanation. Having a uniform API for the backends is very important for the maintainability of matplotlib. I think this MEP is a step in the right direction.
Matplotlib only uses BSD compatible code. If you bring in code from another project make sure it has a PSF, BSD, MIT or compatible license (see the Open Source Initiative licenses page for details on individual licenses). If it doesn’t, you may consider contacting the author and asking them to relicense it. GPL and LGPL code are not acceptable in the main code base, though we are considering an alternative way of distributing L/GPL code through an separate channel, possibly a toolkit. If you include code, make sure you include a copy of that code’s license in the license directory if the code’s license requires you to distribute the license with it. Non-BSD compatible licenses are acceptable in matplotlib toolkits (e.g., basemap), but make sure you clearly state the licenses you are using.

86.1 Why BSD compatible?

The two dominant license variants in the wild are GPL-style and BSD-style. There are countless other licenses that place specific restrictions on code reuse, but there is an important difference to be considered in the GPL and BSD variants. The best known and perhaps most widely used license is the GPL, which in addition to granting you full rights to the source code including redistribution, carries with it an extra obligation. If you use GPL code in your own code, or link with it, your product must be released under a GPL compatible license. i.e., you are required to give the source code to other people and give them the right to redistribute it as well. Many of the most famous and widely used open source projects are released under the GPL, including linux, gcc, emacs and sage.

The second major class are the BSD-style licenses (which includes MIT and the python PSF license). These basically allow you to do whatever you want with the code: ignore it, include it in your own open source project, include it in your proprietary product, sell it, whatever. python itself is released under a BSD compatible license, in the sense that, quoting from the PSF license page:

> There is no GPL-like "copyleft" restriction. Distributing binary-only versions of Python, modified or not, is allowed. There is no requirement to release any of your source code. You can also write extension modules for Python and provide them only in binary form.

Famous projects released under a BSD-style license in the permissive sense of the last paragraph are the BSD operating system, python and TeX.

There are several reasons why early matplotlib developers selected a BSD compatible license. matplotlib is a python extension, and we choose a license that was based on the python license (BSD compatible).
Also, we wanted to attract as many users and developers as possible, and many software companies will not use GPL code in software they plan to distribute, even those that are highly committed to open source development, such as enthought, out of legitimate concern that use of the GPL will “infect” their code base by its viral nature. In effect, they want to retain the right to release some proprietary code. Companies and institutions who use matplotlib often make significant contributions, because they have the resources to get a job done, even a boring one. Two of the matplotlib backends (FLTK and WX) were contributed by private companies. The final reason behind the licensing choice is compatibility with the other python extensions for scientific computing: ipython, numpy, scipy, the enthought tool suite and python itself are all distributed under BSD compatible licenses.
As discussed at length elsewhere [insert links], jet is an empirically bad color map and should not be the default color map. Due to the position that changing the appearance of the plot breaks backward compatibility, this change has been put off for far longer than it should have been. In addition to changing the default color map we plan to take the chance to change the default color-cycle on plots and to adopt a different color map for filled plots (imshow, pcolor, contourf, etc) and for scatter like plots.

87.1 Default Heat Map Colormap

The choice of a new color map is fertile ground to bike-shedding ("No, it should be _this_ color") so we have a proposed set criteria (via Nathaniel Smith) to evaluate proposed color maps.

- it should be a sequential colormap, because diverging colormaps are really misleading unless you know where the “center” of the data is, and for a default colormap we generally won’t.
- it should be perceptually uniform, i.e., human subjective judgments of how far apart nearby colors are should correspond as linearly as possible to the difference between the numerical values they represent, at least locally.
- it should have a perceptually uniform luminance ramp, i.e. if you convert to greyscale it should still be uniform. This is useful both in practical terms (greyscale printers are still a thing!) and because luminance is a very strong and natural cue to magnitude.
- it should also have some kind of variation in hue, because hue variation is a really helpful additional cue to perception, having two cues is better than one, and there’s no reason not to do it.
- the hue variation should be chosen to produce reasonable results even for viewers with the more common types of colorblindness. (Which rules out things like red-to-green.)
- For bonus points, it would be nice to choose a hue ramp that still works if you throw away the luminance variation, because then we could use the version with varying luminance for 2d plots, and the version with just hue variation for 3d plots. (In 3d plots you really want to reserve the luminance channel for lighting/shading, because your brain is really good at extracting 3d shape from luminance variation. If the 3d surface itself has massively varying luminance then this screws up the ability to see shape.)
- Not infringe any existing IP
87.1.1 Example script

87.1.2 Proposed Colormaps

87.2 Default Scatter Colormap

For heat-map like applications it can be desirable to cover as much of the luminence scale as possible, however when color mapping markers, having markers too close to white can be a problem. For that reason we propose using a different (but maybe related) color map to the heat map for marker-based. The design parameters are the same as above, only with a more limited luminence variation.

87.2.1 Example script

```python
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(1234)

fig, (ax1, ax2) = plt.subplots(1, 2)

N = 50
x = np.random.rand(N)
y = np.random.rand(N)
colors = np.random.rand(N)
area = np.pi * (15 * np.random.rand(N))**2  # 0 to 15 point radiuses

ax1.scatter(x, y, s=area, c=colors, alpha=0.5)

X, Y = np.meshgrid(np.arange(0, 2*np.pi, .2),
                   np.arange(0, 2*np.pi, .2))
U = np.cos(X)
V = np.sin(Y)
Q = ax2.quiver(X, Y, U, V, units='width')
qd = np.random.rand(np.prod(X.shape))
Q.set_array(qd)
```

87.2.2 Proposed Colormaps

87.3 Color Cycle / Qualitative color map

When plotting lines it is frequently desirable to plot multiple lines or artists which need to be distinguishable, but there is no inherent ordering.
87.3.1 Example script

```python
import numpy as np
import matplotlib.pyplot as plt

fig, (ax1, ax2) = plt.subplots(1, 2)

x = np.linspace(0, 1, 10)

for j in range(10):
    ax1.plot(x, x * j)

th = np.linspace(0, 2*np.pi, 1024)
for j in np.linspace(0, np.pi, 10):
    ax2.plot(th, np.sin(th + j))

ax2.set_xlim(0, 2*np.pi)
```

87.3.2 Proposed Color cycle
Part IX

Matplotlib Examples
88.1 animation example code: moviewriter.py

```python
# This example uses a MovieWriter directly to grab individual frames and write them to a file. This avoids any event loop integration, but has the advantage of working with even the Agg backend. This is not recommended for use in an interactive setting.

# -*- noplot -*-
import numpy as np
import matplotlib
matplotlib.use("Agg")
import matplotlib.pyplot as plt
import matplotlib.animation as manimation
FFMpegWriter = manimation.writers['ffmpeg']
metadata = dict(title='Movie Test', artist='Matplotlib', comment='Movie support!')
writer = FFMpegWriter(fps=15, metadata=metadata)

fig = plt.figure()
l, = plt.plot([], [], 'k-o')
plt.xlim(-5, 5)
plt.ylim(-5, 5)

x0, y0 = 0, 0

with writer.saving(fig, "writer_test.mp4", 100):
    for i in range(100):
        x0 += 0.1 * np.random.randn()
```

```python
y0 += 0.1 * np.random.randn()
l.set_data(x0, y0)
writer.grab_frame()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

### 88.2 animation example code: subplots.py

[source code]

```python

This example uses subclassing, but there is no reason that the proper function
couldn't be set up and then use FuncAnimation. The code is long, but not
really complex. The length is due solely to the fact that there are a total of
9 lines that need to be changed for the animation as well as 3 subplots that
need initial set up.

```
```python
ax1.set_ylim(-2, 2)
ax1.set_aspect('equal', 'datalim')

ax2.set_xlabel('y')
ax2.set_ylabel('z')
self.line2 = Line2D([], [], color='black')
self.line2a = Line2D([], [], color='red', linewidth=2)
self.line2e = Line2D([], [], color='red', marker='o', markeredgecolor='r')
ax2.add_line(self.line2)
ax2.add_line(self.line2a)
ax2.add_line(self.line2e)
ax2.set_xlim(-1, 1)
ax2.set_ylim(0, 800)

ax3.set_xlabel('x')
ax3.set_ylabel('z')
self.line3 = Line2D([], [], color='black')
self.line3a = Line2D([], [], color='red', linewidth=2)
self.line3e = Line2D([], [], color='red', marker='o', markeredgecolor='r')
ax3.add_line(self.line3)
ax3.add_line(self.line3a)
ax3.add_line(self.line3e)
ax3.set_xlim(-1, 1)
ax3.set_ylim(0, 800)

animation.TimedAnimation.__init__(self, fig, interval=50, blit=True)

def _draw_frame(self, framedata):
    i = framedata
    head = i - 1
    head_slice = (self.t > self.t[i] - 1.0) & (self.t < self.t[i])

    self.line1.set_data(self.x[:i], self.y[:i])
    self.line1a.set_data(self.x[head_slice], self.y[head_slice])
    self.line1e.set_data(self.x[head], self.y[head])

    self.line2.set_data(self.y[:i], self.z[:i])
    self.line2a.set_data(self.y[head_slice], self.z[head_slice])
    self.line2e.set_data(self.y[head], self.z[head])

    self.line3.set_data(self.x[:i], self.z[:i])
    self.line3a.set_data(self.x[head_slice], self.z[head_slice])
    self.line3e.set_data(self.x[head], self.z[head])

    self._drawn_artists = [self.line1, self.line1a, self.line1e,
                           self.line2, self.line2a, self.line2e,
                           self.line3, self.line3a, self.line3e]

def new_frame_seq(self):
    return iter(range(self.t.size))
```

---

88.2. animation example code: subplots.py 2027
```python
def _init_draw(self):
    lines = [self.line1, self.line1a, self.line1e,
             self.line2, self.line2a, self.line2e,
             self.line3, self.line3a, self.line3e]
    for l in lines:
        l.set_data([], [])

ani = SubplotAnimation()
# ani.save('test_sub.mp4')
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
89.1 api example code: agg_oo.py

[source code]

```python
# -*- noplot -*-

"""
The object-oriented interface
"""

A pure OO (look Ma, no pylab!) example using the agg backend
""

from matplotlib.backends.backend_agg import FigureCanvasAgg as FigureCanvas
from matplotlib.figure import Figure

fig = Figure()
canvas = FigureCanvas(fig)
ax = fig.add_subplot(111)
ax.plot([1, 2, 3])
ax.set_title('hi mom')
ax.grid(True)
ax.set_xlabel('time')
ax.set_ylabel('volts')
canvas.print_figure('test')
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
89.2 api example code: barchart_demo.py

Scores by group and gender

---

Barchart
---

A bar plot with errorbars and height labels on individual bars

```python
import numpy as np
import matplotlib.pyplot as plt

N = 5
men_means = (20, 35, 30, 35, 27)
mens_std = (2, 3, 4, 1, 2)

ind = np.arange(N)  # the x locations for the groups
width = 0.35         # the width of the bars

fig, ax = plt.subplots()
rects1 = ax.bar(ind, men_means, width, color='r', yerr=mens_std)

women_means = (25, 32, 34, 20, 25)
women_std = (3, 5, 2, 3, 3)
```

Chapter 89. api Examples
rects2 = ax.bar(ind + width, women_means, width, color='y', yerr=women_std)

# add some text for labels, title and axes ticks
ax.set_ylabel('Scores')
ax.set_title('Scores by group and gender')
ax.set_xticks(ind + width / 2)
ax.set_xticklabels(('G1', 'G2', 'G3', 'G4', 'G5'))

ax.legend((rects1[0], rects2[0]), ('Men', 'Women'))

def autolabel(rects):
    """
    Attach a text label above each bar displaying its height
    """
    for rect in rects:
        height = rect.get_height()
        ax.text(rect.get_x() + rect.get_width()/2., 1.05*height,
                '%d' % int(height),
                ha='center', va='bottom')

autolabel(rects1)
autolabel(rects2)

plt.show()
Matplotlib, Release 2.0.2

89.3 api example code: bbox_intersect.py

3
2
1
0
1
2
3
3

2

1

0

1

2

3

"""
===========================================
Changing colors of lines intersecting a box
===========================================
The lines intersecting the rectangle are colored in red, while the others
are left as blue lines. This example showcases the `intersect_bbox` function.
"""
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.transforms import Bbox
from matplotlib.path import Path
left, bottom, width, height = (-1, -1, 2, 2)
rect = plt.Rectangle((left, bottom), width, height, facecolor="#aaaaaa")
fig, ax = plt.subplots()
ax.add_patch(rect)
bbox = Bbox.from_bounds(left, bottom, width, height)

2032

Chapter 89. api Examples


for i in range(12):
    vertices = (np.random.random((2, 2)) - 0.5) * 6.0
    path = Path(vertices)
    if path.intersects_bbox(bbox):
        color = 'r'
    else:
        color = 'b'
    ax.plot(vertices[:, 0], vertices[:, 1], color=color)

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

89.4 api example code: collections_demo.py

```python
for i in range(12):
    vertices = (np.random.random((2, 2)) - 0.5) * 6.0
    path = Path(vertices)
    if path.intersects_bbox(bbox):
        color = 'r'
    else:
        color = 'b'
    ax.plot(vertices[:, 0], vertices[:, 1], color=color)

plt.show()
```

For the first two subplots, we will use spirals. Their
size will be set in plot units, not data units. Their positions will be set in data units by using the "offsets" and "transOffset" kwargs of the LineCollection and PolyCollection.

The third subplot will make regular polygons, with the same type of scaling and positioning as in the first two.

The last subplot illustrates the use of "offsets=(xo,yo)", that is, a single tuple instead of a list of tuples, to generate successively offset curves, with the offset given in data units. This behavior is available only for the LineCollection.

```
import matplotlib.pyplot as plt
from matplotlib import collections, colors, transforms
import numpy as np

nverts = 50
npts = 100

# Make some spirals
r = np.arange(nverts)
theta = np.linspace(0, 2*np.pi, nverts)
xx = r * np.sin(theta)
yy = r * np.cos(theta)
spiral = list(zip(xx, yy))

# Make some offsets
rs = np.random.RandomState([12345678])
oxo = rs.randn(npts)
yo = rs.randn(npts)
xyo = list(zip(xo, yo))

# Make a list of colors cycling through the default series.
colors = [colors.to_rgba(c)
    for c in plt.rcParams['axes.prop_cycle'].by_key()['color']]

fig, axes = plt.subplots(2, 2)
fig.subplots_adjust(top=0.92, left=0.07, right=0.97,
    hspace=0.3, wspace=0.3)
((ax1, ax2), (ax3, ax4)) = axes
# unpack the axes

col = collections.LineCollection([spiral], offsets=xyo,
    transOffset=ax1.transData)
trans = fig.dpi_scale_trans + transforms.Affine2D().scale(1.0/72.0)
col.set_transform(trans)  # the points to pixels transform
# Note: the first argument to the collection initializer
# must be a list of sequences of x,y tuples; we have only
# one sequence, but we still have to put it in a list.
ax1.add_collection(col, autolim=True)
# autolim=True enables autoscaling. For collections with
# offsets like this, it is neither efficient nor accurate,
# but it is good enough to generate a plot that you can use
# as a starting point. If you know beforehand the range of
# x and y that you want to show, it is better to set them
# explicitly, leave out the autolim kwarg (or set it to False),
# and omit the `ax1.autoscale_view()` call below.

# Make a transform for the line segments such that their size is
# given in points:
col.set_color(colors)

ax1.autoscale_view()  # See comment above, after ax1.add_collection.
ax1.set_title('LineCollection using offsets')

# The same data as above, but fill the curves.
col = collections.PolyCollection([spiral], offsets=xyo,
                                    transOffset=ax2.transData)
trans = transforms.Affine2D().scale(fig.dpi/72.0)
col.set_transform(trans)  # the points to pixels transform
ax2.add_collection(col, autolim=True)
col.set_color(colors)

ax2.autoscale_view()
ax2.set_title('PolyCollection using offsets')

# 7-sided regular polygons

col = collections.RegularPolyCollection(7,
                                        sizes=np.fabs(xx) * 10.0, offsets=xyo,
                                        transOffset=ax3.transData)
trans = transforms.Affine2D().scale(fig.dpi / 72.0)
col.set_transform(trans)  # the points to pixels transform
ax3.add_collection(col, autolim=True)
col.set_color(colors)
ax3.autoscale_view()
ax3.set_title('RegularPolyCollection using offsets')

# Simulate a series of ocean current profiles, successively
# offset by 0.1 m/s so that they form what is sometimes called
# a "waterfall" plot or a "stagger" plot.

nverts = 60
ncurves = 20
offs = (0.1, 0.0)

yy = np.linspace(0, 2*np.pi, nverts)
ym = np.amax(yy)
xx = (0.2 + (ym - yy)/ym)**2 * np.cos(yy - 0.4)*0.5
segs = []
for i in range(ncurves):
    segs.append([xx, yy])
    yy += offs

ax4 = fig.add_subplot(224)
ax4.set_title('Stagger plot')
ax4.plot(segs, label='current profiles')
ax4.legend()
```python
xxx = xx + 0.02*rs.randn(nverts)
curve = list(zip(xxx, yy*100))
segs.append(curve)

col = collections.LineCollection(segs, offsets=offs)
ax4.add_collection(col, autolim=True)
col.set_color(colors)
ax4.autoscale_view()
ax4.set_title('Successive data offsets')
ax4.set_xlabel('Zonal velocity component (m/s)')
ax4.set_ylabel('Depth (m)')
# Reverse the y-axis so depth increases downward
ax4.set_ylim(ax4.get_ylim()[::-1])

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

### 89.5 api example code: colorbar_basics.py

```python
# This example shows how to use colorbar by specifying the mappable object (here the imshow returned object) and the axes to attach the colorbar to.

import numpy as np
import matplotlib.pyplot as plt

# setup some generic data
N = 37
```
x, y = np.mgrid[:N, :N]
Z = (np.cos(x*0.2) + np.sin(y*0.3))

# mask out the negative and positive values, respectively
Zpos = np.ma.masked_less(Z, 0)
Zneg = np.ma.masked_greater(Z, 0)

fig, (ax1, ax2) = plt.subplots(figsize=(8, 3), ncols=2)

# plot just the positive data and save the
# color "mappable" object returned by ax1.imshow
pos = ax1.imshow(Zpos, cmap='Blues', interpolation='none')

# add the colorbar using the figure's method,
# telling which mappable we're talking about and
# which axes object it should be near
fig.colorbar(pos, ax=ax1)

# repeat everything above for the the negative data
neg = ax2.imshow(Zneg, cmap='Reds_r', interpolation='none')
fig.colorbar(neg, ax=ax2)

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
import matplotlib.pyplot as plt
import matplotlib as mpl

# Make a figure and axes with dimensions as desired.
fig = plt.figure(figsize=(8, 3))
ax1 = fig.add_axes([0.05, 0.80, 0.9, 0.15])
ax2 = fig.add_axes([0.05, 0.475, 0.9, 0.15])
ax3 = fig.add_axes([0.05, 0.15, 0.9, 0.15])

# Set the colormap and norm to correspond to the data for which
# the colorbar will be used.
cmap = mpl.cm.cool
norm = mpl.colors.Normalize(vmin=5, vmax=10)

# ColorbarBase derives from ScalarMappable and puts a colorbar
# in a specified axes, so it has everything needed for a
# standalone colorbar. There are many more kwargs, but the
# following gives a basic continuous colorbar with ticks
# and labels.
    cb1 = mpl.colorbar.ColorbarBase(ax1, cmap=cmap,
        norm=norm,
        orientation='horizontal')
    cb1.set_label('Some Units')

# The second example illustrates the use of a ListedColormap, a
# BoundaryNorm, and extended ends to show the "over" and "under"
# value colors.
cmap = mpl.colors.ListedColormap(['r', 'g', 'b', 'c'])
cmap.set_over('0.25')
cmap.set_under('0.75')

# If a ListedColormap is used, the length of the bounds array must be
# one greater than the length of the color list. The bounds must be
# monotonically increasing.
bounds = [1, 2, 4, 7, 8]
norm = mpl.colors.BoundaryNorm(bounds, cmap.N)
    cb2 = mpl.colorbar.ColorbarBase(ax2, cmap=cmap,
        norm=norm,
        # to use 'extend', you must
        # specify two extra boundaries:
        boundaries=[0] + bounds + [13],
        extend='both',
        ticks=bounds,  # optional
        spacing='proportional',
        orientation='horizontal')
    cb2.set_label('Discrete intervals, some other units')

# The third example illustrates the use of custom length colorbar
# extensions, used on a colorbar with discrete intervals.
cmap = mpl.colors.ListedColormap([[0., .4, 1.], [0., .8, 1.],
    [1., .8, 0.], [1., .4, 0.]])
cmap.set_over((1., 0., 0.))
cmap.set_under((0., 0., 1.))

bounds = [-1., -.5, 0., .5, 1.]
norm = mpl.colors.BoundaryNorm(bounds, cmap.N)
cb3 = mpl.colorbar.ColorbarBase(ax3, cmap=cmap,
    norm=norm,
    boundaries=[-10] + bounds + [10],
    extend='both',
    # Make the length of each extension
    # the same as the length of the
    # interior colors:
    extendfrac='auto',
    ticks=bounds,
    spacing='uniform',
    orientation='horizontal')
cb3.set_label('Custom extension lengths, some other units')

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

89.7 api example code: compound_path.py
Make a compound path -- in this case two simple polygons, a rectangle and a triangle. Use CLOSEPOLY and MOVETO for the different parts of the compound path

```python
import numpy as np
from matplotlib.path import Path
from matplotlib.patches import PathPatch
import matplotlib.pyplot as plt

vertices = []
codes = []

codes = [Path.MOVETO] + [Path.LINETO]*3 + [Path.CLOSEPOLY]
vertices = [(1, 1), (1, 2), (2, 2), (2, 1), (0, 0)]

codes += [Path.MOVETO] + [Path.LINETO]*2 + [Path.CLOSEPOLY]
vertices += [(4, 4), (5, 5), (5, 4), (0, 0)]

vertices = np.array(vertices, float)
path = Path(vertices, codes)

pathpatch = PathPatch(path, facecolor='None', edgecolor='green')

fig, ax = plt.subplots()
ax.add_patch(pathpatch)
ax.set_title('A compound path')

ax.dataLim.update_from_data_xy(vertices)
ax.autoscale_view()

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
This example showcases the Hammer projection by alleviating many features of matplotlib.

```python
from __future__ import unicode_literals
import matplotlib
from matplotlib.axes import Axes
from matplotlib.patches import Circle
from matplotlib.path import Path
from matplotlib.ticker import NullLocator, Formatter, FixedLocator
from matplotlib.transforms import Affine2D, BboxTransformTo, Transform
from matplotlib.projections import register_projection
import matplotlib.spines as mspines
import matplotlib.axis as maxis
import numpy as np
```
rcParams = matplotlib.rcParams

# This example projection class is rather long, but it is designed to
# illustrate many features, not all of which will be used every time.
# It is also common to factor out a lot of these methods into common
# code used by a number of projections with similar characteristics
# (see geo.py).

class GeoAxes(Axes):
    """
    An abstract base class for geographic projections
    """

class ThetaFormatter(Formatter):
    """
    Used to format the theta tick labels. Converts the native
    unit of radians into degrees and adds a degree symbol.
    """
    def __init__(self, round_to=1.0):
        self._round_to = round_to

    def __call__(self, x, pos=None):
        degrees = (x / np.pi) * 180.0
        degrees = np.round(degrees / self._round_to) * self._round_to
        if rcParams['text.usetex'] and not rcParams['text.latex.unicode']:
            return r"%0.0f\degree" % degrees
        else:
            return "%0.0f\degree" % degrees

RESOLUTION = 75

def _init_axis(self):
    self.xaxis = maxis.XAxis(self)
    self.yaxis = maxis.YAxis(self)
    # Do not register xaxis or yaxis with spines -- as done in
    # Axes._init_axis() -- until GeoAxes.xaxis.cla() works.
    # self.spines['geo'].register_axis(self.yaxis)
    self._update_transScale()

def cla(self):
    Axes.cla(self)

    self.set_longitude_grid(30)
    self.set_latitude_grid(15)
    self.set_longitude_grid_ends(75)
    self.xaxis.set_minor_locator(NullLocator())
    self.yaxis.set_minor_locator(NullLocator())
    self.xaxis.set_ticks_position('none')
    self.yaxis.set_ticks_position('none')
    self.yaxis.set_tick_params(label1On=True)
    # Why do we need to turn on yaxis tick labels, but
    # xaxis tick labels are already on?
self.grid(rcParams['axes.grid'])

Axes.set_xlim(self, -np.pi, np.pi)
Axes.set_ylim(self, -np.pi / 2.0, np.pi / 2.0)

def _set_lim_and_transforms(self):
    # A (possibly non-linear) projection on the (already scaled) data

    # There are three important coordinate spaces going on here:
    #
    # 1. Data space: The space of the data itself
    #
    # 2. Axes space: The unit rectangle (0, 0) to (1, 1)
    #    covering the entire plot area.
    #
    # 3. Display space: The coordinates of the resulting image,
    #    often in pixels or dpi/inch.

    # This function makes heavy use of the Transform classes in
    # `lib/matplotlib/transforms.py`. For more information, see
    # the inline documentation there.

    # The goal of the first two transformations is to get from the
    # data space (in this case longitude and latitude) to axes
    # space. It is separated into a non-affine and affine part so
    # that the non-affine part does not have to be recomputed when
    # a simple affine change to the figure has been made (such as
    # resizing the window or changing the dpi).

    # 1) The core transformation from data space into
    # rectilinear space defined in the HammerTransform class.
    self.transProjection = self._get_core_transform(self.RESOLUTION)

    # 2) The above has an output range that is not in the unit
    # rectangle, so scale and translate it so it fits correctly
    # within the axes. The peculiar calculations of xscale and
    # yscale are specific to a Aitoff-Hammer projection, so don't
    # worry about them too much.
    self.transAffine = self._get_affine_transform()

    # 3) This is the transformation from axes space to display
    # space.
    self.transAxes = BboxTransformTo(self.bbox)

    # Now put these 3 transforms together -- from data all the way
    # to display coordinates. Using the '+' operator, these
    # transforms will be applied "in order". The transforms are
    # automatically simplified, if possible, by the underlying
    # transformation framework.
    self.transData =
        self.transProjection +
        self.transAffine +

self.transAxes

# The main data transformation is set up. Now deal with
# gridlines and tick labels.

# Longitude gridlines and ticklabels. The input to these
# transforms are in display space in x and axes space in y.
# Therefore, the input values will be in range (-xmin, 0),
# (xmax, 1). The goal of these transforms is to go from that
# space to display space. The tick labels will be offset 4
# pixels from the equator.
self._xaxis_pretransform =
    Affine2D() 
    .scale(1.0, self._longitude_cap * 2.0) 
    .translate(0.0, -self._longitude_cap)
self._xaxis_transform =
    self._xaxis_pretransform +
    self.transData
self._xaxis_text1_transform =
    Affine2D().scale(1.0, 0.0) +
    self.transData +
    Affine2D().translate(0.0, 4.0)
self._xaxis_text2_transform =
    Affine2D().scale(1.0, 0.0) +
    self.transData +
    Affine2D().translate(0.0, -4.0)

# Now set up the transforms for the latitude ticks. The input to
# these transforms are in axes space in x and display space in
# y. Therefore, the input values will be in range (0, -ymin),
# (1, ymax). The goal of these transforms is to go from that
# space to display space. The tick labels will be offset 4
# pixels from the edge of the axes ellipse.
yaxis_stretch = Affine2D().scale(np.pi*2, 1).translate(-np.pi, 0)
yaxis_space = Affine2D().scale(1.0, 1.1)
self._yaxis_transform =
    yaxis_stretch +
    self.transData
yaxis_text_base =
    yaxis_stretch +
    self.transProjection +
    (yaxis_space +
     self.transAffine +
     self.transAxes)
self._yaxis_text1_transform =
    yaxis_text_base +
    Affine2D().translate(-8.0, 0.0)
self._yaxis_text2_transform =
    yaxis_text_base +
    Affine2D().translate(8.0, 0.0)

def _get_affine_transform(self):
    transform = self._get_core_transform(1)
```python
xscale, _ = transform.transform_point((np.pi, 0))
_, yscale = transform.transform_point((0, np.pi / 2.0))
return Affine2D() \
   .scale(0.5 / xscale, 0.5 / yscale) \
   .translate(0.5, 0.5)

def get_xaxis_transform(self, which='grid'):
    """
    Override this method to provide a transformation for the
    x-axis tick labels.
    
    Returns a tuple of the form (transform, valign, halign)
    """
    if which not in ['tick1', 'tick2', 'grid']:
        msg = "'which' must be one of [ 'tick1' | 'tick2' | 'grid' ]"
        raise ValueError(msg)
    return self._xaxis_transform

def get_xaxis_text1_transform(self, pad):
    return self._xaxis_text1_transform, 'bottom', 'center'

def get_xaxis_text2_transform(self, pad):
    """
    Override this method to provide a transformation for the
    secondary x-axis tick labels.
    
    Returns a tuple of the form (transform, valign, halign)
    """
    return self._xaxis_text2_transform, 'top', 'center'

def get_yaxis_transform(self, which='grid'):
    """
    Override this method to provide a transformation for the
    y-axis grid and ticks.
    """
    if which not in ['tick1', 'tick2', 'grid']:
        msg = "'which' must be one of [ 'tick1' | 'tick2' | 'grid' ]"
        raise ValueError(msg)
    return self._yaxis_transform

def get_yaxis_text1_transform(self, pad):
    """
    Override this method to provide a transformation for the
    y-axis tick labels.
    
    Returns a tuple of the form (transform, valign, halign)
    """
    return self._yaxis_text1_transform, 'center', 'right'

def get_yaxis_text2_transform(self, pad):
    """
    Override this method to provide a transformation for the
    secondary y-axis tick labels.
```

Returns a tuple of the form (transform, valign, halign)

    
return self._yaxis_text2_transform, 'center', 'left'

def _gen_axes_patch(self):
    
    Override this method to define the shape that is used for the
    background of the plot. It should be a subclass of Patch.
    
    In this case, it is a Circle (that may be warped by the axes
    transform into an ellipse). Any data and gridlines will be
    clipped to this shape.

    
return Circle((0.5, 0.5), 0.5)

def _gen_axes_spines(self):
    return {'geo': mspsines.Spine.circular_spine(self, (0.5, 0.5), 0.5)}

def set_yscale(self, *args, **kwargs):
    if args[0] != 'linear':
        raise NotImplementedError

    set_xscale = set_yscale

    # Prevent the user from changing the axes limits. In our case, we
    # want to display the whole sphere all the time, so we override
    # set_xlim and set_ylim to ignore any input. This also applies to
    # interactive panning and zooming in the GUI interfaces.
    def set_xlim(self, *args, **kwargs):
        raise TypeError("It is not possible to change axes limits "
                        "for geographic projections. Please consider "
                        "using Basemap or Cartopy."

    set_ylim = set_xlim

def format_coord(self, lon, lat):
    
    Override this method to change how the values are displayed in
    the status bar.
    
    In this case, we want them to be displayed in degrees N/S/E/W.

    
lon = lon * (180.0 / np.pi)
lat = lat * (180.0 / np.pi)
if lat >= 0.0:
    ns = 'N'
else:
    ns = 'S'
if lon >= 0.0:
ew = 'E'
else:
    ew = 'W'
return '\%f\u00b0\u00b0\%s, \%f\u00b0\u00b0\%s' % (abs(lat), ns, abs(lon), ew)

def set_longitude_grid(self, degrees):
    
    Set the number of degrees between each longitude grid.
    
    This is an example method that is specific to this projection class -- it provides a more convenient interface to set the ticking than set_xticks would.
    
    number = (360.0 / degrees) + 1
    self.xaxis.set_major_locator(
        FixedLocator(
            np.linspace(-np.pi, np.pi, number, True)[1:-1]))
    self.xaxis.set_major_formatter(self.ThetaFormatter(degrees))

def set_latitude_grid(self, degrees):
    
    Set the number of degrees between each longitude grid.
    
    This is an example method that is specific to this projection class -- it provides a more convenient interface than set_yticks would.
    
    number = (180.0 / degrees) + 1
    self.yaxis.set_major_locator(
        FixedLocator(
            np.linspace(-np.pi / 2.0, np.pi / 2.0, number, True)[1:-1]))
    self.yaxis.set_major_formatter(self.ThetaFormatter(degrees))

def set_longitude_grid_ends(self, degrees):
    
    Set the latitude(s) at which to stop drawing the longitude grids.
    
    Often, in geographic projections, you wouldn't want to draw longitude gridlines near the poles. This allows the user to specify the degree at which to stop drawing longitude grids.
    
    This is an example method that is specific to this projection class -- it provides an interface to something that has no analogy in the base Axes class.
    
    self._longitude_cap = degrees * (np.pi / 180.0)
    self._xaxis_pretransform =
        .clear() \
        .scale(1.0, self._longitude_cap * 2.0) \
        .translate(0.0, -self._longitude_cap)

def get_data_ratio(self):
    
    89.8. api example code: custom_projection_example.py 2047
Return the aspect ratio of the data itself.

This method should be overridden by any Axes that have a fixed data ratio.

```
return 1.0
```

# Interactive panning and zooming is not supported with this projection,
# so we override all of the following methods to disable it.

def can_zoom(self):
    """
    Return "True" if this axes supports the zoom box button functionality.
    This axes object does not support interactive zoom box.
    """
    return False

def can_pan(self):
    """
    Return "True" if this axes supports the pan/zoom button functionality.
    This axes object does not support interactive pan/zoom.
    """
    return False

def start_pan(self, x, y, button):
    pass

def end_pan(self):
    pass

def drag_pan(self, button, key, x, y):
    pass

class HammerAxes(GeoAxes):
    ""
    A custom class for the Aitoff-Hammer projection, an equal-area map projection.

    ""

    # The projection must specify a name. This will be used by the user to select the projection,
    # i.e. `subplot(111, projection='custom_hammer')`
    name = 'custom_hammer'

class HammerTransform(Transform):
    ""
    The base Hammer transform.
    ""
    input_dims = 2
    output_dims = 2
    is_separable = False
def __init__(self, resolution):
    ""
    Create a new Hammer transform. Resolution is the number of steps
to interpolate between each input line segment to approximate its
path in curved Hammer space.
    ""
    Transform.__init__(self)
    self._resolution = resolution
def transform_non_affine(self, ll):
    longitude = ll[:, 0:1]
    latitude = ll[:, 1:2]

    # Pre-compute some values
    half_long = longitude / 2.0
    cos_latitude = np.cos(latitude)
    sqrt2 = np.sqrt(2.0)
    alpha = np.sqrt(1.0 + cos_latitude * np.cos(half_long))
    x = (2.0 * sqrt2) * (cos_latitude * np.sin(half_long)) / alpha
    y = (sqrt2 * np.sin(latitude)) / alpha
    return np.concatenate((x, y), 1)
transform_non_affine.__doc__ = Transform.transform_non_affine.__doc__
def transform_path_non_affine(self, path):
    # vertices = path.vertices
    ipath = path.interpolated(self._resolution)
    return Path(self.transform(ipath.vertices), ipath.codes)
transform_path_non_affine.__doc__ = 
    Transform.transform_path_non_affine.__doc__
def inverted(self):
    return HammerAxes.InvertedHammerTransform(self._resolution)
inverted.__doc__ = Transform.inverted.__doc__

class InvertedHammerTransform(Transform):
    input_dims = 2
    output_dims = 2
    is_separable = False
def __init__(self, resolution):
    Transform.__init__(self)
    self._resolution = resolution
def transform_non_affine(self, xy):
    x = xy[:, 0:1]
    y = xy[:, 1:2]

    quarter_x = 0.25 * x
    half_y = 0.5 * y
    z = np.sqrt(1.0 - quarter_x*quarter_x - half_y*half_y)
    longitude = 2 * np.arctan((z*x) / (2.0 * (2.0*z**2 - 1.0)))
    return np.concatenate((x, y), 1)
latitude = np.arcsin(y*z)
    return np.concatenate((longitude, latitude), 1)
transform_non_affine.__doc__ = Transform.transform_non_affine.__doc__

def inverted(self):
    return HammerAxes.HammerTransform(self._resolution)
inverted.__doc__ = Transform.inverted.__doc__

def __init__(self, *args, **kwargs):
    self._longitude_cap = np.pi / 2.0
    GeoAxes.__init__(self, *args, **kwargs)
    self.set_aspect(0.5, adjustable='box', anchor='C')
    self.cla()

    def _get_core_transform(self, resolution):
        return self.HammerTransform(resolution)

# Now register the projection with matplotlib so the user can select
# it.
register_projection(HammerAxes)

if __name__ == '__main__':
    import matplotlib.pyplot as plt
    # Now make a simple example using the custom projection.
    plt.subplot(111, projection="custom_hammer")
    p = plt.plot([-1, 1, 1], [-1, -1, 1], "o-")
    plt.grid(True)

    plt.show()
89.9 api example code: custom_scale_example.py

```python
from __future__ import unicode_literals
import numpy as np
from numpy import ma
from matplotlib import scale as mscale
from matplotlib import transforms as mtransforms
from matplotlib.ticker import Formatter, FixedLocator
from matplotlib import rcParams

# BUG: this example fails with any other setting of axisbelow
rcParams['axes.axisbelow'] = False
```

This example showcases how to create a custom scale, by implementing the scaling use for latitude data in a Mercator Projection.
class MercatorLatitudeScale(mscale.ScaleBase):
    
    Scales data in range -\pi/2 to \pi/2 (-90 to 90 degrees) using the system used to scale latitudes in a Mercator projection.

    The scale function:
    \[ \ln(\tan(y) + \sec(y)) \]

    The inverse scale function:
    \[ \operatorname{atan}(\sinh(y)) \]

    Since the Mercator scale tends to infinity at +/- 90 degrees, there is user-defined threshold, above and below which nothing will be plotted. This defaults to +/- 85 degrees.

    source: http://en.wikipedia.org/wiki/Mercator_projection

    # The scale class must have a member `name` that defines the string used to select the scale. For example, `gca().set_yscale("mercator")` would be used to select this scale.
    name = 'mercator'

    def __init__(self, axis, **kwargs):
        
        Any keyword arguments passed to `set_xscale` and `set_yscale` will be passed along to the scale's constructor.

        thresh: The degree above which to crop the data.

        mscale.ScaleBase.__init__(self)
        thresh = kwargs.pop("thresh", np.radians(85))
        if thresh >= np.pi / 2.0:
            raise ValueError("thresh must be less than pi/2")
        self.thresh = thresh

    def get_transform(self):
        
        Override this method to return a new instance that does the actual transformation of the data.

        The MercatorLatitudeTransform class is defined below as a nested class of this one.

        return self.MercatorLatitudeTransform(self.thresh)

    def set_default_locators_and_formatters(self, axis):
Override to set up the locators and formatters to use with the scale. This is only required if the scale requires custom locators and formatters. Writing custom locators and formatters is rather outside the scope of this example, but there are many helpful examples in ``ticker.py``.

In our case, the Mercator example uses a fixed locator from -90 to 90 degrees and a custom formatter class to put convert the radians to degrees and put a degree symbol after the value:

```python
class DegreeFormatter(Formatter):
    def __call__(self, x, pos=None):
        # ° : degree symbol
        return '%d\u00b0' % (np.degrees(x))
```

```python
axis.set_major_locator(FixedLocator(np.radians(np.arange(-90, 90, 10))))
axis.set_major_formatter(DegreeFormatter())
axis.set_minor_formatter(DegreeFormatter())
```

```python
def limit_range_for_scale(self, vmin, vmax, minpos):
    return max(vmin, -self.thresh), min(vmax, self.thresh)
```

```python
class MercatorLatitudeTransform(mtransforms.Transform):
    # There are two value members that must be defined.
    # ``input_dims`` and ``output_dims`` specify number of input
    # dimensions and output dimensions to the transformation.
    # These are used by the transformation framework to do some
    # error checking and prevent incompatible transformations from
    # being connected together. When defining transforms for a
    # scale, which are, by definition, separable and have only one
    # dimension, these members should always be set to 1.
    input_dims = 1
    output_dims = 1
    is_separable = True

    def __init__(self, thresh):
        mtransforms.Transform.__init__(self)
        self.thresh = thresh

    def transform_non_affine(self, a):
        # This transform takes an N x 1 ``numpy`` array and returns a
transformed copy. Since the range of the Mercator scale is limited by the user-specified threshold, the input array must be masked to contain only valid values. \`matplotlib\` will handle masked arrays and remove the out-of-range data from the plot. Importantly, the \`transform\` method *must* return an array that is the same shape as the input array, since these values need to remain synchronized with values in the other dimension.

```python
masked = ma.masked_where((a < -self.thresh) | (a > self.thresh), a)
if masked.mask.any():
    return ma.log(np.abs(ma.tan(masked) + 1.0 / ma.cos(masked)))
else:
    return np.log(np.abs(np.tan(a) + 1.0 / np.cos(a)))
```

def inverted(self):
    """
    Override this method so matplotlib knows how to get the inverse transform for this transform.
    """
    return MercatorLatitudeScale.InvertedMercatorLatitudeTransform(self.thresh)

class InvertedMercatorLatitudeTransform(mtransforms.Transform):
    input_dims = 1
    output_dims = 1
    is_separable = True

    def __init__(self, thresh):
        mtransforms.Transform.__init__(self)
        self.thresh = thresh

    def transform_non_affine(self, a):
        return np.arctan(np.sinh(a))

    def inverted(self):
        return MercatorLatitudeScale.MercatorLatitudeTransform(self.thresh)

# Now that the Scale class has been defined, it must be registered so
# that \`matplotlib\` can find it.
mscale.register_scale(MercatorLatitudeScale)

if __name__ == '__main__':
    import matplotlib.pyplot as plt

    t = np.arange(-180.0, 180.0, 0.1)
    s = np.radians(t)/2.

    plt.plot(t, s, '-', lw=2)
    plt.gca().set_yscale('mercator')
    plt.xlabel('Longitude')
```python
plt.ylabel('Latitude')
plt.title('Mercator: Projection of the Oppressor')
plt.grid(True)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

### 89.10 api example code: date_demo.py

""
{}
--------------
Date tick labels
--------------

Show how to make date plots in matplotlib using date tick locators and formatters. See major_minor_demo1.py for more information on controlling major and minor ticks

All matplotlib date plotting is done by converting date instances into days since the 0001-01-01 UTC. The conversion, tick locating and formatting is done behind the scenes so this is most transparent to
you. The dates module provides several converter functions date2num
and num2date

```python
import datetime
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
import matplotlib.cbook as cbook

years = mdates.YearLocator()  # every year
months = mdates.MonthLocator()  # every month
yearsFmt = mdates.DateFormatter('%Y')

# load a numpy record array from yahoo csv data with fields date, open, close, volume, adj_close from the mpl-data/example directory.
# The record array stores python datetime.date as an object array in the date column
datafile = cbook.get_sample_data('goog.npy')

try:
    # Python3 cannot load python2 .npy files with datetime(object) arrays
    # unless the encoding is set to bytes. However this option was not added until numpy 1.10 so this example will only work with python 2 or with numpy 1.10 and later.
    r = np.load(datafile, encoding='bytes').view(np.recarray)
except TypeError:
    r = np.load(datafile).view(np.recarray)

fig, ax = plt.subplots()
ax.plot(r.date, r.adj_close)

# format the ticks
ax.xaxis.set_major_locator(years)
ax.xaxis.set_major_formatter(yearsFmt)
ax.xaxis.set_minor_locator(months)

datemin = datetime.date(r.date.min().year, 1, 1)
datemax = datetime.date(r.date.max().year + 1, 1, 1)
ax.set_xlim(datemin, datemax)

# rotates and right aligns the x labels, and moves the bottom of the axes up to make room for them
fig.autofmt_xdate()
```

```python
def price(x):
    return '${1.2f}' % x
ax.format_xdata = mdates.DateFormatter('%Y-%m-%d')
ax.format_ydata = price
ax.grid(True)
```
from __future__ import print_function
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.mlab as mlab
import matplotlib.cbook as cbook
import matplotlib.ticker as ticker

datafile = cbook.get_sample_data('aapl.csv', asfileobj=False)
print(' loading %s' % datafile)
r = mlab.csv2rec(datafile)
r.sort()
r = r[-30:]  # get the last 30 days
# first we'll do it the default way, with gaps on weekends
fig, axes = plt.subplots(ncols=2, figsize=(8, 4))
ax = axes[0]
ax.plot(r.date, r.adj_close, 'o-')
ax.set_title("Default")
fig.autofmt_xdate()

# next we'll write a custom formatter
N = len(r)
ind = np.arange(N)  # the evenly spaced plot indices

def format_date(x, pos=None):
    thisind = np.clip(int(x + 0.5), 0, N - 1)
    return r.date[thisind].strftime('%Y-%m-%d')

ax = axes[1]
ax.plot(ind, r.adj_close, 'o-')
ax.xaxis.set_major_formatter(ticker.FuncFormatter(format_date))
ax.set_title("Custom tick formatter")
fig.autofmt_xdate()

plt.show()
Affine transform of an image

For the backends that support draw_image with optional affine transform (e.g., agg, ps backend), the image of the output should have its boundary match the dashed yellow rectangle.

```python
import numpy as np
import matplotlib.mlab as mlab
import matplotlib.pyplot as plt
import matplotlib.transforms as mtransforms

def get_image():
    delta = 0.25
    x = y = np.arange(-3.0, 3.0, delta)
    X, Y = np.meshgrid(x, y)
    Z1 = mlab.bivariate_normal(X, Y, 1.0, 1.0, 0.0, 0.0)
    Z2 = mlab.bivariate_normal(X, Y, 1.5, 0.5, 1, 1)
```
Z = Z2 - Z1  # difference of Gaussians
return Z

def do_plot(ax, Z, transform):
    im = ax.imshow(Z, interpolation='none',
                   origin='lower',
                   extent=[-2, 4, -3, 2], clip_on=True)

    trans_data = transform + ax.transData
    im.set_transform(trans_data)

    # display intended extent of the image
    x1, x2, y1, y2 = im.get_extent()
    ax.plot([x1, x2, x1, x2, x1], [y1, y1, y2, y2, y1], 'y--',
            transform=trans_data)
    ax.set_xlim(-5, 5)
    ax.set_ylim(-4, 4)

    # prepare image and figure
    fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2)
    Z = get_image()

    # image rotation
    do_plot(ax1, Z, mtransforms.Affine2D().rotate_deg(30))

    # image skew
    do_plot(ax2, Z, mtransforms.Affine2D().skew_deg(30, 15))

    # scale and reflection
    do_plot(ax3, Z, mtransforms.Affine2D().scale(-1, .5))

    # everything and a translation
    do_plot(ax4, Z, mtransforms.Affine2D().
            rotate_deg(30).skew_deg(30, 15).scale(-1, .5).translate(.5, -1))

plt.show()
This example draws donuts (miam!) using Path and Patches.

```python
import numpy as np
import matplotlib.path as mpath
import matplotlib.patches as mpatches
import matplotlib.pyplot as plt

def wise(v):
    if v == 1:
        return "CCW"
    else:
        return "CW"

def make_circle(r):
```

```python
t = np.arange(0, np.pi * 2.0, 0.01)
t = t.reshape((len(t), 1))
x = r * np.cos(t)
y = r * np.sin(t)
return np.hstack((x, y))

Path = mpath.Path

fig, ax = plt.subplots()
inside_vertices = make_circle(0.5)
outside_vertices = make_circle(1.0)
codes = np.ones(len(inside_vertices), dtype=mpath.Path.code_type) * mpath.Path.LINETO
codes[0] = mpath.Path.MOVETO

for i, (inside, outside) in enumerate(((1, 1), (1, -1), (-1, 1), (-1, -1))):
    # Concatenate the inside and outside subpaths together, changing their
    # order as needed
    vertices = np.concatenate((outside_vertices[::outside],
                               inside_vertices[::inside]))
    # Shift the path
    vertices[:, 0] += i * 2.5
    # The codes will be all "LINETO" commands, except for "MOVETO"s at the
    # beginning of each subpath
    all_codes = np.concatenate((codes, codes))
    # Create the Path object
    path = mpath.Path(vertices, all_codes)
    # Add plot it
    patch = mpatches.PathPatch(path, facecolor='#885500', edgecolor='black')
    ax.add_patch(patch)
    ax.annotate("Outside %s,\nInside %s" % (wise(outside), wise(inside)),
                (i * 2.5, -1.5), va="top", ha="center")

ax.set_xlim(-2, 10)
ax.set_ylim(-3, 2)
ax.set_title('Mmm, donuts!')
ax.set_aspect(1.0)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Demo to show use of the engineering Formatter.

```python
import matplotlib.pyplot as plt
import numpy as np
from matplotlib.ticker import EngFormatter

prng = np.random.RandomState(123)

fig, ax = plt.subplots()
ax.set_xscale('log')
formatter = EngFormatter(unit='Hz')
ax.xaxis.set_major_formatter(formatter)

xs = np.logspace(1, 9, 100)
ys = (0.8 + 0.4 * prng.uniform(size=100)) * np.log10(xs)**2
```
ax.plot(xs, ys)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

89.15 api example code: filled_step.py
Hatch-filled histograms

This example showcases the hatching capabilities of matplotlib by plotting various histograms.

```
import itertools
from collections import OrderedDict
from functools import partial
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.ticker as mticker
from cycler import cycler
from six.moves import zip

def filled_hist(ax, edges, values, bottoms=None, orientation='v', **kwargs):
    ""
    Draw a histogram as a stepped patch.

    Extra kwargs are passed through to `fill_between`

    Parameters
    ----------
    ax : Axes
        The axes to plot to

    edges : array
        A length n+1 array giving the left edges of each bin and the right edge of the last bin.

    values : array
        A length n array of bin counts or values

    bottoms : scalar or array, optional
        A length n array of the bottom of the bars. If None, zero is used.

    orientation : {'v', 'h'}
        Orientation of the histogram. 'v' (default) has the bars increasing in the positive y-direction.

    Returns
    -------
    ret : PolyCollection
        Artist added to the Axes
    ""
    print(orientation)
    if orientation not in set('hv'):
        raise ValueError("orientation must be in {{'h', 'v'}} "
                        "not {o}".format(o=orientation))
```

---

89.15. api example code: filled_step.py
kwargs.setdefault('step', 'post')
edges = np.asarray(edges)
values = np.asarray(values)
if len(edges) - 1 != len(values):
    raise ValueError('Must provide one more bin edge than value not: 
    'len(edges): {lb} len(values): {lv}'.format(lb=len(edges), lv=len(values)))

if bottoms is None:
    bottoms = np.zeros_like(values)
if np.isscalar(bottoms):
    bottoms = np.ones_like(values) * bottoms

values = np.r_[values, values[-1]]
bottoms = np.r_[bottoms, bottoms[-1]]
if orientation == 'h':
    return ax.fill_betweenx(edges, values, bottoms, **kwargs)
elif orientation == 'v':
    return ax.fill_between(edges, values, bottoms, **kwargs)
else:
    raise AssertionError("you should never be here")

def stack_hist(ax, stacked_data, sty_cycle, bottoms=None, hist_func=None, labels=None, plot_func=None, plot_kwargs=None):
    """
    ax : axes.Axes
    The axes to add artists too

    stacked_data : array or Mapping
    A (N, M) shaped array. The first dimension will be iterated over to
    compute histograms row-wise

    sty_cycle : Cycler or operable of dict
    Style to apply to each set

    bottoms : array, optional
    The initial positions of the bottoms, defaults to 0

    hist_func : callable, optional
    Must have signature 'bin_vals, bin_edges = f(data)'.
    'bin_edges' expected to be one longer than 'bin_vals'

    labels : list of str, optional
    The label for each set.

    If not given and stacked data is an array defaults to 'default set {n}''

    If stacked_data is a mapping, and labels is None, default to the keys
If stacked_data is a mapping and labels is given then only the columns listed by be plotted.

plot_func : callable, optional
Function to call to draw the histogram must have signature:

ret = plot_func(ax, edges, top, bottoms=bottoms,
        label=label, **kwargs)

plot_kwargs : dict, optional
Any extra kwargs to pass through to the plotting function. This will be the same for all calls to the plotting function and will over-ride the values in cycle.

Returns
-------
arts : dict
    Dictionary of artists keyed on their labels

# deal with default binning function
if hist_func is None:
    hist_func = np.histogram

# deal with default plotting function
if plot_func is None:
    plot_func = filled_hist

# deal with default
if plot_kwargs is None:
    plot_kwargs = {}

try:
    l_keys = stacked_data.keys()
    label_data = True
    if labels is None:
        labels = l_keys

except AttributeError:
    label_data = False
    if labels is None:
        labels = itertools.repeat(None)

if label_data:
    loop_iter = enumerate((stacked_data[lab], lab, s) for lab, s in zip(labels, sty_cycle))
else:
    loop_iter = enumerate(zip(stacked_data, labels, sty_cycle))

arts = {}
for j, (data, label, sty) in loop_iter:
    if label is None:
```python
label = 'dflt set {n}'.format(n=j)
label = sty.pop('label', label)
vals, edges = hist_func(data)
if bottoms is None:
    bottoms = np.zeros_like(vals)
top = bottoms + vals
print(sty)
sty.update(plot_kwargs)
print(sty)
ret = plot_func(ax, edges, top, bottoms=bottoms,
               label=label, **sty)
bottoms = top
arts[label] = ret
ax.legend(fontsize=10)
return arts

# set up histogram function to fixed bins
edges = np.linspace(-3, 3, 20, endpoint=True)
hist_func = partial(np.histogram, bins=edges)

# set up style cycles
color_cycle = cycler(facecolor=plt.rcParams['axes.prop_cycle'][:4])
label_cycle = cycler('label', ['set {n}'.format(n=n) for n in range(4)])
hatch_cycle = cycler('hatch', [('/', '*', '+', '|')])

# make some synthetic data
np.random.seed(0)
stack_data = np.random.randn(4, 12250)
dict_data = OrderedDict(zip((c['label'] for c in label_cycle), stack_data))

# work with plain arrays
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(9, 4.5), tight_layout=True)
arts = stack_hist(ax1, stack_data, color_cycle + label_cycle + hatch_cycle,
                  hist_func=hist_func)

arts = stack_hist(ax2, stack_data, color_cycle,
                  hist_func=hist_func,
                  plot_kwargs=dict(edgecolor='w', orientation='h'))

ax1.set_ylabel('counts')
ax1.set_xlabel('x')
ax2.set_xlabel('counts')
ax2.set_ylabel('x')

# work with labeled data
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(9, 4.5),
tight_layout=True, sharey=True)

arts = stack_hist(ax1, dict_data, color_cycle + hatch_cycle,
                  hist_func=hist_func)

arts = stack_hist(ax2, dict_data, color_cycle + hatch_cycle,
                  hist_func=hist_func)
```

---

2068 Chapter 89. api Examples
```python
hist_func=hist_func, labels=['set 0', 'set 3'])
ax1.xaxis.set_major_locator(mticker.MaxNLocator(5))
ax1.set_xlabel('counts')
ax1.set_ylabel('x')
ax2.set_ylabel('x')
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

89.16 api example code: font_family_rc.py

[source code]

""
""
Configuring the font family
""

You can explicitly set which font family is picked up for a given font
style (e.g., 'serif', 'sans-serif', or 'monospace').

In the example below, we only allow one font family (Tahoma) for the
sans-serif font style. You the default family with the font.family rc
param, e.g.,::

```python
cParams['font.family'] = 'sans-serif'
```

and for the font.family you set a list of font styles to try to find
in order::

```python
cParams['font.sans-serif'] = ['Tahoma', 'DejaVu Sans',
                           'Lucida Grande', 'Verdana']
```

""

# -*- noplot -*-

```python
from matplotlib import rcParams
rcParams['font.family'] = 'sans-serif'
rcParams['font.sans-serif'] = ['Tahoma']
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
ax.plot([1, 2, 3], label='test')
ax.legend()
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
# api example code: font_file.py

```
# -*- noplot -*-

""
Using a ttf font file in matplotlib
""

Although it is usually not a good idea to explicitly point to a single
ttf file for a font instance, you can do so using the
font_manager.FontProperties fname argument (for a more flexible
solution, see the font_family_rc.py and fonts_demo.py examples).

""

import sys
import os
import matplotlib.font_manager as fm
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
ax.plot([1, 2, 3])

if sys.platform == 'win32':
    fpath = 'C:\Windows\Fonts\Tahoma.ttf'
elif sys.platform.startswith('linux'):
    basedir = '/usr/share/fonts/truetype'
    fonts = ['freefont/FreeSansBoldOblique.ttf',
             'ttf-liberation/LiberationSans-BoldItalic.ttf',
             'msstcorefonts/Comic_Sans_MS.ttf']
    for fpath in fonts:
        if os.path.exists(os.path.join(basedir, fpath)):
            break
else:
    fpath = '/Library/Fonts/Tahoma.ttf'

if os.path.exists(fpath):
    prop = fm.FontProperties(fname=fpath)
    fname = os.path.split(fpath)[1]
    ax.set_title('this is a special font: %s' % fname, fontproperties=prop)
else:
    ax.set_title('Demo fails--cannot find a demo font')
ax.set_xlabel('This is the default font')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
This example shows how to use a path patch to draw a bunch of rectangles. The technique of using lots of Rectangle instances, or the faster method of using PolyCollections, were implemented before we had proper paths with moveto/lineto, closepoly etc in mpl. Now that we have them, we can draw collections of regularly shaped objects with homogeneous properties more efficiently with a PathCollection. This example makes a histogram -- its more work to set up the vertex arrays at the outset, but it should be much faster for large numbers of objects.

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import matplotlib.path as path

fig, ax = plt.subplots()
```
# histogram our data with numpy
data = np.random.randn(1000)
n, bins = np.histogram(data, 50)

# get the corners of the rectangles for the histogram
left = np.array(bins[:-1])
right = np.array(bins[1:])
bottom = np.zeros(len(left))
top = bottom + n

# we need a (numrects x numsides x 2) numpy array for the path helper
# function to build a compound path
XY = np.array([[left, left, right, right], [bottom, top, top, bottom]]).T

# get the Path object
barpath = path.Path.make_compound_path_from_polys(XY)

# make a patch out of it
patch = patches.PathPatch(barpath)
ax.add_patch(patch)

# update the view limits
ax.set_xlim(left[0], right[-1])
ax.set_ylim(bottom.min(), top.max())

plt.show()
89.19 api example code: image_zcoord.py

```python
import numpy as np
import matplotlib.pyplot as plt

X = 10*np.random.rand(5, 3)

fig, ax = plt.subplots()
ax.imshow(X, interpolation='nearest')

numrows, numcols = X.shape

def format_coord(x, y):
    col = int(x + 0.5)
    row = int(y + 0.5)
```

if col >= 0 and col < numcols and row >= 0 and row < numrows:
    z = X[row, col]
    return '%x=%1.4f, y=%1.4f, z=%1.4f' % (x, y, z)
else:
    return '%x=%1.4f, y=%1.4f' % (x, y)

ax.format_coord = format_coord
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

89.20 api example code: joinstyle.py

Join style

```
miter
128 degrees
128 degrees
128 degrees

round
64 degrees
64 degrees
64 degrees

bevel
32 degrees
32 degrees
32 degrees

16 degrees
16 degrees
16 degrees

8 degrees
8 degrees
8 degrees
```

Illustrate the three different join styles

```
import numpy as np
```
import matplotlib.pyplot as plt

def plot_angle(ax, x, y, angle, style):
    phi = np.radians(angle)
    xx = [x + .5, x, x + .5*np.cos(phi)]
    yy = [y, y, y + .5*np.sin(phi)]
    ax.plot(xx, yy, lw=8, color='blue', solid_joinstyle=style)
    ax.plot(xx[1:], yy[1:], lw=1, color='black')
    ax.plot(xx[1::-1], yy[1::-1], lw=1, color='black')
    ax.plot(xx[1:2], yy[1:2], 'o', color='red', markersize=3)
    ax.text(x, y + .2, '%.0f degrees' % angle)

fig, ax = plt.subplots()
ax.set_title('Join style')

for x, style in enumerate((('miter', 'round', 'bevel'))):
    ax.text(x, 5, style)
    for i in range(5):
        plot_angle(ax, x, i, pow(2.0, 3 + i), style)

ax.set_xlim(-.5, 2.75)
ax.set_ylim(-.5, 5.5)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
import numpy as np
import matplotlib.pyplot as plt

# Make some fake data.
a = b = np.arange(0, 3, .02)
c = np.exp(a)
d = c[::-1]

# Create plots with pre-defined labels.
fig, ax = plt.subplots()
ar.plot(a, c, label='Model length')
ar.plot(a, d, label='Data length')
ar.plot(a, c + d, label='Total message length')

fig.legend(['Model length', 'Data length', 'Total message length'])
legend = ax.legend(loc='upper center', shadow=True, fontsize='x-large')

# Put a nicer background color on the legend.
legend.get_frame().set_facecolor('#00FFCC')

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

89.22 api example code: line_with_text.py

```
import numpy as np
import matplotlib.pyplot as plt

Show how to override basic methods so an artist can contain another artist. In this case, the line contains a Text instance to label it.

import numpy as np
import matplotlib.pyplot as plt

```

89.22. api example code: line_with_text.py 2077
import matplotlib.lines as lines
import matplotlib.transforms as mtransforms
import matplotlib.text as mtext

class MyLine(lines.Line2D):
    def __init__(self, *args, **kwargs):
        # we'll update the position when the line data is set
        self.text = mtext.Text(0, 0, '')
        lines.Line2D.__init__(self, *args, **kwargs)

        # we can't access the label attr until *after* the line is
        # inited
        self.text.set_text(self.get_label())

    def set_figure(self, figure):
        self.text.set_figure(figure)
        lines.Line2D.set_figure(self, figure)

    def set_axes(self, axes):
        self.text.set_axes(axes)
        lines.Line2D.set_axes(self, axes)

    def set_transform(self, transform):
        # 2 pixel offset
        texttrans = transform + mtransforms.Affine2D().translate(2, 2)
        self.text.set_transform(texttrans)
        lines.Line2D.set_transform(self, transform)

    def set_data(self, x, y):
        if len(x):
            self.text.set_position((x[-1], y[-1]))

        lines.Line2D.set_data(self, x, y)

    def draw(self, renderer):
        # draw my label at the end of the line with 2 pixel offset
        lines.Line2D.draw(self, renderer)
        self.text.draw(renderer)

fig, ax = plt.subplots()
x, y = np.random.rand(2, 20)
line = MyLine(x, y, mfc='red', ms=12, label='line label')
#line.text.set_text('line label')
line.text.set_color('red')
line.text.set_fontsize(16)

ax.add_line(line)

plt.show()
89.23  api example code: logo2.py

import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
import matplotlib.cm as cm

mpl.rcParams['xtick.labelsize'] = 10
mpl.rcParams['ytick.labelsize'] = 12
mpl.rcParams['axes.edgecolor'] = 'gray'

axalpha = 0.05
figcolor = 'white'
dpi = 80
fig = plt.figure(figsize=(6, 1.1), dpi=dpi)
fig.figurePatch.set_edgecolor(figcolor)
fig.figurePatch.set_facecolor(figcolor)

def add_math_background():
    ax = fig.add_axes([0., 0., 1., 1.])
    text = []
    text.append(r"W^{3\beta}_{\delta_1 \rho_1 \sigma_2} = U^{3\beta}_{\delta_1 \rho_1} + \frac{1}{8 \pi 2} \int^\alpha_2_{\alpha_2} d \alpha'\left\{\frac{ U^{2\beta}_{\delta_1 \rho_1} - \alpha'U^{1\beta}_{\rho_1 \sigma_2} }{U^{0\beta}_{\rho_1 \sigma_2}}\right\}"")
    text.append(r"\frac{d\rho}{dt} + \rho \vec{v} \cdot \nabla \vec{v} = -\nabla p + \mu \nabla^2 \vec{v} + \rho \vec{g}"")

89.23. api example code: logo2.py
text.append(r'$\int_{-\infty}^{\infty} e^{-x^2} dx = \sqrt{\pi}$', 
(0.15, 0.3), 25))
text.append(r'$F_G = G \frac{m_1 m_2}{r^2}$', 
(0.85, 0.7), 30))
for eq, (x, y), size in text:
    ax.text(x, y, eq, ha='center', va='center', color='#11557c',
            alpha=0.25, transform=ax.transAxes, fontsize=size)
ax.set_axis_off()
return ax

def add_matplotlib_text(ax):
    ax.text(0.95, 0.5, 'matplotlib', color='#11557c', fontsize=65,
            ha='right', va='center', alpha=1.0, transform=ax.transAxes)

def add_polar_bar():
    ax = fig.add_axes([0.025, 0.075, 0.2, 0.85], projection='polar')
    ax.axesPatch.set_alpha(axalpha)
    ax.set_axisbelow(True)
    N = 7
    arc = 2. * np.pi
    theta = np.arange(0.0, arc, arc/N)
    radii = 10 * np.array([0.2, 0.6, 0.8, 0.7, 0.4, 0.5, 0.8])
    width = np.pi / 4 * np.array([0.4, 0.4, 0.6, 0.8, 0.2, 0.5, 0.3])
    bars = ax.bar(theta, radii, width=width, bottom=0.0)
    for r, bar in zip(radii, bars):
        bar.set_facecolor(cm.jet(r/10.))
        bar.set_alpha(0.6)
    for label in ax.get_xticklabels() + ax.get_yticklabels():
        label.set_visible(False)
    for line in ax.get_ygridlines() + ax.get_xgridlines():
        line.set_lw(0.8)
        line.set_alpha(0.9)
        line.set_ls('-')
        line.set_color('0.5')
    ax.set_yticks(np.arange(1, 9, 2))
    ax.set_rmax(9)

if __name__ == '__main__':
    main_axes = add_math_background()
    add_polar_bar()
    add_matplotlib_text(main_axes)
    plt.show()
A mathtext image as numpy array

This example shows how to make images from LaTeX strings.

```python
import matplotlib.mathtext as mathtext
import matplotlib.pyplot as plt
import matplotlib
matplotlib.rc('image', origin='upper')

parser = mathtext.MathTextParser("Bitmap")
parser.to_png('test2.png',
             r'\left[\left\lfloor\frac{5}{\frac{(3)}{4}}\right\rfloor\right]\left(\frac{\left(3\right)}{4}\right)',
             color='green', fontsize=14, dpi=100)

rgba1, depth1 = parser.to_rgba(r'IQ: $\sigma_i=15$', color='blue', fontsize=20, dpi=200)
rgba2, depth2 = parser.to_rgba(r'some other string', color='red', fontsize=20, dpi=200)
```
fig = plt.figure()
fig.figimage(rgba1.astype(float)/255., 100, 100)
fig.figimage(rgba2.astype(float)/255., 100, 300)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

89.25 api example code: patch_collection.py

```python
ten= np.linspace(0, 2*np.pi, 100)

circles = [Circle((0.5, 0.5), 0.3) for _ in range(10)]
wedges = [Wedge((0.5, 0.5), 0.3, 0.3, 0.6, a) for a in range(0, 360, 45)]
polygons = [Polygon([[0.2, 0.2], [0.2, 0.3], [0.3, 0.3], [0.3, 0.2]]) for _ in range(5)]

collection = PatchCollection(circles + wedges + polygons, facecolors='blue', alpha=0.5)
fig, ax = plt.subplots()
ax.add_collection(collection)
plt.show()
```

---

Circles, Wedges and Polygons

---

import numpy as np
import matplotlib
from matplotlib.patches import Circle, Wedge, Polygon
from matplotlib.collections import PatchCollection
import matplotlib.pyplot as plt
fig, ax = plt.subplots()

resolution = 50  # the number of vertices
N = 3
x = np.random.rand(N)
y = np.random.rand(N)
radii = 0.1*np.random.rand(N)
patches = []
for x1, y1, r in zip(x, y, radii):
    circle = Circle((x1, y1), r)
    patches.append(circle)

x = np.random.rand(N)
y = np.random.rand(N)
radii = 0.1*np.random.rand(N)
theta1 = 360.0*np.random.rand(N)
theta2 = 360.0*np.random.rand(N)
for x1, y1, r, t1, t2 in zip(x, y, radii, theta1, theta2):
    wedge = Wedge((x1, y1), r, t1, t2)
    patches.append(wedge)

# Some limiting conditions on Wedge
patches += [
    Wedge((.3, .7), .1, 0, 360),  # Full circle
    Wedge((.7, .8), .2, 0, 360, width=0.05),  # Full ring
    Wedge((.8, .3), .2, 0, 45),  # Full sector
    Wedge((.8, .3), .2, 45, 90, width=0.10),  # Ring sector
]

for i in range(N):
    polygon = Polygon(np.random.rand(N, 2), True)
    patches.append(polygon)

colors = 100*np.random.rand(len(patches))
p = PatchCollection(patches, alpha=0.4)
p.set_array(np.array(colors))
ax.add_collection(p)
fig.colorbar(p, ax=ax)

plt.show()
Exploring normalizations

Let's explore various normalization on a multivariate normal distribution.

```python
from matplotlib import pyplot as plt
import matplotlib.colors as mcolors
import numpy as np
from numpy.random import multivariate_normal

data = np.vstack([multivariate_normal([10, 10], [[3, 2], [2, 3]], size=100000),
                  multivariate_normal([30, 20], [[2, 3], [1, 3]], size=1000)]
gammas = [0.8, 0.5, 0.3]
fig, axes = plt.subplots(nrows=2, ncols=2)
```

Linear normalization

Power law ($\gamma = 0.8$)

Power law ($\gamma = 0.5$)

Power law ($\gamma = 0.3$)
```python
axes[0, 0].set_title('Linear normalization')
axes[0, 0].hist2d(data[:, 0], data[:, 1], bins=100)

for ax, gamma in zip(axes.flat[1:], gammas):
    ax.set_title('Power law $(\gamma=%.1f)$' % gamma)
    ax.hist2d(data[:, 0], data[:, 1], bins=100, norm=mcolors.PowerNorm(gamma))

fig.tight_layout()
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

89.27 api example code: quad_bezier.py

The red point should be on the path
This example showcases the PathPatch object to create a Bezier polycurve path patch.

```python
import matplotlib.path as mpath
import matplotlib.patches as mpatches
import matplotlib.pyplot as plt
Path = mpath.Path

fig, ax = plt.subplots()
pp1 = mpatches.PathPatch(
    Path([(0, 0), (1, 0), (1, 1), (0, 0)],
         [Path.MOVETO, Path.CURVE3, Path.CURVE3, Path.CLOSEPOLY]),
    fc="none", transform=ax.transData)
ax.add_patch(pp1)
ax.plot([0.75], [0.25], "ro")
ax.set_title('The red point should be on the path')
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
5-Factor Solution Profiles Across Four Scenarios

Radar chart (aka spider or star chart)

This example creates a radar chart, also known as a spider or star chart [1].

Although this example allows a frame of either 'circle' or 'polygon', polygon frames don't have proper gridlines (the lines are circles instead of polygons). It's possible to get a polygon grid by setting GRIDLINE_INTERPOLATION_STEPS in...
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.path import Path
from matplotlib.spines import Spine
from matplotlib.projections.polar import PolarAxes
from matplotlib.projections import register_projection

def radar_factory(num_vars, frame='circle'):
    """Create a radar chart with `num_vars` axes.

    This function creates a RadarAxes projection and registers it.
    
    Parameters
    ----------
    num_vars : int
        Number of variables for radar chart.
    frame : {'circle' | 'polygon'}
        Shape of frame surrounding axes.
    """
    theta = np.linspace(0, 2*np.pi, num_vars, endpoint=False)
    theta += np.pi/2

def draw_poly_patch(self):
    verts = unit_poly_verts(theta)
    return plt.Polygon(verts, closed=True, edgecolor='k')

def draw_circle_patch(self):
    # unit circle centered on (0.5, 0.5)
    return plt.Circle((0.5, 0.5), 0.5)

    patch_dict = {'polygon': draw_poly_patch, 'circle': draw_circle_patch}
    if frame not in patch_dict:
        raise ValueError('unknown value for `frame`: %s' % frame)

class RadarAxes(PolarAxes):

    name = 'radar'
    # use 1 line segment to connect specified points
    RESOLUTION = 1
    # define draw_frame method
    draw_patch = patch_dict[frame]

    def fill(self, *args, **kwargs):
"""Override fill so that line is closed by default""

closed = kwargs.pop('closed', True)
return super(RadarAxes, self).fill(closed=closed, *args, **kwargs)

def plot(self, *args, **kwargs):
    """Override plot so that line is closed by default""
    lines = super(RadarAxes, self).plot(*args, **kwargs)
    for line in lines:
        self._close_line(line)

def _close_line(self, line):
    x, y = line.get_data()
    # FIXME: markers at x[0], y[0] get doubled-up
    if x[0] != x[-1]:
        x = np.concatenate((x, [x[0]]))
        y = np.concatenate((y, [y[0]]))
        line.set_data(x, y)

def set_varlabels(self, labels):
    self.set_thetagrids(np.degrees(theta), labels)

def _gen_axes_patch(self):
    return self.draw_patch()

def _gen_axes_spines(self):
    if frame == 'circle':
        return PolarAxes._gen_axes_spines(self)
    # The following is a hack to get the spines (i.e. the axes frame)
    # to draw correctly for a polygon frame.
    # spine_type must be 'left', 'right', 'top', 'bottom', or 'circle'.
    spine_type = 'circle'
    verts = unit_poly_verts(theta)
    # close off polygon by repeating first vertex
    verts.append(verts[0])
    path = Path(verts)
    spine = Spine(self, spine_type, path)
    spine.set_transform(self.transAxes)
    return {'polar': spine}

register_projection(RadarAxes)

theta

def unit_poly_verts(theta):
    """Return vertices of polygon for subplot axes.

    This polygon is circumscribed by a unit circle centered at (0.5, 0.5)
    """
    x0, y0, r = [0.5] * 3
    verts = [(r*np.cos(t) + x0, r*np.sin(t) + y0) for t in theta]
    return verts
def example_data():
    # The following data is from the Denver Aerosol Sources and Health study.
    # The data are pollution source profile estimates for five modeled
    # pollution sources (e.g., cars, wood-burning, etc) that emit 7-9 chemical
    # species. The radar charts are experimented with here to see if we can
    # nicely visualize how the modeled source profiles change across four
    # scenarios:
    # 1) No gas-phase species present, just seven particulate counts on
    #    Sulfate
    #    Nitrate
    #    Elemental Carbon (EC)
    # 2) Inclusion of gas-phase specie carbon monoxide (CO)
    # 3) Inclusion of gas-phase specie ozone (O3).
    # 4) Inclusion of both gas-phase species is present...
    data = [
        ['Sulfate', 'Nitrate', 'EC', 'OC1', 'OC2', 'OC3', 'OP', 'CO', 'O3'],
        ('Basecase', [
            [0.88, 0.01, 0.03, 0.03, 0.00, 0.06, 0.01, 0.00, 0.00],
            [0.07, 0.95, 0.04, 0.05, 0.00, 0.02, 0.01, 0.00, 0.00],
            [0.01, 0.02, 0.85, 0.19, 0.10, 0.00, 0.00, 0.00],
            [0.02, 0.01, 0.07, 0.01, 0.21, 0.12, 0.98, 0.00, 0.00],
            [0.01, 0.01, 0.02, 0.71, 0.74, 0.70, 0.00, 0.00, 0.00]]),
        ('With CO', [
            [0.88, 0.02, 0.02, 0.02, 0.00, 0.05, 0.00, 0.05, 0.00],
            [0.08, 0.94, 0.04, 0.02, 0.00, 0.01, 0.12, 0.04, 0.00],
            [0.01, 0.01, 0.79, 0.10, 0.00, 0.05, 0.00, 0.31, 0.00],
            [0.00, 0.02, 0.03, 0.38, 0.31, 0.31, 0.00, 0.59, 0.00],
            [0.02, 0.02, 0.11, 0.47, 0.69, 0.58, 0.88, 0.00, 0.00]))],
        ('With O3', [
            [0.89, 0.01, 0.07, 0.00, 0.00, 0.05, 0.00, 0.00, 0.03],
            [0.07, 0.95, 0.05, 0.04, 0.00, 0.02, 0.12, 0.00, 0.00],
            [0.01, 0.02, 0.86, 0.27, 0.16, 0.19, 0.00, 0.00, 0.00],
            [0.01, 0.03, 0.00, 0.32, 0.29, 0.27, 0.00, 0.00, 0.95],
            [0.02, 0.00, 0.03, 0.37, 0.56, 0.47, 0.87, 0.00, 0.00]))],
        ('CO & O3', [
            [0.87, 0.01, 0.08, 0.00, 0.00, 0.04, 0.00, 0.00, 0.01],
            [0.09, 0.95, 0.02, 0.03, 0.00, 0.01, 0.13, 0.06, 0.00],
            [0.01, 0.02, 0.71, 0.24, 0.13, 0.16, 0.00, 0.50, 0.00],
            [0.01, 0.03, 0.00, 0.28, 0.24, 0.23, 0.00, 0.44, 0.88],
            [0.02, 0.00, 0.18, 0.45, 0.64, 0.55, 0.86, 0.00, 0.16]])
    return data

if __name__ == '__main__':
N = 9
theta = radar_factory(N, frame='polygon')
data = example_data()
spoke_labels = data.pop(0)

fig, axes = plt.subplots(figsize=(9, 9), nrows=2, ncols=2,
                        subplot_kw=dict(projection='radar'))
fig.subplots_adjust(wspace=0.25, hspace=0.20, top=0.85, bottom=0.05)

colors = ['b', 'r', 'g', 'm', 'y']
# Plot the four cases from the example data on separate axes
for ax, (title, case_data) in zip(axes.flatten(), data):
    ax.set_rgrids([0.2, 0.4, 0.6, 0.8])
    ax.set_title(title, weight='bold', size='medium',
                 position=(0.5, 1.1), horizontalalignment='center',
                 verticalalignment='center')
    for d, color in zip(case_data, colors):
        ax.plot(theta, d, color=color)
        ax.fill(theta, d, facecolor=color, alpha=0.25)
    ax.set_varlabels(spoke_labels)

# add legend relative to top-left plot
ax = axes[0, 0]
labels = ('Factor 1', 'Factor 2', 'Factor 3', 'Factor 4', 'Factor 5')
legend = ax.legend(labels, loc=(0.9, .95),
                   labelsspacing=0.1, fontsize='small')

fig.text(0.5, 0.965, '5-Factor Solution Profiles Across Four Scenarios',
         horizontalalignment='center', color='black', weight='bold',
         size='large')

plt.show()
The default settings produce a diagram like this.
The Sankey class

Demonstrate the Sankey class by producing three basic diagrams.

```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.sankey import Sankey

# Example 1 -- Mostly defaults
# This demonstrates how to create a simple diagram by implicitly calling the
# Sankey.add() method and by appending finish() to the call to the class.
Sankey(flows=[0.25, 0.15, 0.60, -0.20, -0.15, -0.05, -0.50, -0.10],
labels=['', '', '', 'First', 'Second', 'Third', 'Fourth', 'Fifth'],
orientations=[-1, 1, 0, 1, 1, 1, 0, -1]).finish()
plt.title("The default settings produce a diagram like this.")
```

# Notice:
# 1. Axes weren’t provided when Sankey() was instantiated, so they were
#    created automatically.
# 2. The scale argument wasn’t necessary since the data was already
#    normalized.
# 3. By default, the lengths of the paths are justified.
# Example 2

This demonstrates:

1. Setting one path longer than the others
2. Placing a label in the middle of the diagram
3. Using the scale argument to normalize the flows
4. Implicitly passing keyword arguments to PathPatch()
5. Changing the angle of the arrow heads
6. Changing the offset between the tips of the paths and their labels
7. Formatting the numbers in the path labels and the associated unit
8. Changing the appearance of the patch and the labels after the figure is created

```python
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1, xticks=[], yticks=[],
               title="Flow Diagram of a Widget")
sankey = Sankey(ax=ax, scale=0.01, offset=0.2, head_angle=180,
               format='%.0f', unit='\%

flows=[25, 0, 60, -10, -20, -5, -15, -10, -40],
labels=['', '', '', 'First', 'Second', 'Third', 'Fourth',
       'Fifth', 'Hurray!'],
orientations=[-1, 1, 0, 1, 1, -1, -1, 0],
pathlengths=[0.25, 0.25, 0.25, 0.25, 0.25, 0.6, 0.25, 0.25,
3.25],
patchlabel="Widget\nA")  # Arguments to matplotlib.patches.PathPatch()
diagrams = sankey.finish()
diagrams[-1].texts[-1].set_color(r'"

diagrams[-1].text.set_fontweight('bold')
```

# Notice:

1. Since the sum of the flows is nonzero, the width of the trunk isn't uniform. If verbose.level is helpful (in matplotlibrc), a message is given in the terminal window.
2. The second flow doesn't appear because its value is zero. Again, if verbose.level is helpful, a message is given in the terminal window.

# Example 3

This demonstrates:

1. Connecting two systems
2. Turning off the labels of the quantities
3. Adding a legend

```python
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1, xticks=[], yticks=[], title="Two Systems")
flows = [0.25, 0.15, 0.60, -0.10, -0.05, -0.25, -0.15, -0.10, -0.35]
sankey = Sankey(ax=ax, unit=None)
sankey.add(flows=flows, label='one',
          orientations=[-1, 1, 0, 1, 1, -1, -1, 0])
sankey.add(flows=[-0.25, 0.15, 0.1], label='two',
          orientations=[-1, -1, -1], prior=0, connect=(0, 0))
diagrams = sankey.finish()
diagrams[-1].patch.set_hatch('/*')
plt.legend(loc='best')
```

# Notice that only one connection is specified, but the systems form a circuit since: (1) the lengths of the paths are justified and (2) the orientation and ordering of the flows is mirrored.
Why would you want to do this?
(But you could.)

"""
======================================
Long chain of connections using Sankey
======================================

Demonstrate/test the Sankey class by producing a long chain of connections.
"""

```python
from itertools import cycle
import matplotlib.pyplot as plt
from matplotlib.sankey import Sankey

links_per_side = 6
```
def side(sankey, n=1):
    """Generate a side chain.""
    prior = len(sankey.diagrams)
    for i in range(0, 2*n, 2):
        sankey.add(flows=[1, -1], orientations=[-1, -1],
                   patchlabel=str(prior + i),
                   prior=prior + i - 1, connect=(1, 0), alpha=0.5)
        sankey.add(flows=[1, -1], orientations=[1, 1],
                   patchlabel=str(prior + i + 1),
                   prior=prior + i, connect=(1, 0), alpha=0.5)

def corner(sankey):
    """Generate a corner link.""
    prior = len(sankey.diagrams)
    sankey.add(flows=[1, -1], orientations=[0, 1],
               patchlabel=str(prior), facecolor='k',
               prior=prior - 1, connect=(1, 0), alpha=0.5)

fig = plt.figure()
ax = fig.add_subplot(1, 1, 1, xticks=[], yticks=[],
                     title="Why would you want to do this?\n(But you could.)")
sankey = Sankey(ax=ax, unit=None)
sankey.add(flows=[1, -1], orientations=[0, 1],
           patchlabel="0", facecolor='k',
           rotation=45)
side(sankey, n=links_per_side)
corner(sankey)
side(sankey, n=links_per_side)
corner(sankey)
side(sankey, n=links_per_side)
corner(sankey)
side(sankey, n=links_per_side)
sankey.finish()
# Notice:
# 1. The alignment doesn't drift significantly (if at all; with 16007
#     subdiagrams there is still closure).
# 2. The first diagram is rotated 45 deg, so all other diagrams are rotated
#     accordingly.
plt.show()
Sankey diagram

Hurray!

First
10%
Second
20%
Third
5%
Fourth
15%
Fifth
10%
60%
25%

This example showcases a more complex sankey diagram.

```
from __future__ import print_function
__author__ = "Yannick Copin <ycopin@ipnl.in2p3.fr>"
__version__ = "Time-stamp: <10/02/2010 16:49 ycopin@lyopc548.in2p3.fr>"
import numpy as np

def sankey(ax,
          outputs=[100.], outlabels=None,
          inputs=[100.], inlabels='',
          dx=40, dy=10, outangle=45, w=3, inangle=30, offset=2, **kwargs):
    """Draw a Sankey diagram."""
```
outputs: array of outputs, should sum up to 100%
outlabels: output labels (same length as outputs),
or None (use default labels) or '' (no labels)
inputs and inlabels: similar for inputs
dx: horizontal elongation
dy: vertical elongation
outangle: output arrow angle [deg]
w: output arrow shoulder
inangle: input dip angle
offset: text offset
**kwargs: propagated to Patch (e.g., fill=False)

Return (patch,[intexts,outtexts]).

```
import matplotlib.patches as mpatches
from matplotlib.path import Path

outs = np.absolute(outputs)
outsigns = np.sign(outputs)
outsigns[-1] = 0 # Last output

ins = np.absolute(inputs)
insigns = np.sign(inputs)
insigns[0] = 0 # First input

assert sum(outs) == 100, "Outputs don't sum up to 100%"
assert sum(ins) == 100, "Inputs don't sum up to 100%"

def add_output(path, loss, sign=1):
    # Arrow tip height
    h = (loss/2 + w) * np.tan(np.radians(outangle))
    move, (x, y) = path[-1]  # Use last point as reference
    if sign == 0:
        # Final loss (horizontal)
        path.extend(((Path.LINETO, [x + dx, y]),
                      (Path.LINETO, [x + dx, y + w]),
                      (Path.LINETO, [x + dx + h, y - loss/2]),  # Tip
                      (Path.LINETO, [x + dx, y - loss - w]),
                      (Path.LINETO, [x + dx, y - loss])))
        outtips.append((sign, path[-3][1]))
    else:
        # Intermediate loss (vertical)
        path.extend(((Path.CURVE4, [x + dx/2, y]),
                      (Path.CURVE4, [x + dx, y]),
                      (Path.CURVE4, [x + dx, y + sign*dy]),
                      (Path.LINETO, [x + dx - w, y + sign*dy]),
                      (Path.LINETO, [x + dx + loss/2, y + sign*(dy + h)]),
                      (Path.LINETO, [x + dx + loss + w, y + sign*dy]),
                      (Path.LINETO, [x + dx + loss, y + sign*dy]),
                      (Path.CURVE3, [x + dx + loss, y - sign*loss]),
                      (Path.CURVE3, [x + dx/2 + loss, y - sign*loss])))
        outtips.append((sign, path[-5][1]))
```
def add_input(path, gain, sign=1):
    h = (gain / 2) * np.tan(np.radians(inangle))  # Dip depth
    move, (x, y) = path[-1]  # Use last point as reference
    if sign == 0:  # First gain (horizontal)
        path.extend([(Path.LINETO, [x - dx, y]),
                      (Path.LINETO, [x - dx + h, y + gain/2]),  # Dip
                      (Path.LINETO, [x - dx, y + gain])])
        xd, yd = path[-2][1]  # Dip position
        indips.append((sign, [xd - h, yd]))
    else:  # Intermediate gain (vertical)
        path.extend([(Path.CURVE4, [x - dx/2, y]),
                      (Path.CURVE4, [x - dx, y]),
                      (Path.CURVE4, [x - dx, y + sign*dy]),  # Dip
                      (Path.LINETO, [x - dx - gain / 2, y + sign*(dy - h)]),
                      (Path.LINETO, [x - dx - gain, y + sign*dy]),
                      (Path.CURVE3, [x - dx - gain, y - sign*gain]),
                      (Path.CURVE3, [x - dx/2 - gain, y - sign*gain])])
        xd, yd = path[-4][1]  # Dip position
        indips.append((sign, [xd, yd + sign*h]))

outtips = []  # Output arrow tip dir. and positions
urpath = [(Path.MOVETO, [0, 100])]  # 1st point of upper right path
lrpath = [(Path.LINETO, [0, 0])]  # 1st point of lower right path
for loss, sign in zip(outs, outsigns):
    add_output(sign >= 0 and urpath or lrpath, loss, sign=sign)

indips = []  # Input arrow tip dir. and positions
llpath = [(Path.LINETO, [0, 0])]  # 1st point of lower left path
ulpath = [(Path.MOVETO, [0, 100])]  # 1st point of upper left path
for gain, sign in reversed(list(zip(ins, insigns))):
    add_input(sign <= 0 and llpath or ulpath, gain, sign=sign)

def revert(path):
    """A path is not just revertable by path[::-1] because of Bezier curves."""
    rpath = []
    nextmove = Path.LINETO
    for move, pos in path[::-1]:
        rpath.append((nextmove, pos))
        nextmove = move
    return rpath

# Concatenate subpathes in correct order
path = urpath + revert(lrpath) + llpath + revert(ulpath)

codes, verts = zip(*path)
verts = np.array(verts)

# Path patch
path = Path(verts, codes)
patch = mpatches.PathPatch(path, **kwargs)
ax.add_patch(patch)

if False:  # DEBUG
    print("urpath", urpath)
    print("lrpath", revert(lrpath))
    print("llpath", llpath)
    print("ulpath", revert(ulpath))
    xs, ys = zip(*verts)
    ax.plot(xs, ys, 'go-')

# Labels

def set_labels(labels, values):
    """Set or check labels according to values."""
    if labels == '':  # No labels
        return labels
    elif labels is None:  # Default labels
        return ['%2d%%' % val for val in values]
    else:
        assert len(labels) == len(values)
        return labels

def put_labels(labels, positions, output=True):
    """Put labels to positions.""
    texts = []
    lbls = output and labels or labels[::-1]
    for i, label in enumerate(lbls):
        s, (x, y) = positions[i]  # Label direction and position
        if s == 0:
            t = ax.text(x + offset, y, label,
                        ha=output and 'left' or 'right', va='center')
        elif s > 0:
            t = ax.text(x, y + offset, label, ha='center', va='bottom')
        else:
            t = ax.text(x, y - offset, label, ha='center', va='top')
        texts.append(t)
    return texts

outlabels = set_labels(outlabels, outs)
outtexts = put_labels(outlabels, outtips, output=True)

inlabels = set_labels(inlabels, ins)
intexts = put_labels(inlabels, indips, output=False)

# Axes management
ax.set_xlim(verts[:, 0].min() - dx, verts[:, 0].max() + dx)
ax.set_ylim(verts[:, 1].min() - dy, verts[:, 1].max() + dy)
ax.set_aspect('equal', adjustable='datalim')

return patch, [intexts, outtexts]

if __name__ == '__main__':
```python
import matplotlib.pyplot as plt

outputs = [10., -20., 5., 15., -10., 40.]
outlabels = ['First', 'Second', 'Third', 'Fourth', 'Fifth', 'Hurray!']
outlabels = [s + '\n%d%%' % abs(l) for l, s in zip(outputs, outlabels)]

inputs = [60., -25., 15.]

fig = plt.figure()
ax = fig.add_subplot(1, 1, 1, xticks=[], yticks=[], title="Sankey diagram")

patch, (intexts, outtexts) = sankey(ax, outputs=outputs,
                                    outlabels=outlabels, inputs=inputs,
                                    inlabels=None)

outtexts[1].set_color('r')
outtexts[-1].set_fontweight('bold')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
89.32 api example code: sankey_demo_rankine.py

Rankine Power Cycle: Example 8.6 from Moran and Shapiro

Demonstrate the Sankey class with a practical example of a Rankine power cycle.
```python
import matplotlib.pyplot as plt
from matplotlib.sankey import Sankey

fig = plt.figure(figsize=(8, 9))
ax = fig.add_subplot(1, 1, 1, xticks=[], yticks=[],
title="Rankine Power Cycle: Example 8.6 from Moran and "
"Shapiro\x22Fundamentals of Engineering Thermodynamics "
"\x22, 6th ed., 2008")

Hdot = [260.431, 35.078, 180.794, 221.115, 22.700,
142.361, 10.193, 10.210, 43.670, 44.312,
68.631, 10.758, 10.758, 0.017, 0.642,
232.121, 44.559, 100.613, 132.168] # MW

sankey = Sankey(ax=ax, format='%0.3G', unit='MW', gap=0.5, scale=1.0/Hdot[0])
sankey.add(patchlabel='Pump 1', rotation=90, facecolor='#37c959',
flows=[Hdot[13], Hdot[6], -Hdot[7]],
labels=['Shaft power', ' ', None],
pathlengths=[0.4, 0.883, 0.25],
orientations=[1, -1, 0])
sankey.add(patchlabel='Open heater', facecolor='#37c959',
flows=[Hdot[11], Hdot[7], Hdot[4], -Hdot[8]],
labels=[None, ', ', None, None],
pathlengths=[0.25, 0.25, 1.93, 0.25],
orientations=[1, 0, -1, 0], prior=0, connect=(2, 1))
sankey.add(patchlabel='Pump 2', facecolor='#37c959',
flows=[Hdot[14], Hdot[8], -Hdot[9]],
labels=['Shaft power', ' ', None],
pathlengths=[0.4, 0.25, 0.25],
orientations=[1, 0, 0], prior=1, connect=(3, 1))
sankey.add(patchlabel='Closed heater', trunklength=2.914, fc='#37c959',
flows=[Hdot[9], Hdot[1], -Hdot[11], -Hdot[10]],
labels=['', '', None, None],
pathlengths=[0.25, 1.543, 0.25, 0.25],
orientations=[0, -1, 1, -1], prior=2, connect=(2, 0))
sankey.add(patchlabel='Trap', facecolor='#37c959', trunklength=5.102,
flows=[Hdot[11], -Hdot[12]],
labels=['
', None],
pathlengths=[1.0, 1.01],
orientations=[1, 1], prior=3, connect=(2, 0))
sankey.add(patchlabel='Steam\ngenerator', facecolor='ff5555',
flows=[Hdot[15], Hdot[10], Hdot[2], -Hdot[3], -Hdot[0]],
labels=['Heat rate', '', '', None, None],
pathlengths=0.25,
orientations=[1, 0, -1, -1, -1], prior=3, connect=(3, 1))
sankey.add(patchlabel='\n\nTurbine 1', facecolor='#37c959',
flows=[Hdot[0], -Hdot[16], -Hdot[1], -Hdot[2]],
labels=['', None, None, None],
pathlengths=[0.25, 0.153, 1.543, 0.25],
orientations=[0, 1, -1, -1], prior=5, connect=(4, 0))
sankey.add(patchlabel='\n\nReheat', facecolor='#37c959',
```
flows=[Hdot[2], -Hdot[2]],
labels=[None, None],
pathlengths=[0.725, 0.25],
orientations=[-1, 0], prior=6, connect=(3, 0))
sankey.add(patchlabel='Turbine 2', trunklength=3.212, facecolor='#37c959',
flows=[Hdot[3], Hdot[16], -Hdot[5], -Hdot[4], -Hdot[17]],
labels=[None, 'Shaft power', None, '', 'Shaft power'],
pathlengths=[0.751, 0.15, 0.25, 1.93, 0.25],
orientations=[0, -1, 0, -1, 1], prior=6, connect=(1, 1))
sankey.add(patchlabel='Condenser', facecolor='#58b1fa', trunklength=1.764,
flows=[Hdot[5], -Hdot[18], -Hdot[6]],
labels=['', 'Heat rate', None],
pathlengths=[0.45, 0.25, 0.883],
orientations=[-1, 1, 0], prior=8, connect=(2, 0))

diagrams = sankey.finish()
for diagram in diagrams:
    diagram.text.set_fontweight('bold')
    diagram.text.set_fontsize('10')
    for text in diagram.texts:
        text.set_fontsize('10')

# Notice that the explicit connections are handled automatically, but the
# implicit ones currently are not. The lengths of the paths and the trunks
# must be adjusted manually, and that is a bit tricky.

plt.show()
This example makes custom 'pie charts' as the markers for a scatter plot.

Thanks to Manuel Metz for the example

```python
import math
import numpy as np
import matplotlib.pyplot as plt

# first define the ratios
r1 = 0.2  # 20%
r2 = r1 + 0.4  # 40%

# define some sizes of the scatter marker
sizes = [60, 80, 120]

# calculate the points of the first pie marker
```
# these are just the origin (0,0) +
# some points on a circle cos,sin
x = [0] + np.cos(np.linspace(0, 2*math.pi*r1, 10)).tolist()
y = [0] + np.sin(np.linspace(0, 2*math.pi*r1, 10)).tolist()
xy1 = list(zip(x, y))
s1 = max(max(x), max(y))

# ...
x = [0] + np.cos(np.linspace(2*math.pi*r1, 2*math.pi*r2, 10)).tolist()
y = [0] + np.sin(np.linspace(2*math.pi*r1, 2*math.pi*r2, 10)).tolist()
xy2 = list(zip(x, y))
s2 = max(max(x), max(y))

x = [0] + np.cos(np.linspace(2*math.pi*r2, 2*math.pi, 10)).tolist()
y = [0] + np.sin(np.linspace(2*math.pi*r2, 2*math.pi, 10)).tolist()
xy3 = list(zip(x, y))
s3 = max(max(x), max(y))

fig, ax = plt.subplots()
ax.scatter(np.arange(3), np.arange(3), marker=(xy1, 0),
    s=[s1*s1*__ for __ in sizes], facecolor='blue')
ax.scatter(np.arange(3), np.arange(3), marker=(xy2, 0),
    s=[s2*s2*__ for __ in sizes], facecolor='green')
ax.scatter(np.arange(3), np.arange(3), marker=(xy3, 0),
    s=[s3*s3*__ for __ in sizes], facecolor='red')
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
This serves as an intensive exercise of matplotlib's transforms and custom projection API. This example produces a so-called SkewT-logP diagram, which is a common plot in meteorology for displaying vertical profiles of temperature. As far as matplotlib is concerned, the complexity comes from having X and Y axes that are not orthogonal. This is handled by including a skew component to the basic Axes transforms. Additional complexity comes in handling the fact that the upper and lower X-axes have different data ranges, which necessitates a bunch of custom classes for ticks, spines, and the axis to handle this.
```python
from matplotlib.axes import Axes
import matplotlib.transforms as transforms
import matplotlib.axis as maxis
import matplotlib.spines as mspines
from matplotlib.projections import register_projection

# The sole purpose of this class is to look at the upper, lower, or total
# interval as appropriate and see what parts of the tick to draw, if any.
class SkewXTick(maxis.XTick):
    def update_position(self, loc):
        # This ensures that the new value of the location is set before
        # any other updates take place
        self._loc = loc
        super(SkewXTick, self).update_position(loc)

    def _has_default_loc(self):
        return self.get_loc() is None

    def _need_lower(self):
        return (self._has_default_loc() or
                transforms.interval_contains(self.axes.lower_xlim,
                                              self.get_loc()))

    def _need_upper(self):
        return (self._has_default_loc() or
                transforms.interval_contains(self.axes.upper_xlim,
                                              self.get_loc()))

@property
def gridOn(self):
    return (self._gridOn and (self._has_default_loc() or
                               transforms.interval_contains(self.get_view_interval(),
                                                             self.get_loc())))

@gridOn.setter
def gridOn(self, value):
    self._gridOn = value

@property
def tick1On(self):
    return self._tick1On and self._need_lower()

@tick1On.setter
def tick1On(self, value):
    self._tick1On = value

@property
def label1On(self):
    return self._label1On and self._need_lower()
```

@label1On.setter
def label1On(self, value):
    self._label1On = value

@property
def tick2On(self):
    return self._tick2On and self._need_upper()

@tick2On.setter
def tick2On(self, value):
    self._tick2On = value

@property
def label2On(self):
    return self._label2On and self._need_upper()

@label2On.setter
def label2On(self, value):
    self._label2On = value

def get_view_interval(self):
    return self.axes.xaxis.get_view_interval()

# This class exists to provide two separate sets of intervals to the tick,
# as well as create instances of the custom tick
class SkewXAxis(maxis.XAxis):
    def _get_tick(self, major):
        return SkewXTick(self.axes, None, '', major=major)

    def get_view_interval(self):
        return self.axes.upper_xlim[0], self.axes.lower_xlim[1]

# This class exists to calculate the separate data range of the
# upper X-axis and draw the spine there. It also provides this range
# to the X-axis artist for ticking and gridlines
class SkewSpine(mspines.Spine):
    def _adjust_location(self):
        pts = self._path.vertices
        if self.spine_type == 'top':
            pts[:, 0] = self.axes.upper_xlim
        else:
            pts[:, 0] = self.axes.lower_xlim

# This class handles registration of the skew-xaxes as a projection as well
# as setting up the appropriate transformations. It also overrides standard
# spines and axes instances as appropriate.
class SkewXAxes(Axes):
    # The projection must specify a name. This will be used be the
    # user to select the projection, i.e. `subplot(111,"skew",

```python
# projection='skewx'
name = 'skewx'

def _init_axis(self):
    # Taken from Axes and modified to use our modified X-axis
    self.xaxis = SkewXAxis(self)
    self.spines['top'].register_axis(self.xaxis)
    self.spines['bottom'].register_axis(self.xaxis)
    self.yaxis = maxis.YAxis(self)
    self.spines['left'].register_axis(self.yaxis)
    self.spines['right'].register_axis(self.yaxis)

def _gen_axes_spines(self):
    spines = {
        'top': SkewSpine.linear_spine(self, 'top'),
        'bottom': mspines.Spine.linear_spine(self, 'bottom'),
        'left': mspines.Spine.linear_spine(self, 'left'),
        'right': mspines.Spine.linear_spine(self, 'right')
    }
    return spines

def _set_lim_and_transforms(self):
    ""
    This is called once when the plot is created to set up all the
    transforms for the data, text and grids.
    ""
    rot = 30

    # Get the standard transform setup from the Axes base class
    Axes._set_lim_and_transforms(self)

    # Need to put the skew in the middle, after the scale and limits,
    # but before the transAxes. This way, the skew is done in Axes
    # coordinates thus performing the transform around the proper origin
    # We keep the pre-transAxes transform around for other users, like the
    # spines for finding bounds
    self.transDataToAxes = self.transScale + self.transLimits +
    transforms.Affine2D().skew_deg(rot, 0)

    # Create the full transform from Data to Pixels
    self.transData = self.transDataToAxes + self.transAxes

    # Blended transforms like this need to have the skewing applied using
    # both axes, in axes coords like before.
    self._xaxis_transform = (transforms.blended_transform_factory(
        self.transScale + self.transLimits,
        transforms.IdentityTransform()) +
        transforms.Affine2D().skew_deg(rot, 0)) + self.transAxes

@property
def lower_xlim(self):
    return self.axes.viewLim.intervalx

@property
def upper_xlim(self):
```

89.34. api example code: skwt.py
### Chapter 89. api Examples

```python
pts = [[0., 1.], [1., 1.]]
return self.transDataToAxes.inverted().transform(pts)[0, 0]

# Now register the projection with matplotlib so the user can select it.
register_projection(SkewXAxes)

if __name__ == '__main__':
    # Now make a simple example using the custom projection.
    from matplotlib.ticker import (MultipleLocator, NullFormatter,
                                    ScalarFormatter)
    import matplotlib.pyplot as plt
    from six import StringIO
    import numpy as np

    # Some examples data
    data_txt = '''
978.0 345 7.8 0.8 61 4.16 325 14 282.7 294.6 283.4
971.0 404 7.2 0.2 61 4.01 327 17 282.7 294.2 283.4
946.7 610 5.2 -1.8 61 3.56 335 26 282.8 293.0 283.4
944.0 634 5.0 -2.0 61 3.51 336 27 282.8 292.9 283.4
925.0 798 3.4 -2.6 65 3.43 340 32 282.8 292.7 283.4
911.8 914 2.4 -2.7 69 3.46 345 37 282.9 292.9 283.5
906.0 966 2.0 -2.7 71 3.47 348 39 283.0 293.0 283.6
877.9 1219 0.4 -3.2 77 3.46 350 48 283.9 293.9 284.5
850.0 1478 -0.4 -3.7 84 3.44 353 42 284.8 294.8 285.4
841.0 1563 -1.9 -3.8 87 3.45 358 45 285.0 295.0 285.6
823.0 1736 1.4 -0.7 84 4.44 350 40 290.3 303.3 291.0
813.6 1829 4.5 1.2 80 3.56 358 45 294.5 309.8 295.4
809.0 1875 6.0 2.2 77 3.57 362 47 296.6 313.2 297.6
798.0 1988 7.4 -0.6 57 4.61 340 35 299.2 313.3 300.1
791.0 2061 7.6 -1.4 53 4.39 335 33 300.2 313.6 301.0
783.9 2134 7.0 -1.7 54 4.32 330 31 300.4 313.6 301.2
755.1 2438 4.8 -3.1 57 4.06 325 30 301.2 313.7 301.9
727.3 2743 2.5 -4.4 60 3.81 320 24 301.9 313.8 302.6
700.5 3048 0.2 -5.8 64 3.57 323 28 302.7 313.8 303.3
700.0 3054 0.2 -5.8 64 3.56 326 28 302.7 313.8 303.3
698.0 3077 0.0 -6.0 64 3.52 328 28 302.7 313.7 303.4
687.0 3204 -0.1 -7.1 59 3.28 318 28 304.0 314.3 304.6
648.9 3658 -3.2 -10.9 55 2.59 285 30 305.5 313.8 305.9
631.0 3881 -4.7 -12.7 54 2.29 289 33 306.2 313.6 306.6
600.7 4267 -6.4 -16.7 44 1.73 295 39 308.6 314.3 308.9
592.0 4381 -6.9 -17.9 41 1.59 297 41 309.3 314.6 309.6
577.6 4572 -8.1 -19.6 39 1.41 300 44 310.1 314.9 310.3
555.3 4877 -10.0 -22.3 36 1.16 295 39 311.3 315.3 311.5
536.0 5151 -11.7 -24.7 33 0.97 304 39 312.4 315.8 312.6
533.8 5182 -11.9 -25.0 33 0.95 305 39 312.5 315.8 312.7
500.0 5680 -15.9 -29.9 29 0.64 290 44 313.6 315.9 313.7
472.3 6096 -19.7 -33.4 28 0.49 285 46 314.1 315.8 314.1
453.0 6401 -22.4 -36.0 28 0.39 300 50 314.4 315.8 314.4
400.0 7310 -30.7 -43.7 27 0.20 285 44 315.0 315.8 315.0
399.7 7315 -30.8 -43.8 27 0.20 285 44 315.0 315.8 315.0
```
```
# Parse the data
sound_data = StringIO(data_txt)
p, h, T, Td = np.loadtxt(sound_data, usecols=range(0, 4), unpack=True)

# Create a new figure. The dimensions here give a good aspect ratio
fig = plt.figure(figsize=(6.5875, 6.2125))
anx = fig.add_subplot(111, projection='skewx')

plt.grid(True)

# Plot the data using normal plotting functions, in this case using
# log scaling in Y, as dictated by the typical meteorological plot
ax.semilogy(T, p, color='C3')
```
ax.semilogy(Td, p, color='C2')

# An example of a slanted line at constant X
l = ax.axvline(0, color='C0')

# Disables the log-formatting that comes with semilogy
ax.yaxis.set_major_formatter(ScalarFormatter())
ax.yaxis.set_minor_formatter(NullFormatter())
ax.set_yticks(np.linspace(100, 1000, 10))
ax.set_ylim(1050, 100)
ax.xaxis.set_major_locator(MultipleLocator(10))
ax.set_xlim(-50, 50)

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

89.35 api example code: span_regions.py
Using `span_where`

Illustrate some helper functions for shading regions where a logical mask is `True`

See :meth:`matplotlib.collections.BrokenBarHCollection.span_where`

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.collections as collections

t = np.arange(0.0, 2, 0.01)
s1 = np.sin(2*np.pi*t)
s2 = 1.2*np.sin(4*np.pi*t)

fig, ax = plt.subplots()
ax.set_title('using span_where')
ax.plot(t, s1, color='black')
ax.axhline(0, color='black', lw=2)

collection = collections.BrokenBarHCollection.span_where(
    t, ymin=0, ymax=1, where=s1 > 0, facecolor='green', alpha=0.5)
ax.add_collection(collection)

collection = collections.BrokenBarHCollection.span_where(
    t, ymin=-1, ymax=0, where=s1 < 0, facecolor='red', alpha=0.5)
ax.add_collection(collection)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Demonstrate how to do two plots on the same axes with different left and right scales.

The trick is to use "two different axes" that share the same "x" axis. You can use separate `matplotlib.ticker` formatters and locators as desired since the two axes are independent.

Such axes are generated by calling the `Axes.twinx` method. Likewise, `Axes.twiny` is available to generate axes that share a "y" axis but have different top and bottom scales.

The twinx and twiny methods are also exposed as pyplot functions.

```python
import numpy as np
import matplotlib.pyplot as plt
```
```python
fig, ax1 = plt.subplots()
t = np.arange(0.01, 10.0, 0.01)
s1 = np.exp(t)
ax1.plot(t, s1, 'b-')
ax1.set_xlabel('time (s)')
# Make the y-axis label, ticks and tick labels match the line color.
ax1.set_ylabel('exp', color='b')
ax1.tick_params(y, colors='b')

ax2 = ax1.twinx()
s2 = np.sin(2 * np.pi * t)
ax2.plot(t, s2, 'r-')
ax2.set_ylabel('sin', color='r')
ax2.tick_params(y, colors='r')

fig.tight_layout()
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

89.37 api example code: unicode_minus.py
You can use the proper typesetting Unicode minus (see https://en.wikipedia.org/wiki/Plus_sign#Plus_sign) or the ASCII hyphen for minus, which some people prefer. The matplotlibrc param axes.unicode_minus controls the default behavior.

The default is to use the Unicode minus.

```
import numpy as np
import matplotlib
import matplotlib.pyplot as plt

matplotlib.rcParams['axes.unicode_minus'] = False
fig, ax = plt.subplots()
ax.plot(10*np.random.randn(100), 10*np.random.randn(100), 'o')
ax.set_title('Using hyphen instead of Unicode minus')
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
from __future__ import print_function
import numpy as np
import matplotlib.cbook as cbook
import matplotlib.image as image
import matplotlib.pyplot as plt

datafile = cbook.get_sample_data('logo2.png', asfileobj=False)
print('loading %s' % datafile)
im = image.imread(datafile)
im[:, :, -1] = 0.5  # set the alpha channel

fig, ax = plt.subplots()
ax.plot(np.random.rand(20), '-o', ms=20, lw=2, alpha=0.7, mfc='orange')
ax.grid()
```python
fig.figimage(im, 10, 10, zorder=3)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

### 89.39 api example code: watermark_text.py

```python
# Use a Text as a watermark

import numpy as np
import matplotlib
matplotlib.use('Agg')

import matplotlib.pyplot as plt

fig, ax = plt.subplots()
```
ax.plot(np.random.rand(20), '-o', ms=20, lw=2, alpha=0.7, mfc='orange')
ax.grid()

# position bottom right
fig.text(0.95, 0.05, 'Property of MPL',
    fontsize=50, color='gray',
    ha='right', va='bottom', alpha=0.5)

plt.show()
90.1 axes_grid example code: demo_axes_divider.py
import matplotlib.pyplot as plt

def get_demo_image():
    import numpy as np
    from matplotlib.cbook import get_sample_data
    f = get_sample_data("axes_grid/bivariate_normal.npy", asfileobj=False)
    z = np.load(f)
    # z is a numpy array of 15x15
    return z, (-3, 4, -4, 3)

def demo_simple_image(ax):
    Z, extent = get_demo_image()

    im = ax.imshow(Z, extent=extent, interpolation="nearest")
    cb = plt.colorbar(im)
    plt.setp(cb.ax.get_yticklabels(), visible=False)

def demo_locatable_axes_hard(fig1):

    from mpl_toolkits.axes_grid1 \
    import SubplotDivider, LocatableAxes, Size

    divider = SubplotDivider(fig1, 2, 2, 2, aspect=True)

    # axes for image
    ax = LocatableAxes(fig1, divider.get_position())

    # axes for colorbar
    ax_cb = LocatableAxes(fig1, divider.get_position())

    h = [Size.AxesX(ax),  # main axes
         Size.Fixed(0.05),  # padding, 0.1 inch
         Size.Fixed(0.2),  # colorbar, 0.3 inch
         ]

    v = [Size.AxesY(ax)]

    divider.set_horizontal(h)
    divider.set_vertical(v)

    ax.set_axes_locator(divider.new_locator(nx=0, ny=0))
    ax_cb.set_axes_locator(divider.new_locator(nx=2, ny=0))

    fig1.add_axes(ax)
    fig1.add_axes(ax_cb)

    ax_cb.axis["left"].toggle(all=False)
    ax_cb.axis["right"].toggle(ticks=True)

    Z, extent = get_demo_image()
im = ax.imshow(Z, extent=extent, interpolation="nearest")
plt.colorbar(im, cax=ax_cb)
plt.setp(ax_cb.get_yticklabels(), visible=False)

def demo_locatable_axes_easy(ax):
    from mpl_toolkits.axes_grid1 import make_axes_locatable
    divider = make_axes_locatable(ax)
    ax_cb = divider.new_horizontal(size="5\%", pad=0.05)
    fig1 = ax.get_figure()
    fig1.add_axes(ax_cb)
    Z, extent = get_demo_image()
    im = ax.imshow(Z, extent=extent, interpolation="nearest")
    plt.colorbar(im, cax=ax_cb)
    ax_cb.yaxis.tick_right()
    for tl in ax_cb.get_yticklabels():
        tl.set_visible(False)
    ax_cb.yaxis.tick_right()

def demo_images_side_by_side(ax):
    from mpl_toolkits.axes_grid1 import make_axes_locatable
    divider = make_axes_locatable(ax)
    Z, extent = get_demo_image()
    ax2 = divider.new_horizontal(size="100\%", pad=0.05)
    fig1 = ax.get_figure()
    fig1.add_axes(ax2)
    ax.imshow(Z, extent=extent, interpolation="nearest")
    ax2.imshow(Z, extent=extent, interpolation="nearest")
    for tl in ax2.get_yticklabels():
        tl.set_visible(False)

def demo():
    fig1 = plt.figure(1, (6, 6))
    fig1.clf()

    # PLOT 1
    # simple image & colorbar
    ax = fig1.add_subplot(2, 2, 1)
    demo_simple_image(ax)

    # PLOT 2
    # image and colorbar whose location is adjusted in the drawing time.
# a hard way

demo_locatable_axes_hard(fig1)

# PLOT 3
# image and colorbar whose location is adjusted in the drawing time.
# a easy way

ax = fig1.add_subplot(2, 2, 3)
demo_locatable_axes_easy(ax)

# PLOT 4
# two images side by side with fixed padding.

ax = fig1.add_subplot(2, 2, 4)
demo_images_side_by_side(ax)

plt.draw()
plt.show()

demo()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

90.2 axes_grid example code: demo_axes_grid.py

import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import AxesGrid

def get_demo_image():
    import numpy as np
    from matplotlib.cbook import get_sample_data
    f = get_sample_data("axes_grid/bivariate_normal.npy", asfileobj=False)
    z = np.load(f)
    # z is a numpy array of 15x15
    return z, (-3, 4, -4, 3)
```python
def demo_simple_grid(fig):
    
    A grid of 2x2 images with 0.05 inch pad between images and only
    the lower-left axes is labeled.
    
    grid = AxesGrid(fig, 141,  # similar to subplot(141)
        nrows_ncols=(2, 2),
        axes_pad=0.05,
        label_mode="1",
        )

    Z, extent = get_demo_image()
    for i in range(4):
        im = grid[i].imshow(Z, extent=extent, interpolation="nearest")

        # This only affects axes in first column and second row as share_all =
        # False.
        grid.axes_llc.set_xticks([-2, 0, 2])
        grid.axes_llc.set_yticks([-2, 0, 2])


def demo_grid_with_single_cbar(fig):
    
    A grid of 2x2 images with a single colorbar
    
    grid = AxesGrid(fig, 142,  # similar to subplot(142)
        nrows_ncols=(2, 2),
        axes_pad=0.0,
        share_all=True,
        label_mode="L",
        cbar_location="top",
        cbar_mode="single",
        )

    Z, extent = get_demo_image()
    for i in range(4):
        im = grid[i].imshow(Z, extent=extent, interpolation="nearest")

        #plt.colorbar(im, cax = grid.cbar_axes[0])
        grid.cbar_axes[0].colorbar(im)

        for cax in grid.cbar_axes:
            cax.toggle_label(False)

        # This affects all axes as share_all = True.
        grid.axes_llc.set_xticks([-2, 0, 2])
        grid.axes_llc.set_yticks([-2, 0, 2])


def demo_grid_with_each_cbar(fig):
    
    A grid of 2x2 images. Each image has its own colorbar.
```
grid = AxesGrid(fig, 143, # similar to subplot(143)
    nrows_ncols=(2, 2),
    axes_pad=0.1,
    label_mode="1",
    share_all=True,
    cbar_location="top",
    cbar_mode="each",
    cbar_size="7%",
    cbar_pad="2%",
)
Z, extent = get_demo_image()
for i in range(4):
    im = grid[i].imshow(Z, extent=extent, interpolation="nearest")
    grid.cbar_axes[i].colorbar(im)

for cax in grid.cbar_axes:
    cax.toggle_label(False)

# This affects all axes because we set share_all = True.
grid.axes_llc.set_xticks([-2, 0, 2])
grid.axes_llc.set_yticks([-2, 0, 2])

def demo_grid_with_each_cbar_labelled(fig):
    
    A grid of 2x2 images. Each image has its own colorbar.

    grid = AxesGrid(fig, 144, # similar to subplot(144)
        nrows_ncols=(2, 2),
        axes_pad=(0.45, 0.15),
        label_mode="1",
        share_all=True,
        cbar_location="right",
        cbar_mode="each",
        cbar_size="7%",
        cbar_pad="2%",
    )
Z, extent = get_demo_image()

# Use a different colorbar range every time
limits = ((0, 1), (-2, 2), (-1.7, 1.4), (-1.5, 1))
for i in range(4):
    im = grid[i].imshow(Z, extent=extent, interpolation="nearest",
                        vmin=limits[i][0], vmax=limits[i][1])
    grid.cbar_axes[i].colorbar(im)

for i, cax in enumerate(grid.cbar_axes):
    cax.set_yticks((limits[i][0], limits[i][1]))

# This affects all axes because we set share_all = True.
grid.axes_llc.set_xticks([-2, 0, 2])
grid.axes_llc.set_yticks([-2, 0, 2])
if 1:
    F = plt.figure(1, (10.5, 2.5))
    F.subplots_adjust(left=0.05, right=0.95)
    demo_simple_grid(F)
    demo_grid_with_single_cbar(F)
    demo_grid_with_each_cbar(F)
    demo_grid_with_each_cbar_labelled(F)
    plt.draw()
    plt.show()
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import ImageGrid
import numpy as np

def get_demo_image():
    from matplotlib.cbook import get_sample_data
    f = get_sample_data("axes_grid/bivariate_normal.npy", asfileobj=False)
    z = np.load(f)
    # z is a numpy array of 15x15
    return z, (-3, 4, -4, 3)

90.3 axes_grid example code: demo_axes_grid2.py
def add_inner_title(ax, title, loc, size=None, **kwargs):
    from matplotlib.offsetbox import AnchoredText
    from matplotlib.pathEffects import withStroke

    if size is None:
        size = dict(size=plt.rcParams['legend.fontsize'])
    at = AnchoredText(title, loc=loc, prop=size,
                      pad=0., borderpad=0.5,
                      frameon=False, **kwargs)
    ax.add_artist(at)
    at.txt._text.set_path_effects([withStroke(foreground="w", linewidth=3)])
    return at

if 1:
    F = plt.figure(1, (6, 6))
    F.clf()

    # prepare images
    Z, extent = get_demo_image()
    ZS = [Z[i::3, :] for i in range(3)]
    extent = extent[0], extent[1]/3., extent[2], extent[3]

    # demo 1 : colorbar at each axes
    grid = ImageGrid(F, 211,  # similar to subplot(111)
                     nrows_ncols=(1, 3),
                     direction="row",
                     axes_pad=0.05,
                     add_all=True,
                     label_mode="1",
                     share_all=True,
                     cbar_location="top",
                     cbar_mode="each",
                     cbar_size="7%",
                     cbar_pad="1%",
                     )

    for ax, z in zip(grid, ZS):
        im = ax.imshow(z, origin="lower", extent=extent, interpolation="nearest")
        ax.cax.colorbar(im)

    for ax, im_title in zip(grid, ["Image 1", "Image 2", "Image 3"]):
        t = add_inner_title(ax, im_title, loc=3)
        t.patch.set_alpha(0.5)

    for ax, z in zip(grid, ZS):
        ax.cax.toggle_label(True)
        axis = ax.cax.axis[ax.cax.orientation]
        axis.label.set_text("counts s^{-1}\")
        axis.label.set_size(10)
        axis.major_ticklabels.set_size(6)

    # changing the colorbar ticks
grid[1].cax.set_xticks([-1, 0, 1])
grid[2].cax.set_xticks([-1, 0, 1])

grid[0].set_xticks([-2, 0])
grid[0].set_yticks([-2, 0, 2])

# demo 2 : shared colorbar

grid2 = ImageGrid(F, 212,
nrows_ncols=(1, 3),
direction="row",
axes_pad=0.05,
add_all=True,
label_mode="1",
share_all=True,
cbar_location="right",
cbar_mode="single",
cbar_size="10%",
cbar_pad=0.05,
)

grid2[0].set_xlabel("X")
grid2[0].set_ylabel("Y")

vmax, vmin = np.max(ZS), np.min(ZS)

import matplotlib.colors
	norm = matplotlib.colors.Normalize(vmax=vmax, vmin=vmin)

for ax, z in zip(grid2, ZS):
    im = ax.imshow(z, norm=norm,
                   origin="lower", extent=extent,
                   interpolation="nearest")

# With cbar_mode="single", cax attribute of all axes are identical.
    ax.cax.colorbar(im)
    ax.cax.toggle_label(True)

for ax, im_title in zip(grid2, ["(a)", "(b)", "(c)"]):
    t = add_inner_title(ax, im_title, loc=2)
    t.patch.set_ec("none")
    t.patch.set_alpha(0.5)

grid2[0].set_xticks([-2, 0])
grid2[0].set_yticks([-2, 0, 2])

plt.draw()
plt.show()
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1.axes_divider import HBoxDivider
import mpl_toolkits.axes_grid1.axes_size as Size

def make_heights_equal(fig, rect, ax1, ax2, pad):
    # pad in inches

    h1, v1 = Size.AxesX(ax1), Size.AxesY(ax1)
    h2, v2 = Size.AxesX(ax2), Size.AxesY(ax2)

    pad_v = Size.Scaled(1)
    pad_h = Size.Fixed(pad)

    my_divider = HBoxDivider(fig, rect,
                              horizontal=[h1, pad_h, h2],
                              vertical=[v1, pad_v, v2])

    ax1.set_axes_locator(my_divider.new_locator(0))
    ax2.set_axes_locator(my_divider.new_locator(2))
if __name__ == "__main__":
    arr1 = np.arange(20).reshape((4, 5))
    arr2 = np.arange(20).reshape((5, 4))

    fig, (ax1, ax2) = plt.subplots(1, 2)
    ax1.imshow(arr1, interpolation="nearest")
    ax2.imshow(arr2, interpolation="nearest")

    rect = 111  # subplot param for combined axes
    make_heights_equal(fig, rect, ax1, ax2, pad=0.5)  # pad in inches

    for ax in [ax1, ax2]:
        ax.locator_params(nbins=4)

    # annotate
    ax3 = plt.axes([0.5, 0.5, 0.001, 0.001], frameon=False)
    ax3.xaxis.set_visible(False)
    ax3.yaxis.set_visible(False)
    ax3.annotate("Location of two axes are adjusted\n" + "so that they have equal heights\n" + "while maintaining their aspect ratios", (0.5, 0.5),
                 xycoords="axes fraction", va="center", ha="center",
                 bbox=dict(boxstyle="round, pad=1", fc="w"))

    plt.show()
90.5 axes_grid example code: demo_axes_rgb.py
import numpy as np
import matplotlib.pyplot as plt

from mpl_toolkits.axes_grid1.axes_rgb import make_rgb_axes, RGBAxes

def get_demo_image():
    from matplotlib.cbook import get_sample_data
    f = get_sample_data("axes_grid/bivariate_normal.npy", asfileobj=False)
    z = np.load(f)
    # z is a numpy array of 15x15
    return z, (-3, 4, -4, 3)

def get_rgb():
    Z, extent = get_demo_image()
    Z[Z < 0] = 0.
    Z = Z/Z.max()

    R = Z[:13, :13]
    G = Z[2:, 2:]
    B = Z[:13, 2:]

    return R, G, B
def make_cube(r, g, b):
    ny, nx = r.shape
    R = np.zeros([ny, nx, 3], dtype="d")
    R[:, :, 0] = r
    G = np.zeros_like(R)
    G[:, :, 1] = g
    B = np.zeros_like(R)
    B[:, :, 2] = b
    RGB = R + G + B
    return R, G, B, RGB

def demo_rgb():
    fig, ax = plt.subplots()
    ax_r, ax_g, ax_b = make_rgb_axes(ax, pad=0.02)
    # fig.add_axes(ax_r)
    # fig.add_axes(ax_g)
    # fig.add_axes(ax_b)
    r, g, b = get_rgb()
    im_r, im_g, im_b, im_rgb = make_cube(r, g, b)
    kwargs = dict(origin="lower", interpolation="nearest")
    ax.imshow(im_rgb, **kwargs)
    ax_r.imshow(im_r, **kwargs)
    ax_g.imshow(im_g, **kwargs)
    ax_b.imshow(im_b, **kwargs)

def demo_rgb2():
    fig = plt.figure(2)
    ax = RGBAxes(fig, [0.1, 0.1, 0.8, 0.8], pad=0.0)
    # fig.add_axes(ax)
    # ax.add_RGB_to_figure()
    r, g, b = get_rgb()
    kwargs = dict(origin="lower", interpolation="nearest")
    ax.imshow_rgb(r, g, b, **kwargs)
    ax.RGB.set_xlim(0., 9.5)
    ax.RGB.set_ylim(0.9, 10.6)

    for ax1 in [ax.RGB, ax.R, ax.G, ax.B]:
        for spl in ax1.spines.values():
            spl.set_color("w")
        for tick in ax1.xaxis.get_major_ticks() + ax1.yaxis.get_major_ticks():
            tick.tick1line.set_mec("w")
            tick.tick2line.set_mec("w")
    return ax
demo_rgb()
ad = demo_rgb2()

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

90.6 axes_grid example code: demo_axisline_style.py

```python
from mpl_toolkits.axes_grid.axislines import SubplotZero
import matplotlib.pyplot as plt
import numpy as np

if 1:
    fig = plt.figure(1)
    ax = SubplotZero(fig, 111)
    fig.add_subplot(ax)

    for direction in ["xzero", "yzero"]:
        ax.axis[direction].set_axisline_style("-|>")
        ax.axis[direction].set_visible(True)
```

Chapter 90. axes_grid Examples
```python
for direction in ["left", "right", "bottom", "top"]:
    ax.axis[direction].set_visible(False)

x = np.linspace(-0.5, 1., 100)
ax.plot(x, np.sin(x*np.pi))
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

### 90.7 axes_grid example code: demo_colorbar_with_inset_locator.py

```python
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1.inset_locator import inset_axes

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=[6, 3])
axins1 = inset_axes(ax1,
    width="50%", # width = 10% of parent_bbox width
    height="5%", # height : 50%
    loc=1)

im1 = ax1.imshow([[1, 2], [2, 3]])
plt.colorbar(im1, cax=axins1, orientation="horizontal", ticks=[1, 2, 3])
axins1.xaxis.set_ticks_position("bottom")

axins = inset_axes(ax2,
    width="5%", # width = 10% of parent_bbox width
    height="50%", # height : 50%
    loc=1)

im1 = ax2.imshow([[1, 2], [2, 3]])
plt.colorbar(im1, cax=axins, orientation="vertical", ticks=[1, 2, 3])
axins.yaxis.set_ticks_position("right")
```

---

90.7. axes_grid example code: demo_colorbar_with_inset_locator.py 2139
Custom grid and ticklines.

This example demonstrates how to use GridHelperCurveLinear to define custom grids and ticklines by applying a transformation on the grid. This can be used, as showcase on the second plot, to create polar projections in a rectangular box.
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cbook as cbook

from mpl_toolkits.axisartist import Subplot
from mpl_toolkits.axisartist import SubplotHost, ParasiteAxesAuxTrans
from mpl_toolkits.axisartist.grid_helper_curvelinear import GridHelperCurveLinear

def curvelinear_test1(fig):
    
    grid for custom transform.
    
    def tr(x, y):
        x, y = np.asarray(x), np.asarray(y)
        return x, y - x

    def inv_tr(x, y):
        x, y = np.asarray(x), np.asarray(y)
        return x, y + x

    grid_helper = GridHelperCurveLinear((tr, inv_tr))

    ax1 = Subplot(fig, 1, 2, 1, grid_helper=grid_helper)
    # ax1 will have a ticks and gridlines defined by the given
    # transform (+ transData of the Axes). Note that the transform of
    # the Axes itself (i.e., transData) is not affected by the given
    # transform.

    fig.add_subplot(ax1)

    xx, yy = tr([3, 6], [5.0, 10.0])
    ax1.plot(xx, yy, linewidth=2.0)

    ax1.set_aspect(1.)
    ax1.set_xlim(0, 10.)
    ax1.set_ylim(0, 10.)

    ax1.axis['t'] = ax1.new_floating_axis(0, 3.)
    ax1.axis['t2'] = ax1.new_floating_axis(1, 7.)
    ax1.grid(True, zorder=0)

import mpl_toolkits.axisartist.angle_helper as angle_helper
from matplotlib.projections import PolarAxes
from matplotlib.transforms import Affine2D
```python
def curvelinear_test2(fig):
    
    polar projection, but in a rectangular box.

    # PolarAxes.PolarTransform takes radian. However, we want our coordinate
    # system in degree
    tr = Affine2D().scale(np.pi/180., 1.) + PolarAxes.PolarTransform()

    # polar projection, which involves cycle, and also has limits in
    # its coordinates, needs a special method to find the extremes
    # (min, max of the coordinate within the view).
    # 20, 20 : number of sampling points along x, y direction
    extreme_finder = angle_helper.ExtremeFinderCycle(20, 20,
        lon_cycle=360,
        lat_cycle=None,
        lon_minmax=None,
        lat_minmax=(0, np.inf),
    )

    grid_locator1 = angle_helper.LocatorDMS(12)
    # Find a grid values appropriate for the coordinate (degree,
    # minute, second).
    tick_formatter1 = angle_helper.FormatterDMS()
    # And also uses an appropriate formatter. Note that, the
    # acceptable Locator and Formatter class is a bit different than
    # that of mpl's, and you cannot directly use mpl's Locator and
    # Formatter here (but may be possible in the future).
    grid_helper = GridHelperCurveLinear(tr,
        extreme_finder=extreme_finder,
        grid_locator1=grid_locator1,
        tick_formatter1=tick_formatter1
    )

    ax1 = SubplotHost(fig, 1, 2, 2, grid_helper=grid_helper)
    # make ticklabels of right and top axis visible.
    ax1.axis["right"].major_ticklabels.set_visible(True)
    ax1.axis["top"].major_ticklabels.set_visible(True)

    # let right axis shows ticklabels for 1st coordinate (angle)
    ax1.axis["right"].get_helper().nth_coord_ticks = 0
    # let bottom axis shows ticklabels for 2nd coordinate (radius)
    ax1.axis["bottom"].get_helper().nth_coord_ticks = 1

    fig.add_subplot(ax1)

    # A parasite axes with given transform
```
```python
ax2 = ParasiteAxesAuxTrans(ax1, tr, "equal")
# note that ax2.transData == tr + ax1.transData
# Anything you draw in ax2 will match the ticks and grids of ax1.
ax1.parasites.append(ax2)
intp = cbook.simple_linear_interpolation
ax2.plot(intp(np.array([0, 30]), 50),
         intp(np.array([10., 10.]), 50),
         linewidth=2.0)

ax1.set_aspect(1.)
ax1.set_xlim(-5, 12)
ax1.set_ylim(-5, 10)

ax1.grid(True, zorder=0)

if 1:
    fig = plt.figure(1, figsize=(7, 4))
    fig.clf()

    curvelinear_test1(fig)
    curvelinear_test2(fig)

    plt.draw()
    plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Custom grid and ticklines.

This example demonstrates how to use GridHelperCurveLinear to define custom grids and ticklines by applying a transformation on the grid. As showcase on the plot, a 5x5 matrix is displayed on the axes.

```python
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid.grid_helper_curvelinear import GridHelperCurveLinear
from mpl_toolkits.axes_grid.axislines import Subplot
import mpl_toolkits.axes_grid.angle_helper as angle_helper

def curvelinear_test1(fig):
    
    def tr(x, y):
        sgn = np.sign(x)
        x, y = np.abs(np.asarray(x)), np.asarray(y)
        return sgn*x**.5, y

    grid for custom transform.

    import numpy as np
    import matplotlib.pyplot as plt
    from mpl_toolkits.axes_grid.grid_helper_curvelinear import GridHelperCurveLinear
    from mpl_toolkits.axes_grid.axislines import Subplot
    import mpl_toolkits.axes_grid.angle_helper as angle_helper

    def curvelinear_test1(fig):
        
        def tr(x, y):
            sgn = np.sign(x)
            x, y = np.abs(np.asarray(x)), np.asarray(y)
            return sgn*x**.5, y

        grid for custom transform.

        import numpy as np
        import matplotlib.pyplot as plt
        from mpl_toolkits.axes_grid.grid_helper_curvelinear import GridHelperCurveLinear
        from mpl_toolkits.axes_grid.axislines import Subplot
        import mpl_toolkits.axes_grid.angle_helper as angle_helper
```
```python
def inv_tr(x, y):
    sgn = np.sign(x)
    x, y = np.asarray(x), np.asarray(y)
    return sgn*x**2, y

extreme_finder = angle_helper.ExtremeFinderCycle(20, 20,
    lon_cycle=None,
    lat_cycle=None,
    # (0, np.inf),
    lon_minmax=None,
    lat_minmax=None,
)

grid_helper = GridHelperCurveLinear((tr, inv_tr),
    extreme_finder=extreme_finder)

ax1 = Subplot(fig, 111, grid_helper=grid_helper)
    # ax1 will have a ticks and gridlines defined by the given
    # transform (+ transData of the Axes). Note that the transform of
    # the Axes itself (i.e., transData) is not affected by the given
    # transform.
fig.add_subplot(ax1)

ax1.imshow(np.arange(25).reshape(5, 5),
    vmax=50, cmap=plt.cm.gray_r,
    interpolation="nearest",
    origin="lower")

    # tick density
grid_helper.grid_finder.grid_locator1._nbins = 6
grid_helper.grid_finder.grid_locator2._nbins = 6

if 1:
    fig = plt.figure(1, figsize=(7, 4))
    fig.clf()

curvelinear_test1(fig)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import AxesGrid

def get_demo_image():
    import numpy as np
    from matplotlib.cbook import get_sample_data
    f = get_sample_data("axes_grid/bivariate_normal.npy", asfileobj=False)
    z = np.load(f)
    # z is a numpy array of 15x15
    return z, (-3, 4, -4, 3)

def demo_bottom_cbar(fig):
    """
    A grid of 2x2 images with a colorbar for each column.
    """
    grid = AxesGrid(fig, 121,  # similar to subplot(132)
                    nrows_ncols=(2, 2),
                    axes_pad=0.10,
                    share_all=True,
                    label_mode="1",
                    cbar_location="bottom",
                    cbar_mode="edge",
                    cbar_pad=0.25,
                    cbar_size="15%",
                    direction="column"
                   )

    Z, extent = get_demo_image()
    cmaps = [plt.get_cmap("autumn"), plt.get_cmap("summer")]
    for i in range(4):
        im = grid[i].imshow(Z, extent=extent, interpolation="nearest",
                            cmap=cmaps[i//2])
        if i % 2:
cbar = grid.cbar_axes[i//2].colorbar(im)

for cax in grid.cbar_axes:
    cax.toggle_label(True)
    cax.axis[cax.orientation].set_label("Bar")

# This affects all axes as share_all = True.
grid.axes_llc.set_xticks([-2, 0, 2])
grid.axes_llc.set_yticks([-2, 0, 2])

def demo_right_cbar(fig):
    """
    A grid of 2x2 images. Each row has its own colorbar.
    """

grid = AxesGrid(F, 122, # similar to subplot(122)
    nrows_ncols=(2, 2),
    axes_pad=0.10,
    label_mode="1",
    share_all=True,
    cbar_location="right",
    cbar_mode="edge",
    cbar_size="7%",
    cbar_pad="2%",
)
Z, extent = get_demo_image()
cmaps = [plt.get_cmap("spring"), plt.get_cmap("winter")]
for i in range(4):
    im = grid[i].imshow(Z, extent=extent, interpolation="nearest",
                         cmap=cmaps[i//2])
    if i % 2:
        grid.cbar_axes[i//2].colorbar(im)

for cax in grid.cbar_axes:
    cax.toggle_label(True)
    cax.axis[cax.orientation].set_label('Foo')

# This affects all axes because we set share_all = True.
grid.axes_llc.set_xticks([-2, 0, 2])
grid.axes_llc.set_yticks([-2, 0, 2])

if 1:
    F = plt.figure(1, (5.5, 2.5))

    F.subplots_adjust(left=0.05, right=0.93)

demo_bottom_cbar(F)
demo_right_cbar(F)

plt.draw()
plt.show()
Demo of the floating axes.

This demo shows features of functions in floating_axes:
* Using scatter function and bar function with changing the shape of the plot.
* Using GridHelperCurveLinear to rotate the plot and set the boundary of the plot.
* Using FloatingSubplot to create a subplot using the return value from GridHelperCurveLinear.
* Making sector plot by adding more features to GridHelperCurveLinear.

```python
from matplotlib.transforms import Affine2D
import mpl_toolkits.axisartist.floating_axes as floating_axes
import numpy as np
import mpl_toolkits.axisartist.angle_helper as angle_helper
from matplotlib.projections import PolarAxes
from mpl_toolkits.axisartist.grid_finder import (FixedLocator, MaxNLocator,
                                                  DictFormatter)
import matplotlib.pyplot as plt

def setup_axes1(fig, rect):
    """
    A simple one.
    """
    tr = Affine2D().scale(2, 1).rotate_deg(30)
```
grid_helper = floating_axes.GridHelperCurveLinear(
    tr, extremes=(-0.5, 3.5, 0, 4))

ax1 = floating_axes.FloatingSubplot(fig, rect, grid_helper=grid_helper)
fig.add_subplot(ax1)

aux_ax = ax1.get_aux_axes(tr)

grid_helper.grid_finder.grid_locator1._nbins = 4
grid_helper.grid_finder.grid_locator2._nbins = 4

return ax1, aux_ax

def setup_axes2(fig, rect):
    ""
    With custom locator and formatter.
    Note that the extreme values are swapped.
    ""
    tr = PolarAxes.PolarTransform()

    pi = np.pi
    angle_ticks = [(0, r"$0$"),
                   (.25*pi, r"$\frac{1}{4}\pi$"),
                   (.5*pi, r"$\frac{1}{2}\pi$")]

    grid_locator1 = FixedLocator([v for v, s in angle_ticks])
    tick_formatter1 = DictFormatter(dict(angle_ticks))

    grid_locator2 = MaxNLocator(2)

    grid_helper = floating_axes.GridHelperCurveLinear(
        tr, extremes=(.5*pi, 0, 2, 1),
        grid_locator1=grid_locator1, grid_locator2=grid_locator2,
        tick_formatter1=tick_formatter1, tick_formatter2=None)

    ax1 = floating_axes.FloatingSubplot(fig, rect, grid_helper=grid_helper)
    fig.add_subplot(ax1)

    # create a parasite axes whose transData in RA, cz
    aux_ax = ax1.get_aux_axes(tr)

    aux_ax.patch = ax1.patch # for aux_ax to have a clip path as in ax
    ax1.patch.zorder = 0.9 # but this has a side effect that the patch is
    # drawn twice, and possibly over some other
    # artists. So, we decrease the zorder a bit to
    # prevent this.

    return ax1, aux_ax
def setup_axes3(fig, rect):
    """
    Sometimes, things like axis_direction need to be adjusted.
    """

    # rotate a bit for better orientation
    tr_rotate = Affine2D().translate(-95, 0)

    # scale degree to radians
    tr_scale = Affine2D().scale(np.pi/180., 1.)

    tr = tr_rotate + tr_scale + PolarAxes.PolarTransform()

    grid_locator1 = angle_helper.LocatorHMS(4)
    tick_formatter1 = angle_helper.FormatterHMS()

    grid_locator2 = MaxNLocator(3)

    ra0, ra1 = 8.*15, 14.*15
    cz0, cz1 = 0, 14000

    grid_helper = floating_axes.GridHelperCurveLinear(
        tr, extremes=(ra0, ra1, cz0, cz1),
        grid_locator1=grid_locator1,
        grid_locator2=grid_locator2,
        tick_formatter1=tick_formatter1,
        tick_formatter2=None)

    ax1 = floating_axes.FloatingSubplot(fig, rect, grid_helper=grid_helper)
    fig.add_subplot(ax1)

    # adjust axis
    ax1.axis["left"].set_axis_direction("bottom")
    ax1.axis["right"].set_axis_direction("top")

    ax1.axis["bottom"].set_visible(False)
    ax1.axis["top"].set_axis_direction("bottom")
    ax1.axis["top"].toggle(ticklabels=True, label=True)
    ax1.axis["top"].major_ticklabels.set_axis_direction("top")
    ax1.axis["top"].label.set_axis_direction("top")

    ax1.axis["left"].label.set_text(r"cz [km$^{-1}$]")
    ax1.axis["top"].label.set_text(r"$\alpha_{1950}$")

    # create a parasite axes whose transData in RA, cz
    aux_ax = ax1.get_aux_axes(tr)

    aux_ax.patch = ax1.patch  # for aux_ax to have a clip path as in ax
    aux_ax.patch.zorder = 0.9  # but this has a side effect that the patch is
    # drawn twice, and possibly over some other
    # artists. So, we decrease the zorder a bit to
    # prevent this.

    return ax1, aux_ax
fig = plt.figure(1, figsize=(8, 4))
fig.subplots_adjust(wspace=0.3, left=0.05, right=0.95)

ax1, aux_ax1 = setup_axes1(fig, 131)
aux_ax1.bar([0, 1, 2, 3], [3, 2, 1, 3])

ax2, aux_ax2 = setup_axes2(fig, 132)
theta = np.random.rand(10)*.5*np.pi
radius = np.random.rand(10) + 1.
aux_ax2.scatter(theta, radius)

ax3, aux_ax3 = setup_axes3(fig, 133)

theta = (8 + np.random.rand(10)*(14 - 8))*15.  # in degrees
radius = np.random.rand(10)*14000.
aux_ax3.scatter(theta, radius)

plt.show()
90.12 axes_grid example code: demo_floating_axis.py

Axis within rectangular frame

The following code demonstrates how to put a floating polar curve within a rectangular box. In order to get a better sense of polar curves, please look at demo_curvelinear_grid.py.

```python
import numpy as np
import matplotlib.pyplot as plt
import mpl_toolkits.axisartist.angle_helper as angle_helper
from matplotlib.projections import PolarAxes
from matplotlib.transforms import Affine2D
from mpl_toolkits.axisartist import SubplotHost
from mpl_toolkits.axisartist import GridHelperCurveLinear

def curvelinear_test2(fig):
    
    polar projection, but in a rectangular box.
```
```
global ax1
# see demo_curvelinear_grid.py for details
tr = Affine2D().scale(np.pi / 180., 1.) + PolarAxes.PolarTransform()

extreme_finder = angle_helper.ExtremeFinderCycle(20,
20,
lon_cycle=360,
lat_cycle=None,
lon_minmax=None,
lat_minmax=(0,
np.inf),
)

grid_locator1 = angle_helper.LocatorDMS(12)
tick_formatter1 = angle_helper.FormatterDMS()

grid_helper = GridHelperCurveLinear(tr,
    extreme_finder=extreme_finder,
    grid_locator1=grid_locator1,
    tick_formatter1=tick_formatter1
)

ax1 = SubplotHost(fig, 1, 1, 1, grid_helper=grid_helper)

fig.add_subplot(ax1)

# Now creates floating axis

# floating axis whose first coordinate (theta) is fixed at 60
ax1.axis["lat"] = axis = ax1.new_floating_axis(0, 60)
axis.label.set_text(r"$\theta = 60^\circ$"
axis.label.set_visible(True)

# floating axis whose second coordinate (r) is fixed at 6
ax1.axis["lon"] = axis = ax1.new_floating_axis(1, 6)
axis.label.set_text(r"$r = 6$"

ax1.set_aspect(1.)
ax1.set_xlim(-5, 12)
ax1.set_ylim(-5, 10)

ax1.grid(True)

fig = plt.figure(1, figsize=(5, 5))
fig.clf()

curvelinear_test2(fig)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
import matplotlib.pyplot as plt

from mpl_toolkits.axes_grid1 import ImageGrid
fig = plt.figure(1)

grid1 = ImageGrid(fig, 121, (2, 2), axes_pad=0.1,
    aspect=True, share_all=True)

for i in [0, 1]:
    grid1[i].set_aspect(2)

grid2 = ImageGrid(fig, 122, (2, 2), axes_pad=0.1,
    aspect=True, share_all=True)

for i in [1, 3]:
    grid2[i].set_aspect(2)

plt.show()
Parasite axis demo

The following code is an example of a parasite axis. It aims to show a user how to plot multiple different values onto one single plot. Notice how in this example, par1 and par2 are both calling twinx meaning both are tied directly to the x-axis. From there, each of those two axis can behave separately from the each other, meaning they can take on separate values from themselves as well as the x-axis.

```python
from mpl_toolkits.axes_grid1 import host_subplot
import mpl_toolkits.axisartist as AA
import matplotlib.pyplot as plt

host = host_subplot(111, axes_class=AA.Axes)
plt.subplots_adjust(right=0.75)

par1 = host.twinx()
par2 = host.twinx()

offset = 60
new_fixed_axis = par2.get_grid_helper().new_fixed_axis
```

```python
offset = 60
new_fixed_axis = par2.get_grid_helper().new_fixed_axis
```
par2.axis["right"] = new_fixed_axis(loc="right",
    axes=par2,
    offset=(offset, 0))

par2.axis["right"].toggle(all=True)

host.set_xlim(0, 2)
host.set_ylim(0, 2)

host.set_xlabel("Distance")
host.set_ylabel("Density")
par1.set_ylabel("Temperature")
par2.set_ylabel("Velocity")

p1, = host.plot([0, 1, 2], [0, 1, 2], label="Density")
p2, = par1.plot([0, 1, 2], [0, 3, 2], label="Temperature")
p3, = par2.plot([0, 1, 2], [50, 30, 15], label="Velocity")

par1.set_ylim(0, 4)
par2.set_ylim(1, 65)

host.legend()

host.axis["left"].label.set_color(p1.get_color())
par1.axis["right"].label.set_color(p2.get_color())
par2.axis["right"].label.set_color(p3.get_color())

plt.draw()
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
import matplotlib.pyplot as plt

from mpl_toolkits.axes_grid1.inset_locator import inset_axes, zoomed_inset_axes
from mpl_toolkits.axes_grid1.anchored_artists import AnchoredSizeBar

def add_sizebar(ax, size):
    asb = AnchoredSizeBar(ax.transData,
                          size,
                          str(size),
                          loc=8,
                          pad=0.1, borderpad=0.5, sep=5,
                          frameon=False)

    ax.add_artist(asb)

fig, (ax, ax2) = plt.subplots(1, 2, figsize=[5.5, 3])

# first subplot
ax.set_aspect(1.)

axins = inset_axes(ax,
                    width="30%",  # width = 30% of parent_bbox
                    height=1.,  # height : 1 inch
                    loc=3)

plt.xticks(visible=False)
plt.yticks(visible=False)

# second subplot
ax2.set_aspect(1.)

axins = zoomed_inset_axes(ax2, 0.5, loc=1)  # zoom = 0.5
# fix the number of ticks on the inset axes
axins.yaxis.get_major_locator().set_params(nbins=7)
axins.xaxis.get_major_locator().set_params(nbins=7)

plt.xticks(visible=False)
plt.yticks(visible=False)

add_sizebar(ax2, 0.5)
add_sizebar(axins, 0.5)

plt.draw()
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

90.16 axes_grid example code: inset_locator_demo2.py

```python
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1.inset_locator import zoomed_inset_axes
from mpl_toolkits.axes_grid1.inset_locator import mark_inset
```
import numpy as np

def get_demo_image():
    from matplotlib.cbook import get_sample_data
    import numpy as np
    f = get_sample_data("axes_grid/bivariate_normal.npy", asfileobj=False)
    z = np.load(f)
    # z is a numpy array of 15x15
    return z, (-3, 4, -4, 3)

fig, ax = plt.subplots(figsize=[5, 4])

# prepare the demo image
Z, extent = get_demo_image()
Z2 = np.zeros([150, 150], dtype="d")
ny, nx = Z.shape
Z2[30:30 + ny, 30:30 + nx] = Z

# extent = [-3, 4, -4, 3]
ax.imshow(Z2, extent=extent, interpolation="nearest",
         origin="lower")

axins = zoomed_inset_axes(ax, 6, loc=1)  # zoom = 6
axins.imshow(Z2, extent=extent, interpolation="nearest",
             origin="lower")

# sub region of the original image
x1, x2, y1, y2 = -1.5, -.9, -2.5, -1.9
axins.set_xlim(x1, x2)
axins.set_ylim(y1, y2)

# fix the number of ticks on the inset axes
axins.yaxis.get_major_locator().set_params(nbins=7)
axins.xaxis.get_major_locator().set_params(nbins=7)

plt.xticks(visible=False)
plt.yticks(visible=False)

# draw a bbox of the region of the inset axes in the parent axes and
# connecting lines between the bbox and the inset axes area
mark_inset(ax, axins, loc1=2, loc2=4, fc="none", ec="0.5")

plt.draw()
plt.show()
90.17 axes_grid example code: make_room_for_ylabel_using_axesgrid.py
from mpl_toolkits.axes_grid1 import make_axes_locatable
from mpl_toolkits.axes_grid1.axes_divider import make_axes_area_auto_adjustable

if __name__ == "__main__":
    import matplotlib.pyplot as plt

    def ex1():
        plt.figure(1)
        ax = plt.axes([0, 0, 1, 1])
        #ax = plt.subplot(111)
        ax.set_yticks([0.5])
        ax.set_yticklabels(['very long label'])
        make_axes_area_auto_adjustable(ax)

    def ex2():
        plt.figure(2)
        ax1 = plt.axes([0, 0, 1, 0.5])
        ax2 = plt.axes([0, 0.5, 1, 0.5])
        ax1.set_yticks([0.5])
        ax1.set_yticklabels(['very long label'])
ax1.set_ylabel("Y label")

ax2.set_title("Title")

make_axes_area_auto_adjustable(ax1, pad=0.1, use_axes=[ax1, ax2])
make_axes_area_auto_adjustable(ax2, pad=0.1, use_axes=[ax1, ax2])

def ex3():
    fig = plt.figure(3)
    ax1 = plt.axes([0, 0, 1, 1])
    divider = make_axes_locatable(ax1)

    ax2 = divider.new_horizontal("100\%", pad=0.3, sharey=ax1)
    ax2.tick_params(labelleft="off")
    fig.add_axes(ax2)

    divider.add_auto_adjustable_area(use_axes=[ax1], pad=0.1,
                                     adjust_dirs="left")
    divider.add_auto_adjustable_area(use_axes=[ax2], pad=0.1,
                                     adjust_dirs="right")
    divider.add_auto_adjustable_area(use_axes=[ax1, ax2], pad=0.1,
                                     adjust_dirs="top", "bottom")

    ax1.set_yticks([0.5])
    ax1.set_yticklabels(["very long label"])

    ax2.set_title("Title")
    ax2.set_xlabel("X - Label")

ex1()
ex2()
ex3()
plt.show()
Proper Motion [′′/yr]

Linear velocity at 2.3 kpc [km/s]

```
import matplotlib.transforms as mtransforms
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1.parasite_axes import SubplotHost

obs = [['01_S1', 3.88, 0.14, 1970, 63],
        ['01_S4', 5.6, 0.82, 1622, 150],
        ['02_S1', 2.4, 0.54, 1570, 40],
        ['03_S1', 4.1, 0.62, 2380, 170]]

fig = plt.figure()

ax_kms = SubplotHost(fig, 1, 1, 1, aspect=1.)

# angular proper motion(″/yr) to linear velocity(km/s) at distance=2.3kpc
pm_to_kms = 1./206265.*2300*3.085e18/3.15e7/1.e5

aux_trans = mtransforms.Affine2D().scale(pm_to_kms, 1.)
ax_pm = ax_kms.twin(aux_trans)
ax_pm.set_viewlim_mode("transform")

fig.add_subplot(ax_kms)
```
for n, ds, dse, w, we in obs:
    time = ((2007 + (10. + 4/30.)/12) - 1988.5)
    v = ds / time * pm_to_kms
    ve = dse / time * pm_to_kms
    ax_kms.errorbar([v], [w], xerr=[ve], yerr=[we], color="k")

ax_kms.axis["bottom"].set_label("Linear velocity at 2.3 kpc [km/s]")
ax_kms.axis["left"].set_label("FWHM [km/s]")
ax_pm.axis["top"].set_label(r"Proper Motion [$'$/yr]")
ax_pm.axis["top"].label.set_visible(True)
ax_pm.axis["right"].major_ticklabels.set_visible(False)

ax_kms.set_xlim(950, 3700)
ax_kms.set_ylim(950, 3100)
# xlim and ylim of ax_pms will be automatically adjusted.

plt.draw()
plt.show()
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import make_axes_locatable

# the random data
x = np.random.randn(1000)
y = np.random.randn(1000)

fig, axScatter = plt.subplots(figsize=(5.5, 5.5))

# the scatter plot:
axScatter.scatter(x, y)
axScatter.set_aspect(1.)

# create new axes on the right and on the top of the current axes
# The first argument of the new_vertical(new_horizontal) method is
# the height (width) of the axes to be created in inches.
divider = make_axes_locatable(axScatter)
axHistx = divider.append_axes("top", 1.2, pad=0.1, sharex=axScatter)
axHisty = divider.append_axes("right", 1.2, pad=0.1, sharey=axScatter)

# make some labels invisible
plt.setp(axHistx.get_xticklabels() + axHisty.get_yticklabels(),
         visible=False)

# now determine nice limits by hand:
binwidth = 0.25
xymax = np.max([np.max(np.fabs(x)), np.max(np.fabs(y))])
lim = (int(xymax/binwidth) + 1)*binwidth
bins = np.arange(-lim, lim + binwidth, binwidth)
axHistx.hist(x, bins=bins)
axHisty.hist(y, bins=bins, orientation='horizontal')

# the xaxis of axHistx and yaxis of axHisty are shared with axScatter,
# thus there is no need to manually adjust the xlim and ylim of these
# axis.

#axHistx.axis["bottom"].major_ticklabels.set_visible(False)
for tl in axHistx.get_xticklabels():
    tl.set_visible(False)
axHistx.set_xticks([0, 50, 100])

#axHisty.axis["left"].major_ticklabels.set_visible(False)
for tl in axHisty.get_yticklabels():
    tl.set_visible(False)
axHisty.set_yticks([0, 50, 100])

plt.draw()
plt.show()
```python
import matplotlib.pyplot as plt

def draw_text(ax):
    from matplotlib.offsetbox import AnchoredText
    at = AnchoredText("Figure 1a", loc=2, prop=dict(size=8), frameon=True,
    )
    at.patch.set_boxstyle("round,pad=0.,rounding_size=0.2")
    ax.add_artist(at)

    at2 = AnchoredText("Figure 1(b)",
    loc=3, prop=dict(size=8), frameon=True,
    bbox_to_anchor=(0., 1.),
    bbox_transform=ax.transAxes
    )
    at2.patch.set_boxstyle("round,pad=0.,rounding_size=0.2")
    ax.add_artist(at2)

def draw_circle(ax):
    # circle in the canvas coordinate
    from mpl_toolkits.axes_grid1.anchored_artists import AnchoredDrawingArea
```
from matplotlib.patches import Circle
da = AnchoredDrawingArea(20, 20, 0, 0,
    loc=1, pad=0., frameon=False)

    p = Circle((10, 10), 10)
da.da.add_artist(p)
ax.add_artist(da)


def draw_ellipse(ax):
    from mpl_toolkits.axes_grid1.anchored_artists import AnchoredEllipse
    # draw an ellipse of width=0.1, height=0.15 in the data coordinate
    ae = AnchoredEllipse(ax.transData, width=0.1, height=0.15, angle=0.,
        loc=3, pad=0.5, borderpad=0.4, frameon=True)

    ax.add_artist(ae)


def draw_sizebar(ax):
    from mpl_toolkits.axes_grid1.anchored_artists import AnchoredSizeBar
    # draw a horizontal bar with length of 0.1 in Data coordinate
    # (ax.transData) with a label underneath.
    asb = AnchoredSizeBar(ax.transData,
        0.1,
        r"1"^\prime",
        loc=8,
        pad=0.1, borderpad=0.5, sep=5,
        frameon=False)

    ax.add_artist(asb)

if 1:
    ax = plt.gca()
    ax.set_aspect(1.)

    draw_text(ax)
draw_circle(ax)
    draw_ellipse(ax)
draw_sizebar(ax)

plt.show()
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import ImageGrid
import numpy as np

im = np.arange(100)
im.shape = 10, 10

fig = plt.figure(1, (4., 4.))
grid = ImageGrid(fig, 111,
                 # similar to subplot(111)
                 nrows_ncols=(2, 2),  # creates 2x2 grid of axes
                 axes_pad=0.1,  # pad between axes in inch.
                 )

for i in range(4):
    grid[i].imshow(im)  # The AxesGrid object work as a list of axes.

plt.show()
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import ImageGrid

def get_demo_image():
    import numpy as np
    from matplotlib.cbook import get_sample_data
    f = get_sample_data("axes_grid/bivariate_normal.npy", asfileobj=False)
    z = np.load(f)
    # z is a numpy array of 15x15
    return z, (-3, 4, -4, 3)

F = plt.figure(1, (5.5, 3.5))
grid = ImageGrid(F, 111,  # similar to subplot(111)
                 nrows_ncols=(1, 3),
                 axes_pad=0.1,
                 add_all=True,
                 label_mode="L",
                 )

Z, extent = get_demo_image()  # demo image

im1 = Z
im2 = Z[:, :10]
im3 = Z[:, 10:]

vmin, vmax = Z.min(), Z.max()
for i, im in enumerate([im1, im2, im3]):
    ax = grid[i]
```
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import host_subplot
import mpl_toolkits.axisartist as AA
import numpy as np

ax = host_subplot(111, axes_class=AA.Axes)
xx = np.arange(0, 2*np.pi, 0.01)
ax.plot(xx, np.sin(xx))

ax2 = ax.twin()  # ax2 is responsible for "top" axis and "right" axis
ax2.set_xticks([0, 0.5*np.pi, np.pi, 1.5*np.pi, 2*np.pi])
ax2.set_xticklabels([r'$0$', r'$\frac{1}{2}\pi$', r'$\pi$', r'$\frac{3}{2}\pi$', r'$2\pi$'])
```

**90.23 axes_grid example code: simple_axisline4.py**
ax2.axis['right'].major_ticklabels.set_visible(False)
ax2.axis['top'].major_ticklabels.set_visible(True)

plt.draw()
plt.show()
91.1 color example code: color_cycle_default.py

```python
Colors in the default prop_cycle

Display the colors from the default prop_cycle.

```
prop_cycle = plt.rcParams['axes.prop_cycle']
colors = prop_cycle.by_key()['color']

lwbase = plt.rcParams['lines.linewidth']
thin = float('%1f' % (lwbase / 2))
thick = lwbase * 3

fig, axs = plt.subplots(nrows=2, ncols=2, sharex=True, sharey=True)
for icol in range(2):
    if icol == 0:
        lwx, lwy = thin, lwbase
    else:
        lwx, lwy = lwbase, thick
    for irow in range(2):
        for i, color in enumerate(colors):
            axs[irow, icol].axhline(i, color=color, lw=lwx)
            axs[irow, icol].axvline(i, color=color, lw=lwy)

axs[1, icol].set_facecolor('k')
axs[1, icol].xaxis.set_ticks(np.arange(0, 10, 2))
axs[0, icol].set_title('line widths (pts): %.1f, %.1f' % (lwx, lwy),
                        fontsize='medium')

for irow in range(2):
    axs[irow, 0].yaxis.set_ticks(np.arange(0, 10, 2))

fig.suptitle('Colors in the default prop_cycle', fontsize='large')

plt.show()
### Styling with cycler

Demo of custom property-cycle settings to control colors and other style properties for multi-line plots.

This example demonstrates two different APIs:

1. Setting the default rc parameter specifying the property cycle. This affects all subsequent axes (but not axes already created).
2. Setting the property cycle for a single pair of axes.

```python
from cycler import cycler
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 2 * np.pi)
offsets = np.linspace(0, 2 * np.pi, 4, endpoint=False)
# Create array with shifted-sine curve along each column
```
yy = np.transpose([np.sin(x + phi) for phi in offsets])

# 1. Setting prop cycle on default rc parameter
plt.rc('lines', linewidth=4)
plt.rc('axes', prop_cycle=(cycler('color', ['r', 'g', 'b', 'y']) +
                           cycler('linestyle', ['-','--','\:\:\:\:\:', '-.'])))

fig, (ax0, ax1) = plt.subplots(nrows=2)
ax0.plot(yy)
ax0.set_title('Set default color cycle to rgyb')

# 2. Define prop cycle for single set of axes
ax1.set_prop_cycle(cycler('color', [c, 'm', 'y', 'k']) +
                    cycler('lw', [1, 2, 3, 4]))
ax1.plot(yy)
ax1.set_title('Set axes color cycle to cmyk')

# Tweak spacing between subplots to prevent labels from overlapping
fig.subplots_adjust(hspace=0.3)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

91.3 color example code: colormaps_reference.py

Perceptually Uniform Sequential colormaps

<table>
<thead>
<tr>
<th>viridis</th>
</tr>
</thead>
<tbody>
<tr>
<td>plasma</td>
</tr>
<tr>
<td>inferno</td>
</tr>
<tr>
<td>magma</td>
</tr>
</tbody>
</table>
Sequential colormaps

- Greys
- Purples
- Blues
- Greens
- Oranges
- Reds
- YlOrBr
- YlOrRd
- OrRd
- PuRd
- RdPu
- BuPu
- GnBu
- PuBu
- YIGnBu
- PuBuGn
- BuGn
- YIGn
Diverging colormaps

- PiYG
- PRGn
- BrBG
- PuOr
- RdGy
- RdBu
- RdYlBu
- RdYlGn
- Spectral
- coolwarm
- bwr
- seismic

91.3. color example code: colormaps_reference.py
## Qualitative colormaps

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Pastel1</td>
<td><img src="image1.png" alt="Image" /></td>
</tr>
<tr>
<td>Pastel2</td>
<td><img src="image2.png" alt="Image" /></td>
</tr>
<tr>
<td>Paired</td>
<td><img src="image3.png" alt="Image" /></td>
</tr>
<tr>
<td>Accent</td>
<td><img src="image4.png" alt="Image" /></td>
</tr>
<tr>
<td>Dark2</td>
<td><img src="image5.png" alt="Image" /></td>
</tr>
<tr>
<td>Set1</td>
<td><img src="image6.png" alt="Image" /></td>
</tr>
<tr>
<td>Set2</td>
<td><img src="image7.png" alt="Image" /></td>
</tr>
<tr>
<td>Set3</td>
<td><img src="image8.png" alt="Image" /></td>
</tr>
<tr>
<td>tab10</td>
<td><img src="image9.png" alt="Image" /></td>
</tr>
<tr>
<td>tab20</td>
<td><img src="image10.png" alt="Image" /></td>
</tr>
<tr>
<td>tab20b</td>
<td><img src="image11.png" alt="Image" /></td>
</tr>
<tr>
<td>tab20c</td>
<td><img src="image12.png" alt="Image" /></td>
</tr>
</tbody>
</table>
Reference for colormaps included with Matplotlib.

This reference example shows all colormaps included with Matplotlib. Note that any colormap listed here can be reversed by appending "_r" (e.g., "pink_r"). These colormaps are divided into the following categories:

Sequential:
These colormaps are approximately monochromatic colormaps varying smoothly between two color tones—usually from low saturation (e.g., white) to high saturation (e.g., a bright blue). Sequential colormaps are ideal for representing most scientific data since they show a clear progression from low-to-high values.

Diverging:
These colormaps have a median value (usually light in color) and vary smoothly to two different color tones at high and low values. Diverging colormaps are ideal when your data has a median value that is significant (e.g., 0, such that positive and negative values are represented by different colors of the colormap).

Qualitative:
These colormaps vary rapidly in color. Qualitative colormaps are useful for choosing a set of discrete colors. For example:

```python
color_list = plt.cm.Set3(np.linspace(0, 1, 12))
```
gives a list of RGB colors that are good for plotting a series of lines on a dark background.

**Miscellaneous:**
Colormaps that don’t fit into the categories above.

```python
import numpy as np
import matplotlib.pyplot as plt

# Have colormaps separated into categories:
# http://matplotlib.org/examples/color/colormaps_reference.html

cmaps = [("Perceptually Uniform Sequential", [ 'viridis', 'plasma', 'inferno', 'magma']),
         ('Sequential', [ 'Greys', 'Purples', 'Blues', 'Greens', 'Oranges', 'Reds',
                          'YlOrBr', 'YlOrRd', 'OrRd', 'PuRd', 'RdPu', 'BuPu',
                          'GnBu', 'PuBu', 'YlGnBu', 'PuBuGn', 'BuGn', 'YlGn'])),
         ('Sequential (2)', [ 'binary', 'gist_yarg', ' gist_gray', 'gray', 'bone', 'pink',
                              'spring', 'summer', 'autumn', 'winter', 'cool', 'warm',
                              'hot', 'afmhot', 'gist_heat', 'copper']),
         ('Diverging', [ 'PiYG', 'PRGn', 'BrBG', 'PuOr', 'RdGy', 'RdBu',
                         'YlGnBu', 'RdYlBu', 'RdYlGn', 'Spectral', 'coolwarm', 'bwr', 'seismic']),
         ('Qualitative', [ 'Pastel1', 'Pastel2', 'Paired', 'Accent',
                           'Dark2', 'Set1', 'Set2', 'Set3',
                           'tab10', 'tab20', 'tab20c', 'tab20c']),
         ('Miscellaneous', [ 'flag', 'prism', 'ocean', ' gist_earth', 'terrain', 'gist_stern',
                              'gnuplot', 'gnuplot2', 'CMRmap', 'cubehelix', 'brg', ' hsv',
                              'gist rainbow', ' rainbow', 'jet', 'nipy spectral', 'gist_ncar'])]

nrows = max(len(cmap_list) for cmap_category, cmap_list in cmaps)
gradient = np.linspace(0, 1, 256)
gradient = np.vstack((gradient, gradient))

```python
def plot_color_gradients(cmap_category, cmap_list, nrows):
    fig, axes = plt.subplots(nrows=nrows)
    fig.subplots_adjust(top=0.95, bottom=0.01, left=0.2, right=0.99)
    axes[0].set_title(cmap_category + ' colormaps', fontsize=14)

    for ax, name in zip(axes, cmap_list):
        ax.imshow(gradient, aspect='auto', cmap=plt.get_cmap(name))
```
pos = list(ax.get_position().bounds)
x_text = pos[0] - 0.01
fig.text(x_text, y_text, name, va='center', ha='right', fontsize=10)

# Turn off *all* ticks & spines, not just the ones with colormaps.
for ax in axes:
    ax.set_axis_off()

for cmap_category, cmap_list in cmaps:
    plot_color_gradients(cmap_category, cmap_list, nrows)

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

91.4. color example code: named_colors.py

```
# Visualizing named colors

Simple plot example with the named colors and its visual representation.

91.4. color example code: named_colors.py 2185
from __future__ import division
import matplotlib.pyplot as plt
from matplotlib import colors as mcolors

colors = dict(mcolors.BASE_COLORS, **mcolors.CSS4_COLORS)

# Sort colors by hue, saturation, value and name.
by_hsv = sorted(((tuple(mcolors.rgb_to_hsv(mcolors.to_rgba(color)[:3])), name)
    for name, color in colors.items()))
sorted_names = [name for hsv, name in by_hsv]
n = len(sorted_names)
ncols = 4
nrows = n // ncols + 1

fig, ax = plt.subplots(figsize=(8, 5))

# Get height and width
X, Y = fig.get_dpi() * fig.get_size_inches()
h = Y / (nrows + 1)
w = X / ncols

for i, name in enumerate(sorted_names):
    col = i % ncols
    row = i // ncols
    y = Y - (row * h) - h
    xi_line = w * (col + 0.05)
    xf_line = w * (col + 0.25)
    xi_text = w * (col + 0.3)
    ax.text(xi_text, y, name, fontsize=(h * 0.8),
            horizontalalignment='left',
            verticalalignment='center')
    ax.hlines(y + h * 0.1, xi_line, xf_line,
              color=colors[name], linewidth=(h * 0.6))

ax.set_xlim(0, X)
ax.set_ylim(0, Y)
ax.set_axis_off()
fig.subplots_adjust(left=0, right=1, top=1, bottom=0, hspace=0, wspace=0)
plt.show()
92.1 event_handling example code: close_event.py

[source code]

```python
from __future__ import print_function
import matplotlib.pyplot as plt

def handle_close(evt):
    print('Closed Figure!')

fig = plt.figure()
fig.canvas.mpl_connect('close_event', handle_close)

plt.text(0.35, 0.5, 'Close Me!', dict(size=30))
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

92.2 event_handling example code: data_browser.py

[source code]

```python
import numpy as np

class PointBrowser(object):
    """
    Click on a point to select and highlight it -- the data that
generated the point will be shown in the lower axes. Use the 'n'
and 'p' keys to browse through the next and previous points
"""

    def __init__(self):
        self.lastind = 0

        self.text = ax.text(0.05, 0.95, 'selected: none',
```
```python
def onpress(self, event):
    if self.lastind is None:
        return
    if event.key not in ('n', 'p'):
        return
    if event.key == 'n':
        inc = 1
    else:
        inc = -1
    self.lastind += inc
    self.lastind = np.clip(self.lastind, 0, len(xs) - 1)
    self.update()

def onpick(self, event):
    if event.artist != line:
        return True
    N = len(event.ind)
    if not N:
        return True
    # the click locations
    x = event.mouseevent.xdata
    y = event.mouseevent.ydata
    distances = np.hypot(x - xs[event.ind], y - ys[event.ind])
    indmin = distances.argmin()
    dataind = event.ind[indmin]
    self.lastind = dataind
    self.update()

def update(self):
    if self.lastind is None:
        return
    dataind = self.lastind
    ax2.cla()
    ax2.plot(X[dataind])
    ax2.text(0.05, 0.9, 'mu=%1.3f
sigma=%1.3f' % (xs[dataind], ys[dataind]),
        transform=ax2.transAxes, va='top')
    ax2.set_ylim(-0.5, 1.5)
    self.selected.set_visible(True)
    self.selected.set_data(xs[dataind], ys[dataind])
```
self.text.set_text('selected: %d' % dataind)
fig.canvas.draw()

if __name__ == '__main__':
    import matplotlib.pyplot as plt

    X = np.random.rand(100, 200)
    xs = np.mean(X, axis=1)
    ys = np.std(X, axis=1)

    fig, (ax, ax2) = plt.subplots(2, 1)
    ax.set_title('click on point to plot time series')
    line, = ax.plot(xs, ys, 'o', picker=5) # 5 points tolerance

    browser = PointBrowser()

    fig.canvas.mpl_connect('pick_event', browser.onpick)
    fig.canvas.mpl_connect('key_press_event', browser.onpress)

    plt.show()
event.canvas.draw()

def leave_figure(event):
    print('leave_figure', event.canvas.figure)
    event.canvas.figure.patch.set_facecolor('grey')
    event.canvas.draw()

fig1, (ax, ax2) = plt.subplots(2, 1)
fig1.suptitle('mouse hover over figure or axes to trigger events')

fig1.canvas.mpl_connect('figure_enter_event', enter_figure)
fig1.canvas.mpl_connect('figure_leave_event', leave_figure)
fig1.canvas.mpl_connect('axes_enter_event', enter_axes)
fig1.canvas.mpl_connect('axes_leave_event', leave_axes)

fig2, (ax, ax2) = plt.subplots(2, 1)
fig2.suptitle('mouse hover over figure or axes to trigger events')

fig2.canvas.mpl_connect('figure_enter_event', enter_figure)
fig2.canvas.mpl_connect('figure_leave_event', leave_figure)
fig2.canvas.mpl_connect('axes_enter_event', enter_axes)
fig2.canvas.mpl_connect('axes_leave_event', leave_axes)

plt.show()
def on_idle(event):
    on_idle.count += 1
    print('idle', on_idle.count)
    line1.set_ydata(np.sin(2*np.pi*t*(N - on_idle.count)/float(N)))
    event.canvas.draw()
    # test boolean return removal
    if on_idle.count == N:
        return False
    return True
on_idle.cid = None
on_idle.count = 0
fig.canvas.mpl_connect('idle_event', on_idle)
plt.show()

fig, ax = plt.subplots()
fig.canvas.mpl_connect('key_press_event', press)
ax.plot(np.random.rand(12), np.random.rand(12), 'go')
xl = ax.set_xlabel('easy come, easy go')
xl.set_visible(not visible)
fig.canvas.draw()

plt.show()
92.6 event_handling example code: lasso_demo.py

```python
""
Show how to use a lasso to select a set of points and get the indices of the selected points. A callback is used to change the color of the selected points.

This is currently a proof-of-concept implementation (though it is usable as is). There will be some refinement of the API.
""

```
ax.add_collection(self.collection)

self.cid = self.canvas.mpl_connect('button_press_event', self.onpress)

def callback(self, verts):
    facecolors = self.collection.get_facecolors()
    p = path.Path(verts)
    ind = p.contains_points(self.xys)
    for i in range(len(self.xys)):
        if ind[i]:
            facecolors[i] = Datum.colorin
        else:
            facecolors[i] = Datum.colorout

    self.canvas.draw_idle()
    self.canvas.widgetlock.release(self.lasso)
    del self.lasso

def onpress(self, event):
    if self.canvas.widgetlock.locked():
        return
    if event.inaxes is None:
        return
    self.lasso = Lasso(event.inaxes, (event.xdata, event.ydata), self.callback)
    self.canvas.widgetlock(self.lasso)

if __name__ == '__main__':
    data = [Datum(*xy) for xy in rand(100, 2)]
    ax = plt.axes(xlim=(0, 1), ylim=(0, 1), autoscale_on=False)
    ax.set_title('Lasso points using left mouse button')

    lman = LassoManager(ax, data)
    plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

92.7 event_handling example code: legend_picking.py

[source code]

""
Enable picking on the legend to toggle the original line on and off
""
import numpy as np
import matplotlib.pyplot as plt

92.7. event_handling example code: legend_picking.py 2193
```python
t = np.arange(0.0, 0.2, 0.1)
y1 = 2*np.sin(2*np.pi*t)
y2 = 4*np.sin(2*np.pi*2*t)

fig, ax = plt.subplots()
ax.set_title('Click on legend line to toggle line on/off')
line1, = ax.plot(t, y1, lw=2, color='red', label='1 HZ')
line2, = ax.plot(t, y2, lw=2, color='blue', label='2 HZ')
leg = ax.legend(loc='upper left', fancybox=True, shadow=True)
leg.get_frame().set_alpha(0.4)

# we will set up a dict mapping legend line to orig line, and enable picking on the legend line
lines = [line1, line2]
lined = dict()
for legline, origline in zip(leg.get_lines(), lines):
    legline.set_picker(5)  # 5 pts tolerance
    lined[legline] = origline

def onpick(event):
    # on the pick event, find the orig line corresponding to the legend proxy line, and toggle the visibility
    legline = event.artist
    origline = lined[legline]
    vis = not origline.get_visible()
    origline.set_visible(vis)
    # Change the alpha on the line in the legend so we can see what lines have been toggled
    if vis:
        legline.set_alpha(1.0)
    else:
        legline.set_alpha(0.2)
    fig.canvas.draw()

fig.canvas.mpl_connect('pick_event', onpick)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

### 92.8 event_handling example code: looking_glass.py

[source code]
fig, ax = plt.subplots()
circ = patches.Circle((0.5, 0.5), 0.25, alpha=0.8, fc='yellow')
ax.add_patch(circ)

ax.plot(x, y, alpha=0.2)
line, = ax.plot(x, y, alpha=1.0, clip_path=circ)
ax.set_title("Left click and drag to move looking glass")

class EventHandler(object):
    def __init__(self):
        fig.canvas.mpl_connect('button_press_event', self.onpress)
        fig.canvas.mpl_connect('button_release_event', self.onrelease)
        fig.canvas.mpl_connect('motion_notify_event', self.onmove)
        self.x0, self.y0 = circ.center
        self.pressevent = None

    def onpress(self, event):
        if event.inaxes != ax:
            return

        if not circ.contains(event)[0]:
            return

        self.pressevent = event

    def onrelease(self, event):
        self.pressevent = None
        self.x0, self.y0 = circ.center

    def onmove(self, event):
        if self.pressevent is None or event.inaxes != self.pressevent.inaxes:
            return

        dx = event.xdata - self.pressevent.xdata
        dy = event.ydata - self.pressevent.ydata
        circ.center = self.x0 + dx, self.y0 + dy
        line.set_clip_path(circ)
        fig.canvas.draw()

handler = EventHandler()
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

92.9 event_handling example code: path_editor.py

[source code]
import numpy as np
import matplotlib.path as mpath
import matplotlib.patches as mpatches
import matplotlib.pyplot as plt

Path = mpath.Path

fig, ax = plt.subplots()

pathdata = [
    (Path.MOVETO, (1.58, -2.57)),
    (Path.CURVE4, (0.35, -1.1)),
    (Path.CURVE4, (-1.75, 2.0)),
    (Path.CURVE4, (0.375, 2.0)),
    (Path.LINETO, (0.85, 1.15)),
    (Path.CURVE4, (2.2, 3.2)),
    (Path.CURVE4, (3, 0.05)),
    (Path.CURVE4, (2.0, -0.5)),
    (Path.CLOSEPOLY, (1.58, -2.57)),
]

codes, verts = zip(*pathdata)
path = mpath.Path(verts, codes)
patch = mpatches.PathPatch(path, facecolor='green', edgecolor='yellow', alpha=0.5)
ax.add_patch(patch)

class PathInteractor(object):
    ""
    An path editor.
    ""

    """Key-bindings
    """
    't' toggle vertex markers on and off. When vertex markers are on,
    you can move them, delete them

    """

    showverts = True
    epsilon = 5  # max pixel distance to count as a vertex hit

    def __init__(self, pathpatch):
        self.ax = pathpatch.axes
        canvas = self.ax.figure.canvas
        self.pathpatch = pathpatch
        self.pathpatch.set_animated(True)

        x, y = zip(*self.pathpatch.get_path().vertices)
        self.line, = ax.plot(x, y, marker='o', markerfacecolor='r', animated=True)
```python
self._ind = None  # the active vert

canvas.mpl_connect('draw_event', self.draw_callback)
canvas.mpl_connect('button_press_event', self.button_press_callback)
canvas.mpl_connect('button_release_event', self.button_release_callback)
canvas.mpl_connect('motion_notify_event', self.motion_notify_callback)
self.canvas = canvas

def draw_callback(self, event):
    self.background = self.canvas.copy_from_bbox(self.ax.bbox)
    self.ax.draw_artist(self.pathpatch)
    self.ax.draw_artist(self.line)
    self.canvas.blit(self.ax.bbox)

def pathpatch_changed(self, pathpatch):
    # this method is called whenever the pathpatchgon object is called
    # only copy the artist props to the line (except visibility)
    vis = self.line.get_visible()
    plt.Artist.update_from(self.line, pathpatch)
    self.line.set_visible(vis)  # don't use the pathpatch visibility state

def get_ind_under_point(self, event):
    # get the index of the vertex under point if within tolerance

    # display coords
    xy = np.asarray(self.pathpatch.get_path().vertices)
    xyt = self.pathpatch.get_transform().transform(xy)
    xt, yt = xyt[:, 0], xyt[:, 1]
    d = np.sqrt((xt - event.x)**2 + (yt - event.y)**2)
    ind = d.argmin()

    if d[ind] >= self.epsilon:
        ind = None
    return ind

def button_press_callback(self, event):
    # whenever a mouse button is pressed
    if not self.showverts:
        return
    if event.inaxes is None:
        return
    if event.button != 1:
        return
    self._ind = self.get_ind_under_point(event)

def button_release_callback(self, event):
    # whenever a mouse button is released
    if not self.showverts:
        return
    if event.button != 1:
        return
```

---

92.9. event_handling example code: path_editor.py 2197
def key_press_callback(self, event):
    'whenever a key is pressed'
    if not event.inaxes:
        return
    if event.key == 't':
        self.showverts = not self.showverts
        self.line.set_visible(self.showverts)
        if not self.showverts:
            self._ind = None
    self.canvas.draw()

def motion_notify_callback(self, event):
    'on mouse movement'
    if not self.showverts:
        return
    if self._ind is None:
        return
    if event.inaxes is None:
        return
    if event.button != 1:
        return
    x, y = event.xdata, event.ydata
    vertices = self.pathpatch.get_path().vertices
    vertices[self._ind] = x, y
    self.line.set_data(zip(*vertices))
    self.canvas.restore_region(self.background)
    self.ax.draw_artist(self.pathpatch)
    self.ax.draw_artist(self.line)
    self.canvas.blit(self.ax.bbox)

interactor = PathInteractor(patch)
ax.set_title('drag vertices to update path')
ax.set_xlim(-3, 4)
ax.set_ylim(-3, 4)

plt.show()
You can enable picking by setting the "picker" property of an artist (for example, a matplotlib Line2D, Text, Patch, Polygon, AxesImage, etc...)

There are a variety of meanings of the picker property

- **None** - picking is disabled for this artist (default)
- **boolean** - if True then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist
- **float** - if picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if it's data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, for example, the indices of the data within epsilon of the pick event
- **function** - if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event.

\[
\text{hit, props = picker(artist, mouseevent)}
\]

To determine the hit test. If the mouse event is over the artist, return hit=True and props is a dictionary of properties you want added to the PickEvent attributes.

After you have enabled an artist for picking by setting the "picker" property, you need to connect to the figure canvas pick_event to get pick callbacks on mouse press events. For example,

\[
\text{def pick_handler(event):}
\]

\[
\text{mouseevent = event.mouseevent}
\]

\[
\text{artist = event.artist}
\]

\[
\text{# now do something with this...}
\]

The pick event (matplotlib.backend_bases.PickEvent) which is passed to your callback is always fired with two attributes:

- **mouseevent** - the mouse event that generate the pick event. The mouse event in turn has attributes like x and y (the coordinates in display space, such as pixels from left, bottom) and xdata, ydata (the coords in data space). Additionally, you can get information about which buttons were pressed, which keys were pressed, which Axes the mouse is over, etc. See matplotlib.backend_bases.MouseEvent for details.
artist - the matplotlib.artist that generated the pick event.

Additionally, certain artists like Line2D and PatchCollection may attach additional meta data like the indices into the data that meet the picker criteria (for example, all the points in the line that are within the specified epsilon tolerance)

The examples below illustrate each of these methods.

```python
from __future__ import print_function
import matplotlib.pyplot as plt
from matplotlib.lines import Line2D
from matplotlib.patches import Rectangle
from matplotlib.text import Text
from matplotlib.image import AxesImage
import numpy as np
import numpy.random as rand

if 1:
    # simple picking, lines, rectangles and text
    fig, (ax1, ax2) = plt.subplots(2, 1)
    ax1.set_title('click on points, rectangles or text', picker=True)
    ax1.set_ylabel('ylabel', picker=True, bbox=dict(facecolor='red'))
    line, = ax1.plot(rand(100), 'o', picker=5)  # 5 points tolerance

    # pick the rectangle
    bars = ax2.bar(range(10), rand(10), picker=True)
    for label in ax2.get_xticklabels():  # make the xtick labels pickable
        label.set_picker(True)

    def onpick1(event):
        if isinstance(event.artist, Line2D):
            thisline = event.artist
            xdata = thisline.get_xdata()
            ydata = thisline.get_ydata()
            ind = event.ind
            print('onpick1 line:', zip(np.take(xdata, ind), np.take(ydata, ind)))
        elif isinstance(event.artist, Rectangle):
            patch = event.artist
            print('onpick1 patch:', patch.get_path())
        elif isinstance(event.artist, Text):
            text = event.artist
            print('onpick1 text:', text.get_text())

    fig.canvas.mpl_connect('pick_event', onpick1)

if 1:
    # picking with a custom hit test function
    # you can define custom pickers by setting picker to a callable
    # function. The function has the signature
    #
    # hit, props = func(artist, mouseevent)
    #
```
# to determine the hit test. if the mouse event is over the artist,
# return hit=True and props is a dictionary of
# properties you want added to the PickEvent attributes

```python
def line_picker(line, mouseevent):
    
    """
    find the points within a certain distance from the mouseclick in
data coords and attach some extra attributes, pickx and picky
which are the data points that were picked
"""
    if mouseevent.xdata is None:
        return False, dict()
    xdata = line.get_xdata()
ydata = line.get_ydata()
maxd = 0.05
d = np.sqrt((xdata - mouseevent.xdata)**2. + (ydata - mouseevent.ydata)**2.)
    ind = np.nonzero(np.less_equal(d, maxd))
    if len(ind):
        pickx = np.take(xdata, ind)
picky = np.take(ydata, ind)
        props = dict(ind=ind, pickx=pickx, picky=picky)
        return True, props
    else:
        return False, dict()

def onpick2(event):
    print('onpick2 line:', event.pickx, event.picky)

fig, ax = plt.subplots()
ax.set_title('custom picker for line data')
line, = ax.plot(rand(100), rand(100), 'o', picker=line_picker)
fig.canvas.mpl_connect('pick_event', onpick2)

if 1:
    # picking on a scatter plot (matplotlib.collections.RegularPolyCollection)
    x, y, c, s = rand(4, 100)

    def onpick3(event):
        ind = event.ind
        print('onpick3 scatter:', ind, np.take(x, ind), np.take(y, ind))

    fig, ax = plt.subplots()
col = ax.scatter(x, y, 100*s, c, picker=True)
    #fig.savefig('pscoll.eps')
    fig.canvas.mpl_connect('pick_event', onpick3)

if 1:
    # picking images (matplotlib.image.AxesImage)
    fig, ax = plt.subplots()
im1 = ax.imshow(rand(10, 5), extent=(1, 2, 1, 2), picker=True)
im2 = ax.imshow(rand(5, 10), extent=(3, 4, 1, 2), picker=True)
im3 = ax.imshow(rand(20, 25), extent=(1, 2, 3, 4), picker=True)
```
im4 = ax.imshow(rand(30, 12), extent=(3, 4, 3, 4), picker=True)
ax.axis([0, 5, 0, 5])

def onpick4(event):
    artist = event.artist
    if isinstance(artist, AxesImage):
        im = artist
        A = im.get_array()
        print('onpick4 image', A.shape)

    fig.canvas.mpl_connect('pick_event', onpick4)

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

92.11 event_handling example code: pick_event_demo2.py

[source code]

""
compute the mean and standard deviation (stddev) of 100 data sets and plot
mean vs stddev. When you click on one of the mu, sigma points, plot the raw
data from the dataset that generated the mean and stddev.
""

import numpy
import matplotlib.pyplot as plt

X = numpy.random.rand(100, 1000)
xs = numpy.mean(X, axis=1)
ys = numpy.std(X, axis=1)

fig, ax = plt.subplots()
ax.set_title('click on point to plot time series')
line, = ax.plot(xs, ys, 'o', picker=5)  # 5 points tolerance

def onpick(event):
    if event.artist != line:
        return True

    N = len(event.ind)
    if not N:
        return True

    figi = plt.figure()
    for subplotnum, dataind in enumerate(event.ind):
        ax = figi.add_subplot(N, 1, subplotnum + 1)
ax.plot(X[dataind])
ax.text(0.05, 0.9, 'mu=%1.3f
nsigma=%1.3f' % (xs[dataind], ys[dataind]),
       transform=ax.transAxes, va='top')
ax.set_ylim(-0.5, 1.5)
figi.show()  
return True

fig.canvas.mpl_connect('pick_event', onpick)

plt.show()
self.y = y
self.w = .3
self.score = 0
self.xoffset = .3
self.yoffset = .1
if type == 'r':
    self.xoffset *= -1.0

if type == 'l' or type == 'r':
    self.signx = -1.0
    self.signy = 1.0
else:
    self.signx = 1.0
    self.signy = -1.0

def contains(self, loc):
    return self.disp.get_bbox().contains(loc.x, loc.y)

class Puck(object):
    def __init__(self, disp, pad, field):
        self.vmax = .2
        self.disp = disp
        self.field = field
        self._reset(pad)

    def _reset(self, pad):
        self.x = pad.x + pad.xoffset
        if pad.y < 0:
            self.y = pad.y + pad.yoffset
        else:
            self.y = pad.y - pad.yoffset
        self.vx = pad.x - self.x
        self.vy = pad.y + pad.w/2 - self.y
        self._speedlimit()
        self._slower()
        self._slower()

    def update(self, pads):
        self.x += self.vx
        self.y += self.vy
        for pad in pads:
            if pad.contains(self):
                self.vx *= 1.2 * pad.signx
                self.vy *= 1.2 * pad.signy
                fudge = .001
                # probably cleaner with something like...
                # if not self.field.contains(self.x, self.y):
                if self.x < fudge:
                    #print("player A loses")
                    pads[1].score += 1
                    self._reset(pads[0])
                    return True
if self.x > 7 - fudge:
    #print("player B loses")
    pads[0].score += 1
    self._reset(pads[1])
    return True

if self.y < -1 + fudge or self.y > 1 - fudge:
    self.vy *= -1.0
    # add some randomness, just to make it interesting
    self.vy = (randn() / 300.0 + 1 / 300.0) * np.sign(self.vy)
    self._speedlimit()
    return False

def _slower(self):
    self.vx /= 5.0
    self.vy /= 5.0

def _faster(self):
    self.vx *= 5.0
    self.vy *= 5.0

def _speedlimit(self):
    if self.vx > self.vmax:
        self.vx = self.vmax
    if self.vx < -self.vmax:
        self.vx = -self.vmax

    if self.vy > self.vmax:
        self.vy = self.vmax
    if self.vy < -self.vmax:
        self.vy = -self.vmax

class Game(object):
    def __init__(self, ax):
        # create the initial line
        self.ax = ax
        padAx = padBx = .50
        padAy = padBy = .30
        padBx += 6.3
        pA, = self.ax.barh(padAy, .2, height=.3, color='k', alpha=.5, edgecolor='b', lw=2, label="Player B", animated=True)
        pB, = self.ax.barh(padBy, .2, height=.3, left=padBx, color='k', alpha=.5, edgecolor='r', lw=2, label="Player A", animated=True)

        pA, = self.ax.barh(padAy, .2, height=.3, color='k', alpha=.5, edgecolor='b', lw=2, label="Player B", animated=True)
        pB, = self.ax.barh(padBy, .2, height=.3, left=padBx, color='k', alpha=.5, edgecolor='r', lw=2, label="Player A", animated=True)

        self.centerline, = self.ax.plot([3.5, 3.5], [1, -1], 'k', alpha=.5, animated=True, lw=8)
        self.puckdisp = self.ax.scatter([1], [1], label="_nolegend_", s=200, c='g', alpha=.9, animated=True)
self.canvas = self.ax.figure.canvas
self.background = None
self.cnt = 0
self.distract = True
self.res = 100.0
self.on = False
self.inst = True    # show instructions from the beginning
self.background = None
self.pads = []
self.pads.append(Pad(pA, 0, padAy))
self.pads.append(Pad(pB, padBx, padBy, 'r'))
self.pucks = []
self.i = self.ax.annotate(instructions, (.5, 0.5),
                         name='monospace',
                         verticalalignment='center',
                         horizontalalignment='center',
                         multialignment='left',
                         textcoords='axes fraction', animated=True)
self.canvas.mpl_connect('key_press_event', self.key_press)

def draw(self, evt):
    draw_artist = self.ax.draw_artist
    if self.background is None:
        self.background = self.canvas.copy_from_bbox(self.ax.bbox)

# restore the clean slate background
self.canvas.restore_region(self.background)

# show the distractors
if self.distract:
    self.line.set_ydata(np.sin(self.x + self.cnt/self.res))
    self.line2.set_ydata(np.cos(self.x - self.cnt/self.res))
    self.line3.set_ydata(np.tan(self.x + self.cnt/self.res))
    self.line4.set_ydata(np.tan(self.x - self.cnt/self.res))
    draw_artist(self.line)
    draw_artist(self.line2)
    draw_artist(self.line3)
    draw_artist(self.line4)

# show the instructions - this is very slow
if self.inst:
    self.ax.draw_artist(self.i)

# pucks and pads
if self.on:
    self.ax.draw_artist(self.centerline)
    for pad in self.pads:
        pad.disp.set_y(pad.y)
        pad.disp.set_x(pad.x)
        self.ax.draw_artist(pad.disp)

    for puck in self.pucks:
if puck.update(self.pads):
    # we only get here if someone scored
    self.pads[0].disp.set_label(" " + str(self.pads[0].score))
    self.pads[1].disp.set_label(" " + str(self.pads[1].score))
    self.ax.legend(loc='center')
    self.leg = self.ax.get_legend()
    self.leg.get_frame().set_alpha(.2)
    plt.setp(self.leg.get_texts(), fontweight='bold', fontsize='xx-large')
    self.leg.get_frame().set_facecolor('0.2')
    self.background = None
    self.ax.figure.canvas.draw()
    return True
puck.disp.setOffsets([puck.x, puck.y])
self.ax.draw_artist(puck.disp)

# just redraw the axes rectangle
self.canvas.blit(self.ax.bbox)

if self.cnt == 50000:
    # just so we don't get carried away
    print("...and you've been playing for too long!!!")
    plt.close()

self.cnt += 1
return True

def key_press(self, event):
    if event.key == '3':
        self.res *= 5.0
    if event.key == '4':
        self.res /= 5.0

    if event.key == 'e':
        self.pads[0].y += .1
        if self.pads[0].y > 1 - .3:
            self.pads[0].y = 1 - .3
    if event.key == 'd':
        self.pads[0].y -= .1
        if self.pads[0].y < -1:
            self.pads[0].y = -1

    if event.key == 'i':
        self.pads[1].y += .1
        if self.pads[1].y > 1 - .3:
            self.pads[1].y = 1 - .3
    if event.key == 'k':
        self.pads[1].y -= .1
        if self.pads[1].y < -1:
            self.pads[1].y = -1

    if event.key == 'a':
self.pucks.append(Puck(self.puckdisp, self.pads[randint(2)], self.ax.bbox))
if event.key == 'A' and len(self.pucks):
    self.pucks.pop()
if event.key == ' ' and len(self.pucks):
    self.pucks[0]._reset(self.pads[randint(2)])
if event.key == '1':
    for p in self.pucks:
        p._slower()
if event.key == '2':
    for p in self.pucks:
        p._faster()
if event.key == 'n':
    self.distract = not self.distract
if event.key == 'g':
    #self.ax.clear()
    self.on = not self.on
if event.key == 't':
    self.inst = not self.inst
    self.i.set_visible(self.i.get_visible())
if event.key == 'q':
    plt.close()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

92.13 event_handling example code: poly_editor.py

[source code]

""
This is an example to show how to build cross-GUI applications using
matplotlib event handling to interact with objects on the canvas
""

import numpy as np
from matplotlib import Line2D
from matplotlib import Artist
from matplotlib.mlab import dist_point_to_segment

class PolygonInteractor(object):
    ""
    An polygon editor.
    Key-bindings
    't' toggle vertex markers on and off. When vertex markers are on,
    you can move them, delete them
    'd' delete the vertex under point
'i' insert a vertex at point. You must be within epsilon of the 
line connecting two existing vertices

showverts = True
epsilon = 5   # max pixel distance to count as a vertex hit

def __init__(self, ax, poly):
    if poly.figure is None:
        raise RuntimeError('You must first add the polygon to a figure or canvas, ' 
→before defining the interactor')
    self.ax = ax
    canvas = poly.figure.canvas
    self.poly = poly

    x, y = zip(*self.poly.xy)
    self.line = Line2D(x, y, marker='o', markerfacecolor='r', animated=True)
    self.ax.add_line(self.line)
    self._update_line(poly)
    cid = self.poly.add_callback(self.poly_changed)
    self._ind = None # the active vert

    canvas.mpl_connect('draw_event', self.draw_callback)
    canvas.mpl_connect('button_press_event', self.button_press_callback)
    canvas.mpl_connect('key_press_event', self.key_press_callback)
    canvas.mpl_connect('button_release_event', self.button_release_callback)
    canvas.mpl_connect('motion_notify_event', self.motion_notify_callback)
    self.canvas = canvas

def draw_callback(self, event):
    self.background = self.canvas.copy_from_bbox(self.ax.bbox)
    self.ax.draw_artist(self.poly)
    self.ax.draw_artist(self.line)
    self.canvas.blit(self.ax.bbox)

def poly_changed(self, poly):
    """this method is called whenever the polygon object is called"
    # only copy the artist props to the line (except visibility)
    vis = self.line.get_visible()
    Artist.update_from(self.line, poly)
    self.line.set_visible(vis) # don't use the poly visibility state

def get_ind_under_point(self, event):
    'get the index of the vertex under point if within epsilon tolerance'

    # display coords
    xy = np.asarray(self.poly.xy)
    xyt = self.poly.get_transform().transform(xy)
    xt, yt = xyt[:, 0], xyt[:, 1]
    d = np.sqrt((xt - event.x)**2 + (yt - event.y)**2)
indseq = np.nonzero(np.equal(d, np.amin(d)))[0]
ind = indseq[0]

if d[ind] >= self.epsilon:
    ind = None

return ind

def button_press_callback(self, event):
    'whenever a mouse button is pressed'
    if not self.showverts:
        return
    if event.inaxes is None:
        return
    if event.button != 1:
        return
    self._ind = self.get_ind_under_point(event)

def button_release_callback(self, event):
    'whenever a mouse button is released'
    if not self.showverts:
        return
    if event.button != 1:
        return
    self._ind = None

def key_press_callback(self, event):
    'whenever a key is pressed'
    if not event.inaxes:
        return
    if event.key == 't':
        self.showverts = not self.showverts
        self.line.set_visible(self.showverts)
        if not self.showverts:
            self._ind = None
    elif event.key == 'd':
        ind = self.get_ind_under_point(event)
        if ind is not None:
            self.poly.xy = [tup for i, tup in enumerate(self.poly.xy) if i != ind]
            self.line.set_data(zip(*self.poly.xy))
    elif event.key == 'i':
        xys = self.poly.get_transform().transform(self.poly.xy)
p = event.x, event.y  # display coords
for i in range(len(xys) - 1):
    s0 = xys[i]
s1 = xys[i + 1]
d = dist_point_to_segment(p, s0, s1)
if d <= self.epsilon:
    self.poly.xy = np.array(
        list(self.poly.xy[:i]) +
        [(event.xdata, event.ydata)] +
        list(self.poly.xy[i:]))
    self.line.set_data(zip(*self.poly.xy))
```python
break

self.canvas.draw()

def motion_notify_callback(self, event):
    'on mouse movement'
    if not self.showverts:
        return
    if self._ind is None:
        return
    if event.inaxes is None:
        return
    if event.button != 1:
        return
    x, y = event.xdata, event.ydata

    self.poly.xy[self._ind] = x, y
    if self._ind == 0:
        self.poly.xy[-1] = x, y
    elif self._ind == len(self.poly.xy) - 1:
        self.poly.xy[0] = x, y
    self.line.set_data(zip(*self.poly.xy))

    self.canvas.restore_region(self.background)
    self.ax.draw_artist(self.poly)
    self.ax.draw_artist(self.line)
    self.canvas.blit(self.ax.bbox)

if __name__ == '__main__':
    import matplotlib.pyplot as plt
    from matplotlib.patches import Polygon

    theta = np.arange(0, 2*np.pi, 0.1)
    r = 1.5

    xs = r*np.cos(theta)
    ys = r*np.sin(theta)

    poly = Polygon(list(zip(xs, ys)), animated=True)

    fig, ax = plt.subplots()
    ax.add_patch(poly)
    p = PolygonInteractor(ax, poly)

    #ax.add_line(p.line)
    ax.set_title('Click and drag a point to move it')
    ax.set_xlim((-2, 2))
    ax.set_ylim((-2, 2))
    plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
### 92.14 event_handling example code: pong_gtk.py

[source code]

```python
from __future__ import print_function

# For detailed comments on animation and the techniques used here, see
# the wiki entry
# http://www.scipy.org/wikis/topical_software/MatplotlibAnimation
import time

import gobject
import matplotlib
matplotlib.use('GTKAgg')
import matplotlib.pyplot as plt
import pipong

fig, ax = plt.subplots()
canvas = ax.figure.canvas

def start_anim(event):
    gobject.idle_add(animation.draw, animation)
    gobject.timeout_add(10, animation.draw, animation)
    canvas.mpl_disconnect(start_anim.cid)

animation = pipong.Game(ax)
start_anim.cid = canvas.mpl_connect('draw_event', start_anim)

tstart = time.time()
plt.grid()
plt.show()
print('
FPS: %f' % animation.cnt/(time.time() - tstart))
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

### 92.15 event_handling example code: resample.py

[source code]

```python
import numpy as np
import matplotlib.pyplot as plt

# A class that will downsample the data and recompute when zoomed.
class DataDisplayDownsampler(object):
    def __init__(self, xdata, ydata):
```

---

2212 Chapter 92. event_handling Examples
```python
self.origYData = ydata
self.origXData = xdata
self.ratio = 5
self.delta = xdata[-1] - xdata[0]

def downsample(self, xstart, xend):
    # Very simple downsampling that takes the points within the range
    # and picks every Nth point
    mask = (self.origXData > xstart) & (self.origXData < xend)
    xdata = self.origXData[mask]
    xdata = xdata[::self.ratio]
    ydata = self.origYData[mask]
    ydata = ydata[::self.ratio]
    return xdata, ydata

def update(self, ax):
    # Update the line
    lims = ax.viewLim
    if np.abs(lims.width - self.delta) > 1e-8:
        self.delta = lims.width
        xstart, xend = lims.intervalx
        self.line.set_data(*self.downsample(xstart, xend))
        ax.figure.canvas.draw_idle()

    # Create a signal
    xdata = np.linspace(16, 365, 365-16)
    ydata = np.sin(2*np.pi*xdata/153) + np.cos(2*np.pi*xdata/127)
    d = DataDisplayDownsampler(xdata, ydata)

    fig, ax = plt.subplots()
    # Hook up the line
    d.line = ax.plot(xdata, ydata, 'o-')[0]
    ax.set_autoscale_on(False)  # Otherwise, infinite loop
    # Connect for changing the view limits
    ax.callbacks.connect('xlim_changed', d.update)

    plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see [Search examples](https://matplotlib.org/codex/search.html))

### 92.16 event_handling example code: test_mouseclicks.py

[source code]

```python
from __future__ import print_function
```
import matplotlib
#matplotlib.use("WxAgg")
#matplotlib.use("TkAgg")
#matplotlib.use("GTKAgg")
#matplotlib.use("Qt4Agg")
#matplotlib.use("MacOSX")
import matplotlib.pyplot as plt

#print("***** TESTING WITH BACKEND: %s"%matplotlib.get_backend() + " *****")

def OnClick(event):
    if event.dblclick:
        print("DBLCLICK", event)
    else:
        print("DOWN ", event)

def OnRelease(event):
    print("UP ", event)

fig = plt.gcf()
cid_up = fig.canvas.mpl_connect('button_press_event', OnClick)
cid_down = fig.canvas.mpl_connect('button_release_event', OnRelease)

plt.gca().text(0.5, 0.5, "Click on the canvas to test mouse events.",
    ha="center", va="center")
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

92.17 event_handling example code: timers.py

[source code]

# Simple example of using general timer objects. This is used to update
# the time placed in the title of the figure.
import matplotlib.pyplot as plt
import numpy as np
from datetime import datetime

def update_title(axes):
    axes.set_title(datetime.now())
    axes.figure.canvas.draw()

fig, ax = plt.subplots()
x = np.linspace(-3, 3)
ax.plot(x, x\(^2\))

# Create a new timer object. Set the interval to 100 milliseconds  
# (1000 is default) and tell the timer what function should be called.
timer = fig.canvas.new_timer(interval=100)
timer.add_callback(update_title, ax)
timer.start()

# Or could start the timer on first figure draw
#def start_timer(evt):
#    timer.start()
#    fig.canvas.mpl_disconnect(drawid)
#drawid = fig.canvas.mpl_connect('draw_event', start_timer)

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

92.18 event_handling example code: trifinder_event_demo.py

[source code]

"""
Example showing the use of a TriFinder object. As the mouse is moved over the
triangulation, the triangle under the cursor is highlighted and the index of
the triangle is displayed in the plot title.
"""

import matplotlib.pyplot as plt
from matplotlib.tri import Triangulation
from matplotlib.patches import Polygon
import numpy as np
import math

def update_polygon(tri):
    if tri == -1:
        points = [0, 0, 0]
    else:
        points = triangulation.triangles[tri]
x = triangulation.x[points]
y = triangulation.y[points]
polygon.set_xy(list(zip(x, y)))

def motion_notify(event):
    if event.inaxes is None:
        tri = -1
    else:
        tri = trifinder(event.xdata, event.ydata)
update_polygon(tri)
plt.title('In triangle %i' % tri)
event.canvas.draw()

# Create a Triangulation.
 n_angles = 16
 n_radii = 5
 min_radius = 0.25
 radii = np.linspace(min_radius, 0.95, n_radii)
 angles = np.linspace(0, 2*math.pi, n_angles, endpoint=False)
 angles = np.repeat(angles..., np.newaxis, axis=1)
 angles[:, 1::2] += math.pi / n_angles
 x = (radii*np.cos(angles)).flatten()
 y = (radii*np.sin(angles)).flatten()
 triangulation = Triangulation(x, y)
 xmid = x[triangulation.triangles].mean(axis=1)
 ymid = y[triangulation.triangles].mean(axis=1)
 mask = np.where(xmid*xmid + ymid*ymid < min_radius*min_radius, 1, 0)
 triangulation.set_mask(mask)

# Use the triangulation's default TriFinder object.
 trifinder = triangulation.get_trifinder()

# Setup plot and callbacks.
 plt.subplot(111, aspect='equal')
 plt.triplot(triangulation, 'bo-')
 polygon = Polygon([[0, 0], [0, 0]], facecolor='y')  # dummy data for xs,ys
 update_polygon(-1)
 plt.gca().add_patch(polygon)
 plt.gcf().canvas.mpl_connect('motion_notify_event', motion_notify)
 plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

92.19 event_handling example code: viewlims.py

[source code]

# Creates two identical panels. Zooming in on the right panel will show
# a rectangle in the first panel, denoting the zoomed region.
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.patches import Rectangle

# We just subclass Rectangle so that it can be called with an Axes
# instance, causing the rectangle to update its shape to match the
# bounds of the Axes
class UpdatingRect(Rectangle):
    def __call__(self, ax):
        self.set_bounds(*ax.viewLim.bounds)
        ax.figure.canvas.draw_idle()
# A class that will regenerate a fractal set as we zoom in, so that you
# can actually see the increasing detail. A box in the left panel will show
# the area to which we are zoomed.
class MandelbrotDisplay(object):
    def __init__(self, h=500, w=500, niter=50, radius=2., power=2):
        self.height = h
        self.width = w
        self.niter = niter
        self.radius = radius
        self.power = power
    def __call__(self, xstart, xend, ystart, yend):
        self.x = np.linspace(xstart, xend, self.width)
        self.y = np.linspace(ystart, yend, self.height).reshape(-1, 1)
        c = self.x + 1.0j * self.y
        threshold_time = np.zeros((self.height, self.width))
        z = np.zeros(threshold_time.shape, dtype=np.complex)
        mask = np.ones(threshold_time.shape, dtype=np.bool)
        for i in range(self.niter):
            z[mask] = z[mask]**self.power + c[mask]
            mask = (np.abs(z) < self.radius)
        threshold_time += mask
        return threshold_time
    def ax_update(self, ax):
        ax.set_autoscale_on(False)  # Otherwise, infinite loop
        # Get the number of points from the number of pixels in the window
        dims = ax.axesPatch.get_window_extent().bounds
        self.width = int(dims[2] + 0.5)
        self.height = int(dims[2] + 0.5)
        # Get the range for the new area
        xstart, ystart, xdelta, ydelta = ax.viewLim.bounds
        xend = xstart + xdelta
        yend = ystart + ydelta
        # Update the image object with our new data and extent
        im = ax.images[-1]
        im.set_data(self.__call__(xstart, xend, ystart, yend))
        im.set_extent((xstart, xend, ystart, yend))
        ax.figure.canvas.draw_idle()
md = MandelbrotDisplay()
Z = md(-2., 0.5, -1.25, 1.25)
fig1, (ax1, ax2) = plt.subplots(1, 2)
ax1.imshow(Z, origin='lower', extent=(md.x.min(), md.x.max(), md.y.min(), md.y.max()))
ax2.imshow(Z, origin='lower', extent=(md.x.min(), md.x.max(), md.y.min(), md.y.max()))
rect = UpdatingRect([0, 0], 0, 0, facecolor='None', edgecolor='black', linewidth=1.0)
rect.set_bounds(ax2.viewLim.bounds)
ax1.add_patch(rect)

# Connect for changing the view limits
ax2.callbacks.connect('xlim_changed', rect)
ax2.callbacks.connect('ylim_changed', rect)

ax2.callbacks.connect('xlim_changed', md.ax_update)
ax2.callbacks.connect('ylim_changed', md.ax_update)
ax2.set_title("Zoom here")

plt.show()
x, y = event.xdata, event.ydata
axzoom.set_xlim(x - 0.1, x + 0.1)
axzoom.set_ylim(y - 0.1, y + 0.1)
figzoom.canvas.draw()

figsrc.canvas.mpl_connect('button_press_event', onpress)
show()
93.1 frontpage example code: plot_3D.py

[source code]

```python
"""
====================
Frontpage 3D example
====================
This example reproduces the frontpage 3D example.
"""

from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cbook
from matplotlib import cm
import matplotlib.pyplot as plt
import numpy as np

filename = cbook.get_sample_data('jacksboro_fault_dem.npz', asfileobj=False)
with np.load(filename) as dem:
    z = dem['elevation']
    nrows, ncols = z.shape
    x = np.linspace(dem['xmin'], dem['xmax'], ncols)
    y = np.linspace(dem['ymin'], dem['ymax'], nrows)
    x, y = np.meshgrid(x, y)

    region = np.s_[5:50, 5:50]
    x, y, z = x[region], y[region], z[region]

    fig, ax = plt.subplots(subplot_kw=dict(projection='3d'))

    ls = LightSource(270, 45)
    # To use a custom hillshading mode, override the built-in shading and pass
    # in the rgb colors of the shaded surface calculated from "shade".
    rgb = ls.shade(z, cmap=cm.gist_earth, vert_exag=0.1, blend_mode='soft')
    surf = ax.plot_surface(x, y, z, rstride=1, cstride=1, facecolors=rgb,
                           linewidth=0, antialiased=False, shade=False)
    ax.set_xticks([])
```
import matplotlib.pyplot as plt
import numpy as np
from matplotlib import mlab, cm

extent = (-3, 3, -3, 3)
delta = 0.5
x = np.arange(-3.0, 4.001, delta)
y = np.arange(-4.0, 3.001, delta)
X, Y = np.meshgrid(x, y)
Z1 = mlab.bivariate_normal(X, Y, 1.0, 1.0, 0.0, -0.5)
Z2 = mlab.bivariate_normal(X, Y, 1.5, 0.5, 1, 1)
Z = (Z1 - Z2) * 10

levels = np.linspace(-2.0, 1.601, 40)
norm = cm.colors.Normalize(vmax=abs(Z).max(), vmin=-abs(Z).max())

fig, ax = plt.subplots()
cset1 = ax.contourf(X, Y, Z, levels, norm=norm)
ax.set_xlim(-3, 3)
ax.set_ylim(-3, 3)
ax.set_xticks([])
ax.set_yticks([])
fig.savefig("contour_frontpage.png", dpi=25)  # results in 160x120 px image

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
93.3  frontpage example code: plot_histogram.py

[source code]

""
============================================================
Frontpage histogram example
============================================================

This example reproduces the frontpage histogram example.
""

import matplotlib.pyplot as plt
import numpy as np

random_state = np.random.RandomState(19680801)
X = random_state.randn(10000)

fig, ax = plt.subplots()
ax.hist(X, bins=25, normed=True)
ax.plot(x, 1 / np.sqrt(2*np.pi) * np.exp(-(x**2)/2), linewidth=4)
ax.set_xticks([])
ax.set_yticks([])
fig.savefig("histogram_frontpage.png", dpi=25)  # results in 160x120 px image

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

93.4  frontpage example code: plot_membrane.py

[source code]

""
============================================================
Frontpage plot example
============================================================

This example reproduces the frontpage simple plot example.
""

import matplotlib.pyplot as plt
import matplotlib.cbook as cbook
import numpy as np

datafile = cbook.get_sample_data("membrane.dat", asfileobj=False)
x = np.fromfile(datafile, np.float32)
# 0.0005 is the sample interval
fig, ax = plt.subplots()
ax.plot(x, linewidth=4)
ax.set_xlim(5000, 6000)
ax.set_ylim(-0.6, 0.1)
ax.set_xticks([])
ax.set_yticks([])
fig.savefig("membrane_frontpage.png", dpi=25)  # results in 160x120 px image

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
94.1 images_contours_and_fields example code: contourf_log.py

```python
import matplotlib.pyplot as plt
import numpy as np
from numpy import ma
from matplotlib import colors, ticker, cm
from matplotlib.mlab import bivariate_normal

# Demonstrate use of a log color scale in contourf

import matplotlib.pyplot as plt
import numpy as np
from numpy import ma
from matplotlib import colors, ticker, cm
from matplotlib.mlab import bivariate_normal
```
```python
N = 100
x = np.linspace(-3.0, 3.0, N)
y = np.linspace(-2.0, 2.0, N)

X, Y = np.meshgrid(x, y)

# A low hump with a spike coming out of the top right.
# Needs to have z/colour axis on a log scale so we see both hump and spike.
# linear scale only shows the spike.
z = (bivariate_normal(X, Y, 0.1, 0.2, 1.0, 1.0)
    + 0.1 * bivariate_normal(X, Y, 1.0, 1.0, 0.0, 0.0))

# Put in some negative values (lower left corner) to cause trouble with logs:
z[:5, :5] = -1

# The following is not strictly essential, but it will eliminate
# a warning. Comment it out to see the warning.
z = ma.masked_where(z <= 0, z)

# Automatic selection of levels works; setting the
# log locator tells contourf to use a log scale:
fig, ax = plt.subplots()
cs = ax.contourf(X, Y, z, locator=ticker.LogLocator(), cmap=cm.PuBu_r)

# Alternatively, you can manually set the levels
# and the norm:
#lev_exp = np.arange(np.floor(np.log10(z.min()))-1),
#    np.ceil(np.log10(z.max())+1))
#levs = np.power(10, lev_exp)
#cs = P.contourf(X, Y, z, levs, norm=colors.LogNorm())

# The 'extend' kwarg does not work yet with a log scale.
cbar = fig.colorbar(cs)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
94.2 images_contours_and_fields example code: image_demo.py

Simple demo of the imshow function.

```python
import matplotlib.pyplot as plt
import matplotlib.cbook as cbook

image_file = cbook.get_sample_data('ada.png')
image = plt.imread(image_file)

fig, ax = plt.subplots()
ax.imshow(image)
ax.axis('off')  # clear x- and y-axes
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Demo of image that's been clipped by a circular patch.

```python
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import matplotlib.cbook as cbook

image_file = cbook.get_sample_data('grace_hopper.png')
image = plt.imread(image_file)

fig, ax = plt.subplots()
im = ax.imshow(image)
patch = patches.Circle((260, 200), radius=200, transform=ax.transData)
im.set_clip_path(patch)
ax.axis('off')
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Show all different interpolation methods for imshow

```python
import matplotlib.pyplot as plt
import numpy as np

# from the docs:
# If interpolation is None, default to rc image.interpolation. See also
# the filternorm and filterrad parameters. If interpolation is 'none', then
# no interpolation is performed on the Agg, ps and pdf backends. Other
# backends will fall back to 'nearest'.
#
# http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.imshow

methods = [None, 'none', 'nearest', 'bilinear', 'bicubic', 'spline16',
           'spline36', 'hanning', 'hamming', 'hermite', 'kaiser', 'quadric',
           'catrom', 'gaussian', 'bessel', 'mitchell', 'sinc', 'lanczos']

np.random.seed(0)
grid = np.random.rand(4, 4)
fig, axes = plt.subplots(3, 6, figsize=(12, 6),
                         subplot_kw={'xticks': [], 'yticks': []})
fig.subplots_adjust(hspace=0.3, wspace=0.05)

fig.subplots_adjust(hspace=0.3, wspace=0.05)
```
for ax, interp_method in zip(axes.flat, methods):
    ax.imshow(grid, interpolation=interp_method, cmap='viridis')
    ax.set_title(interp_method)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

94.5 images_contours_and_fields example code: pcolormesh_levels.py

import matplotlib.pyplot as plt
from matplotlib.colors import BoundaryNorm
from matplotlib.ticker import MaxNLocator
import numpy as np

""
Shows how to combine Normalization and Colormap instances to draw "levels" in pcolor, pcolormesh and imshow type plots in a similar way to the levels keyword argument to contour/contourf.
""

import matplotlib.pyplot as plt
from matplotlib.colors import BoundaryNorm
from matplotlib.ticker import MaxNLocator
import numpy as np
# make these smaller to increase the resolution
dx, dy = 0.05, 0.05

# generate 2 2d grids for the x & y bounds
y, x = np.mgrid[slice(1, 5 + dy, dy),
                slice(1, 5 + dx, dx)]

z = np.sin(x)**10 + np.cos(10 + y*x) * np.cos(x)

# x and y are bounds, so z should be the value *inside* those bounds.
# Therefore, remove the last value from the z array.
z = z[:-1, :-1]
levels = MaxNLocator(nbins=15).tick_values(z.min(), z.max())

# pick the desired colormap, sensible levels, and define a normalization
# instance which takes data values and translates those into levels.
cmap = plt.get_cmap('PiYG')
norm = BoundaryNorm(levels, ncolors=cmap.N, clip=True)

fig, (ax0, ax1) = plt.subplots(nrows=2)

im = ax0.pcolormesh(x, y, z, cmap=cmap, norm=norm)
fig.colorbar(im, ax=ax0)
ax0.set_title('pcolormesh with levels')

# contours are *point* based plots, so convert our bound into point
# centers
cf = ax1.contourf(x[:-1, :-1] + dx/2.,
                  y[:-1, :-1] + dy/2., z, levels=levels,
                  cmap=cmap)
fig.colorbar(cf, ax=ax1)
ax1.set_title('contourf with levels')

# adjust spacing between subplots so `ax1` title and `ax0` tick labels
# don't overlap
fig.tight_layout()

plt.show()
94.6 images_contours_and_fields example code: streamplot_demo_features.py
Demo of the `streamplot` function.

A streamplot, or streamline plot, is used to display 2D vector fields. This example shows a few features of the stream plot function:

* Varying the color along a streamline.
* Varying the density of streamlines.
* Varying the line width along a stream line.

```python
import numpy as np
import matplotlib.pyplot as plt

Y, X = np.mgrid[-3:3:100j, -3:3:100j]
U = -1 - X**2 + Y
V = 1 + X - Y**2
speed = np.sqrt(U*U + V*V)

fig0, ax0 = plt.subplots()
strm = ax0.streamplot(X, Y, U, V, color=U, linewidth=2, cmap=plt.cm.autumn)
fig0.colorbar(strm.lines)

fig1, (ax1, ax2) = plt.subplots(ncols=2)
ax1.streamplot(X, Y, U, V, density=[0.5, 1])
ax1.streamplot(X, Y, U, V, linewidth=5*speed / speed.max())
```

94.6. images_contours_and_fields example code: streamplot_demo_features.py
Demo of the streamplot function with masking.

This example shows how streamlines created by the streamplot function skips masked regions and NaN values.

```python
import numpy as np
import matplotlib.pyplot as plt

w = 3
Y, X = np.mgrid[-w:w:100j, -w:w:100j]
U = -1 - X**2 + Y
V = 1 + X - Y**2
```

94.7 images_contours_and_fields example code: streamplot_demo_masking.py
speed = np.sqrt(U*U + V*V)

mask = np.zeros(U.shape, dtype=bool)
mask[40:60, 40:60] = True
U[:20, :20] = np.nan
U = np.ma.array(U, mask=mask)

fig, ax = plt.subplots()
ax.streamplot(X, Y, U, V, color='r')
ax.imshow(~mask, extent=(-w, w, -w, w), alpha=0.5, interpolation='nearest', cmap=plt.cm.gray)

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

94.8 images_contours_and_fields example code: streamplot_demo_start_points.py

Demo of the `streamplot` function.
A streamplot, or streamline plot, is used to display 2D vector fields. This example shows a few features of the stream plot function:

* Varying the color along a streamline.
* Varying the density of streamlines.
* Varying the line width along a stream line.

```python
import numpy as np
import matplotlib.pyplot as plt

X, Y = (np.linspace(-3, 3, 100),
       np.linspace(-3, 3, 100))

U, V = np.mgrid[-3:3:100j, 0:0:100j]

seed_points = np.array([[-2, 0, 1], [-2, 0, 1]])

fig0, ax0 = plt.subplots()
strm = ax0.streamplot(X, Y, U, V, color=U, linewidth=2,
                      cmap=plt.cm.autumn, start_points=seed_points.T)
fig0.colorbar(strm.lines)
ax0.plot(seed_points[0], seed_points[1], 'bo')
ax0.axis((-3, 3, -3, 3))
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
How fast do you want to go today?

```
# Horizontal bar chart

This example showcases a simple horizontal bar chart.
```

---

CHAPTER

NINETYFIVE

LINES_BARS_AND_MARKERS EXAMPLES

95.1 lines_bars_and_markers example code: barh_demo.py
import matplotlib.pyplot as plt
plt.rcdefaults()
import numpy as np
import matplotlib.pyplot as plt

plt.rcdefaults()
fig, ax = plt.subplots()

# Example data
people = ('Tom', 'Dick', 'Harry', 'Slim', 'Jim')
y_pos = np.arange(len(people))
performance = 3 + 10 * np.random.rand(len(people))
error = np.random.rand(len(people))

ax.barh(y_pos, performance, xerr=error, align='center',
        color='green', ecolor='black')
ax.set_yticks(y_pos)
ax.set_yticklabels(people)
ax.invert_yaxis()  # labels read top-to-bottom
ax.set_xlabel('Performance')
ax.set_title('How fast do you want to go today?')

plt.show()
This example showcases the most basic fill plot a user can do with matplotlib.

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 1, 500)
y = np.sin(4 * np.pi * x) * np.exp(-5 * x)

fig, ax = plt.subplots()

ax.fill(x, y, zorder=10)
ax.grid(True, zorder=5)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
In addition to the basic fill plot, this demo shows a few optional features:

* Multiple curves with a single command.
* Setting the fill color.
* Setting the opacity (alpha value).

```python
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 2 * np.pi, 500)
y1 = np.sin(x)
y2 = np.sin(3 * x)

fig, ax = plt.subplots()
ax.fill(x, y1, 'b', x, y2, 'r', alpha=0.3)
plt.show()
```
A simple plot with a custom dashed line

A Line object’s `set_dashes` method allows you to specify dashes with a series of on/off lengths (in points).

```python
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 10, 500)
dashes = [10, 5, 100, 5]  # 10 points on, 5 off, 100 on, 5 off

fig, ax = plt.subplots()
line1, = ax.plot(x, np.sin(x), '--', linewidth=2,
                 label='Dashes set retroactively')
line1.set_dashes(dashes)
```

95.4 lines_bars_and_markers example code: line_demo_dash_control.py
```python
line2, = ax.plot(x, -1 * np.sin(x), dashes=[30, 5, 10, 5],
                  label='Dashes set proactively')
ax.legend(loc='lower right')
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

95.5 lines_bars_and_markers example code: line_styles_reference.py

```python
line styles

'--'

'-.-'

'-. '

':'

''

================================
Line-style reference
================================

Reference for line-styles included with Matplotlib.
```

```python
import numpy as np
import matplotlib.pyplot as plt
```
color = 'cornflowerblue'
points = np.ones(5)  # Draw 5 points for each line
text_style = dict(horiztonalalignment='right', verticalalignment='center',
                  fontsize=12, fontdict={'family': 'monospace'})

def format_axes(ax):
    ax.margins(0.2)
    ax.set_axis_off()

def nice_repr(text):
    return repr(text).lstrip('u')

# Plot all line styles.
fig, ax = plt.subplots()
linestyles = ['-', '--', '-.', ':']
for y, linestyle in enumerate(linestyles):
    ax.text(-0.1, y, nice_repr(linestyle), **text_style)
    ax.plot(y * points, linestyle=linestyle, color=color, linewidth=3)
    format_axes(ax)
    ax.set_title('line styles')

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
This example showcases different linestyles copying those of TikZ/PGF.

```python
import numpy as np
import matplotlib.pyplot as plt
from collections import OrderedDict
from matplotlib.transforms import blended_transform_factory

linestyles = OrderedDict(
    [('solid', (0, ())),
     ('loosely dotted', (0, (1, 10))),
     ('dotted', (0, (1, 5))),
     ('densely dotted', (0, (1, 1))),(0, (5, 10)),
     ('loosely dashed', (0, (5, 10))),
     ('dashed', (0, (5, 5))),
     ('densely dashed', (0, (5, 1))),(0, (3, 10, 1, 10))),(0, (3, 5, 1, 5))),(0, (3, 1, 1, 1))),(0, (3, 10, 1, 10))),(0, (3, 5, 1, 5))),(0, (3, 1, 1, 1))),(0, (3, 10, 1, 10))),(0, (3, 5, 1, 5))),(0, (3, 1, 1, 1))]
```
plt.figure(figsize=(10, 6))
ax = plt.subplot(1, 1, 1)

X, Y = np.linspace(0, 100, 10), np.zeros(10)
for i, (name, linestyle) in enumerate(linestyles.items()):
    ax.plot(X, Y+i, linestyle=linestyle, linewidth=1.5, color='black')

ax.set_ylim(-0.5, len(linestyles)-0.5)
plt.yticks(np.arange(len(linestyles)), linestyles.keys())
plt.xticks([])

# For each line style, add a text annotation with a small offset from
# the reference point (0 in Axes coords, y tick value in Data coords).
reference_transform = blended_transform_factory(ax.transAxes, ax.transData)
for i, (name, linestyle) in enumerate(linestyles.items()):
    ax.annotate(str(linestyle), xy=(0.0, i), xycoords=reference_transform,
                xytext=(-6, -12), textcoords='offset points',
                color="blue", fontsize=8, ha="right", family="monospace")

plt.tight_layout()
plt.show()
fill style

'none'  
'top'  
'bottom'  
'right'  
'left'  
'full'

Reference for marker fill-styles included with Matplotlib.

```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.lines import Line2D

points = np.ones(5)  # Draw 3 points for each line
text_style = dict(horizontalalignment='right', verticalalignment='center',
                   fontsize=12, fontdict={'family': 'monospace'})
marker_style = dict(color='cornflowerblue', linestyle=':', marker='o',
                     markersize=15, markerfacecoloralt='gray')

def format_axes(ax):
    ax.margins(0.2)
    ax.set_axis_off()
```
def nice_repr(text):
    return repr(text).lstrip('u')

fig, ax = plt.subplots()

# Plot all fill styles.
for y, fill_style in enumerate(Line2D.fillStyles):
    ax.text(-0.5, y, nice_repr(fill_style), **text_style)
    ax.plot(y * points, fillstyle=fill_style, **marker_style)
    format_axes(ax)
ax.set_title('fill style')
plt.show()
filled markers

'''
================================
Filled and unfilled-marker types
================================

Reference for filled- and unfilled-marker types included with Matplotlib.
'''

from six import iteritems
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.lines import Line2D

points = np.ones(3)  # Draw 3 points for each line

# Text style
text_style = dict(horizontalalignment='right', verticalalignment='center',
                   fontsize=12, fontdict={'family': 'monospace'})

# Marker style
marker_style = dict(linestyle=':', color='cornflowerblue', markersize=10)

def format_axes(ax):
    ax.margins(0.2)
    ax.set_axis_off()

def nice_repr(text):
    return repr(text).lstrip('u')
```python
def split_list(a_list):
    i_half = len(a_list) // 2
    return (a_list[:i_half], a_list[i_half:]),

# Plot all un-filled markers
# --------------------------

fig, axes = plt.subplots(ncols=2)

# Filter out filled markers and marker settings that do nothing.
# We use iteritems from six to make sure that we get an iterator
# in both python 2 and 3
unfilled_markers = [m for m, func in iteritems(Line2D.markers)
    if func != 'nothing' and m not in Line2D.filled_markers]

# Reverse-sort for pretty. We use our own sort key which is essentially
# a python3 compatible reimplementation of python2 sort.
unfilled_markers = sorted(unfilled_markers,
    key=lambda x: (str(type(x)), str(x)))[::-1]

for ax, markers in zip(axes, split_list(unfilled_markers)):
    for y, marker in enumerate(markers):
        ax.text(-0.5, y, nice_repr(marker), **text_style)
        ax.plot(y * points, marker=marker, **marker_style)
    format_axes(ax)

fig.suptitle('un-filled markers', fontsize=14)

# Plot all filled markers.
# ------------------------

fig, axes = plt.subplots(ncols=2)

for ax, markers in zip(axes, split_list(Line2D.filled_markers)):
    for y, marker in enumerate(markers):
        ax.text(-0.5, y, nice_repr(marker), **text_style)
        ax.plot(y * points, marker=marker, **marker_style)
        format_axes(ax)

fig.suptitle('filled markers', fontsize=14)

plt.show()```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
95.9 lines_bars_and_markers example code: scatter_with_legend.py

```python
import matplotlib.pyplot as plt
from numpy.random import rand

fig, ax = plt.subplots()
for color in ['red', 'green', 'blue']:
    n = 750
    x, y = rand(2, n)
    scale = 200.0 * rand(n)
    ax.scatter(x, y, c=color, s=scale, label=color,
               alpha=0.3, edgecolors='none')

ax.legend()
ax.grid(True)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
96.1 misc example code: contour_manual.py

User-specified contours
Example of displaying your own contour lines and polygons using ContourSet.

```python
import matplotlib.pyplot as plt
from matplotlib.contour import ContourSet
import matplotlib.cm as cm

# Contour lines for each level are a list/tuple of polygons.
lines0 = [[[0, 0], [0, 4]]]
lines1 = [[[2, 0], [1, 2], [1, 3]]]
lines2 = [[[3, 0], [3, 2]], [[3, 3], [3, 4]]]  # Note two lines.

# Filled contours between two levels are also a list/tuple of polygons.
# Points can be ordered clockwise or anticlockwise.
filled01 = [[[0, 0], [0, 4], [1, 3], [1, 2], [2, 0]]]
filled12 = [[[2, 0], [3, 0], [3, 2], [1, 3], [1, 2]],  # Note two polygons.
            [[1, 4], [3, 4], [3, 3]]]

plt.figure()

# Filled contours using filled=True.
cs = ContourSet(plt.gca(), [0, 1, 2], [filled01, filled12], filled=True, cmap=cm.bone)
cbar = plt.colorbar(cs)

# Contour lines (non-filled).
```

User specified filled contours with holes
lines = ContourSet(plt.gca(), [0, 1, 2], [lines0, lines1, lines2], cmap=cm.cool, linewidths=3)
cbar.add_lines(lines)

plt.axis([-0.5, 3.5, -0.5, 4.5])
plt.title('User-specified contours')

# Multiple filled contour lines can be specified in a single list of polygon vertices along with a list of vertex kinds (code types) as described in the Path class. This is particularly useful for polygons with holes. Here a code type of 1 is a MOVETO, and 2 is a LINETO.

plt.figure()
filled01 = [[[0, 0], [3, 0], [3, 3], [0, 3], [1, 1], [1, 2], [2, 2], [2, 1]]
kinds01 = [[1, 2, 2, 2, 1, 2, 2, 2]]
cs = ContourSet(plt.gca(), [0, 1], [filled01], [kinds01], filled=True)
cbar = plt.colorbar(cs)

plt.axis([-0.5, 3.5, -0.5, 3.5])
plt.title('User specified filled contours with holes')
plt.show()
# make a charname to charcode and glyphind dictionary
coded = {}
glyphd = {}
for ccode, glyphind in codes:
    name = font.get_glyph_name(glyphind)
    coded[name] = ccode
glyphd[name] = glyphind

code = coded['A']
glyph = font.load_char(code)
# print(glyph.bbox)
print(glyphd['A'], glyphd['V'], coded['A'], coded['V'])
print('AV', font.get_kerning(glyphd['A'], glyphd['V'], KERNING_DEFAULT))
print('AV', font.get_kerning(glyphd['A'], glyphd['V'], KERNING_UNFITTED))
print('AV', font.get_kerning(glyphd['A'], glyphd['V'], KERNING_UNSCALED))
print('AV', font.get_kerning(glyphd['A'], glyphd['T'], KERNING_UNSCALED))

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

96.3 misc example code: ftface_props.py

```python
from __future__ import print_function

""
This is a demo script to show you how to use all the properties of an
FT2Font object. These describe global font properties. For
individual character metrics, use the Glyph object, as returned by
load_char
""

import matplotlib
import matplotlib.ft2font as ft

#fname = '/usr/local/share/matplotlib/VeraIt.ttf'
fname = matplotlib.get_data_path() + '/fonts/ttf/DejaVuSans-Oblique.ttf'
#fname = '/usr/local/share/matplotlib/cmr10.ttf'

font = ft.FT2Font(fname)

print('Num faces :', font.num_faces)      # number of faces in file
print('Num glyphs :', font.num_glyphs)    # number of glyphs in the face
print('Family name :', font.family_name)  # face family name
print('Style name :', font.style_name)    # face style name
print('PS name :', font.postscript_name)  # the postscript name
print('Num fixed :', font.num_fixed_sizes) # number of embedded bitmap in face

# the following are only available if face.scalable
if font.scalable:
    # the face global bounding box (xmin, ymin, xmax, ymax)
    print('Bbox :', font.bbox)
```

2254 Chapter 96. misc Examples
# number of font units covered by the EM
print('EM :', font.units_per_EM)

# the ascender in 26.6 units
print('Ascender :', font.ascender)

# the descender in 26.6 units
print('Descender :', font.descender)

# the height in 26.6 units
print('Height :', font.height)

# maximum horizontal cursor advance
print('Max adv width :', font.max_advance_width)

# same for vertical layout
print('Max adv height :', font.max_advance_height)

# vertical position of the underline bar
print('Underline pos :', font.underline_position)

# vertical thickness of the underline
print('Underline thickness :', font.underline_thickness)

for style in ('Italic', 'Bold', 'Scalable', 'Fixed sizes', 'Fixed width', 'SFNT', 'Horizontal', 'Vertical', 'Kerning', 'Fast glyphs', 'Multiple masters', 'Glyph names', 'External stream'):
    bitpos = getattr(ft, style.replace(' ', '_').upper()) - 1
    print('%-17s: %s, bool(font.style_flags & (1 << bitpos)))

print(dir(font))

print(font.get_kerning)

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

96.4 misc example code: image_thumbnail.py

[source code]

# -*- noplot -*-

"""
You can use matplotlib to generate thumbnails from existing images.
matplotlib natively supports PNG files on the input side, and other
image types transparently if your have PIL installed
"""

from __future__ import print_function

96.4. misc example code: image_thumbnail.py
import sys
import os
import glob
import matplotlib.image as image

if len(sys.argv) != 2:
    print('Usage: python %s IMAGEDIR' % __file__)
    raise SystemExit

indir = sys.argv[1]
if not os.path.isdir(indir):
    print('Could not find input directory "%s"' % indir)
    raise SystemExit

outdir = 'thumbs'
if not os.path.exists(outdir):
    os.makedirs(outdir)

for fname in glob.glob(os.path.join(indir, '*.png')):
    basedir, basename = os.path.split(fname)
    outfile = os.path.join(outdir, basename)
    fig = image.thumbnail(fname, outfile, scale=0.15)
    print('saved thumbnail of %s to %s' % (fname, outfile))

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

96.5 misc example code: multiprocess.py

from __future__ import print_function
import time
from multiprocessing import Process, Pipe
import numpy as np
import matplotlib
matplotlib.use('GTKAgg')
import matplotlib.pyplot as plt
import gobject

class ProcessPlotter(object):
    def __init__(self):
        self.x = []
        self.y = []
```python
def terminate(self):
    plt.close('all')

def poll_draw(self):
    def call_back():
        while 1:
            if not self.pipe.poll():
                break

            command = self.pipe.recv()

            if command is None:
                self.terminate()
                return False

            else:
                self.x.append(command[0])
                self.y.append(command[1])
                self.ax.plot(self.x, self.y, 'ro')

                self.fig.canvas.draw()

            return True

    return call_back

def __call__(self, pipe):
    print('starting plotter...')

    self.pipe = pipe
    self.fig, self.ax = plt.subplots()
    self.gid = gobject.timeout_add(1000, self.poll_draw())

    print('...done')
    plt.show()

class NBPlot(object):
    def __init__(self):
        self.plot_pipe, plotter_pipe = Pipe()
        self.plotter = ProcessPlotter()
        self.plot_process = Process(target=self.plotter,
                                     args=(plotter_pipe,))
        self.plot_process.daemon = True
        self.plot_process.start()

    def plot(self, finished=False):
        send = self.plot_pipe.send
        if finished:
            send(None)
        else:
            data = np.random.random(2)
```

96.5. misc example code: multiprocess.py
send(data)

def main():
    pl = NBPlot()
    for ii in range(10):
        pl.plot()
        time.sleep(0.5)
    try:
        input = raw_input
    except NameError:
        pass
    input('press Enter...')
    pl.plot(finished=True)

if __name__ == '__main__':
    main()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

96.6 misc example code: rasterization_demo.py
```python
import numpy as np
import matplotlib.pyplot as plt

d = np.arange(100).reshape(10, 10)
x, y = np.meshgrid(np.arange(11), np.arange(11))

theta = 0.25*np.pi
xx = x*np.cos(theta) - y*np.sin(theta)
yy = x*np.sin(theta) + y*np.cos(theta)

fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2)
ax1.set_aspect(1)
ax1.pcolormesh(xx, yy, d)
ax1.set_title("No Rasterization")

ax2.set_aspect(1)
ax2.set_title("Rasterization")
m = ax2.pcolormesh(xx, yy, d)
m.set_rasterized(True)

ax3.set_aspect(1)
ax3.pcolormesh(xx, yy, d)
ax3.text(0.5, 0.5, "Text", alpha=0.2,
       va="center", ha="center", size=50, transform=ax3.transAxes)
ax3.set_title("No Rasterization")

ax4.set_aspect(1)
m = ax4.pcolormesh(xx, yy, d)
m.set_zorder(-20)
ax4.text(0.5, 0.5, "Text", alpha=0.2,
       zorder=-15,
       va="center", ha="center", size=50, transform=ax4.transAxes)
ax4.set_rasterization_zorder(-10)
ax4.set_title("Rasterization z$<-10$")

# ax2.title.set_rasterized(True) # should display a warning
plt.savefig("test_rasterization.pdf", dpi=150)
plt.savefig("test_rasterization.eps", dpi=150)
if not plt.rcParams["text.usetex"]:
    plt.savefig("test_rasterization.svg", dpi=150)
    # svg backend currently ignores the dpi
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

96.6. misc example code: rasterization_demo.py 2259
96.7 misc example code: rc_traits.py

[source code]

```python
# -*- noplot -*-
# Here is some example code showing how to define some representative
# rc properties and construct a matplotlib artist using traits.
# matplotlib does not ship with enthought.traits, so you will need to
# install it separately.

from __future__ import print_function
import sys
import os
import re
import traits.api as traits
from matplotlib.cbook import is_string_like
from matplotlib.artist import Artist
doprint = True
flexible_true_trait = traits.Trait(True,
       {'true': True, 't': True, 'yes': True, 'y': True, 'on': True, True: True,
        'false': False, 'f': False, 'no': False, 'n': False, 'off': False, False: False
       })
flexible_false_trait = traits.Trait(False, flexible_true_trait)

colors = {
    'c': '#00bfbf',
    'b': '#0000ff',
    'g': '#008000',
    'k': '#000000',
    'm': '#bf00bf',
    'r': '#ff0000',
    'w': '#ffffff',
    'y': '#bfbf00',
    'gold': '#FFD700',
    'peachpuff': '#FFDAB9',
    'navajowhite': '#FFDEAD',
}

def hex2color(s):
    "Convert hex string (like html uses, eg, #efefef) to a r,g,b tuple"
    return tuple([int(n, 16)/255.0 for n in (s[1:3], s[3:5], s[5:7])])

class RGBA(traits.HasTraits):
    # r,g,b,a in the range 0-1 with default color 0,0,0,1 (black)
    r = traits.Range(0., 1., 0.)
    g = traits.Range(0., 1., 0.)
    b = traits.Range(0., 1., 0.)
    a = traits.Range(0., 1., 1.)
```

2260 Chapter 96. misc Examples
```python
def __init__(self, r=0., g=0., b=0., a=1.):
    self.r = r
    self.g = g
    self.b = b
    self.a = a

def __repr__(self):
    return '
r,g,b,a = (%1.2f, %1.2f, %1.2f, %1.2f)'
    (self.r, self.g, self.b, self.a)

def tuple_to_rgba(ob, name, val):
    tup = [float(x) for x in val]
    if len(tup) == 3:
        r, g, b = tup
        return RGBA(r, g, b)
    elif len(tup) == 4:
        r, g, b, a = tup
        return RGBA(r, g, b, a)
    else:
        raise ValueError
tuple_to_rgba.info = 'a RGB or RGBA tuple of floats'

def hex_to_rgba(ob, name, val):
    rgx = re.compile('^#([0-9A-Fa-f]{6})$')
    if not is_string_like(val):
        raise TypeError
    if rgx.match(val) is None:
        raise ValueError
    r, g, b = hex2color(val)
    return RGBA(r, g, b, 1.0)
hex_to_rgba.info = 'a hex color string'

def colorname_to_rgba(ob, name, val):
    hex = colors[val.lower()]
    r, g, b = hex2color(hex)
    return RGBA(r, g, b, 1.0)
colorname_to_rgba.info = 'a named color'

def float_to_rgba(ob, name, val):
    val = float(val)
    return RGBA(val, val, val, 1.0)
float_to_rgba.info = 'a grayscale intensity'

Color = traits.Trait(RGBA(), float_to_rgba, colorname_to_rgba, RGBA, hex_to_rgba, tuple_to_rgba)
```

96.7. misc example code: rc_traits.py
def file_exists(ob, name, val):
    fh = file(val, 'r')
    return val

linestyles = ('-', '--', '-.', ':', 'steps', 'None')
TICKLEFT, TICKRIGHT, TICKUP, TICKDOWN = range(4)
linemarkers = (None, '.', ',', 'o', '^', '<', 'v', '>', 's', '+', 'x', 'd', 'D', '|', '_', 'h', 'H', 'p', '1', '2', '3', '4', TICKLEFT, TICKRIGHT, TICKUP, TICKDOWN, 'None')

class LineRC(traits.HasTraits):
    linewidth = traits.Float(0.5)
    linestyle = traits.Trait(*linestyles)
    color = Color
    marker = traits.Trait(*linemarkers)
    markerfacecolor = Color
    markeredgecolor = Color
    markeredgewidth = traits.Float(0.5)
    markersize = traits.Float(6)
    antialiased = flexible_true_trait
    data_clipping = flexible_false_trait

class PatchRC(traits.HasTraits):
    linewidth = traits.Float(1.0)
    facecolor = Color
    edgecolor = Color
    antialiased = flexible_true_trait

timezones = ['UTC', 'US/Central', 'ES/Eastern'] # fixme: and many more
backends = ('GTKAgg', 'Cairo', 'GDK', 'GTK', 'Agg', 'GTKCairo', 'PS', 'SVG', 'Template', 'TkAgg', 'WX')

class RC(traits.HasTraits):
    backend = traits.Trait(*backends)
    interactive = flexible_false_trait
    toolbar = traits.Trait('toolbar2', 'classic', None)
    timezone = traits.Trait(*timezones)
    lines = traits.Trait(LineRC())
    patch = traits.Trait(PatchRC())

rc = RC()
rc.lines.color = 'r'
if doprint:
    print('RC')
    rc.print_traits()
    print('RC lines')
    rc.lines.print_traits()
    print('RC patches')
    rc.patch.print_traits()

class Patch(Artist, traits.HasTraits):
    linewidth = traits.Float(0.5)
    facecolor = Color
    fc = facecolor
    edgecolor = Color
    fill = flexible_true_trait

    def __init__(self,
                 edgecolor=None,
                 facecolor=None,
                 linewidth=None,
                 antialiased=None,
                 fill=1,
                 **kwargs)
        Artist.__init__(self)
        if edgecolor is None:
            edgecolor = rc.patch.edgecolor
        if facecolor is None:
            facecolor = rc.patch.facecolor
        if linewidth is None:
            linewidth = rc.patch.linewidth
        if antialiased is None:
            antialiased = rc.patch.antialiased

        self.edgecolor = edgecolor
        self.facecolor = facecolor
        self.linewidth = linewidth
        self.antialiased = antialiased
        self.fill = fill

p = Patch()
p.facecolor = '#bfbf00'
p.edgecolor = 'gold'
p.facecolor = (1, .5, .5, .25)
p.facecolor = 0.25
p.fill = 'f'
p.fill_ = p.fill
print('p.facecolor', type(p.facecolor), p.facecolor)
print('p.fill', type(p.fill), p.fill)
if p.fill_:
    print('fill')
else:
print('no fill')
if doprint:
    print()
    print('Patch')
p.print_traits()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

96.8  misc example code: rec_groupby_demo.py

```python
from __future__ import print_function
import numpy as np
import matplotlib.mlab as mlab
import matplotlib.cbook as cbook
datafile = cbook.get_sample_data('aapl.csv', asfileobj=False)
print('loading', datafile)
r = mlab.csv2rec(datafile)
r.sort()

def daily_return(prices):
    'an array of daily returns from price array'
    g = np.zeros_like(prices)
    g[1:] = (prices[1:] - prices[:-1])/prices[:-1]
    return g

def volume_code(volume):
    'code the continuous volume data categorically'
    ind = np.searchsorted([1e5, 1e6, 5e6, 10e6, 1e7], volume)
    return ind

summaryfuncs = (
    ('date', lambda x: [thisdate.year for thisdate in x], 'years'),
    ('date', lambda x: [thisdate.month for thisdate in x], 'months'),
    ('date', lambda x: [thisdate.weekday() for thisdate in x], 'weekday'),
    ('adj_close', daily_return, 'dreturn'),
    ('volume', volume_code, 'volcode'),
)

rsum = mlab.rec_summarize(r, summaryfuncs)
```

# stats is a list of (dtype_name, function, output_dtype_name).
# rec_groupby will summarize the attribute identified by the
# dtype_name over the groups in the groupby list, and assign the
# result to the output_dtype_name
stats = (
    ('dreturn', len, 'rcnt'),
    ('dreturn', np.mean, 'rmean'),
    ('dreturn', np.median, 'rmedian'),
    ('dreturn', np.std, 'rsigma'),
)

# you can summarize over a single variable, like years or months
print('summary by years')
ry = mlab.rec_groupby(rsum, ('years'), stats)
print(mlab.rec2txt(ry))

print('summary by months')
rm = mlab.rec_groupby(rsum, ('months'), stats)
print(mlab.rec2txt(rm))

# or over multiple variables like years and months
print('summary by year and month')
rym = mlab.rec_groupby(rsum, ('years', 'months'), stats)
print(mlab.rec2txt(rym))

print('summary by volume')
rv = mlab.rec_groupby(rsum, ('volcode'), stats)
print(mlab.rec2txt(rv))

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

96.9 misc example code: rec_join_demo.py

from __future__ import print_function
import numpy as np
import matplotlib.mlab as mlab
import matplotlib.cbook as cbook
datafile = cbook.get_sample_data('aapl.csv', asfileobj=False)
print('loading', datafile)
r = mlab.csv2rec(datafile)
r.sort()
r1 = r[-10:]

# Create a new array
r2 = np.empty(12, dtype=[('date', '|O4'), ('high', np.float),
                         ('marker', np.float)])
r2 = r2.view(np.recarray)
r2.date = r.date[-17:-5]
r2.high = r.high[-17:-5]
r2.marker = np.arange(12)

print("r1:")
print(mlab.rec2txt(r1))
```python
print("r2:")
print(mlab.rec2txt(r2))

defaults = {'marker': -1, 'close': np.nan, 'low': -4444.}

for s in ('inner', 'outer', 'leftouter'):
    rec = mlab.rec_join(["date", "high"], r1, r2,
                        jointype=s, defaults=default)
    print("\n%sjoin :
%" % (s, mlab.rec2txt(rec)))
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

### 96.10 misc example code: sample_data_demo.py

```
""
Grab mpl data from the ~/.matplotlib/sample_data cache if it exists, else
fetch it from GitHub and cache it
""

from __future__ import print_function
import matplotlib.cbook as cbook
import matplotlib.pyplot as plt
fname = cbook.get_sample_data('ada.png', asfileobj=False)
```
print('fname', fname)
im = plt.imread(fname)
plt.imshow(im)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

96.11 misc example code: svg_filter_line.py

"""
Demonstrate SVG filtering effects which might be used with mpl.

Note that the filtering effects are only effective if your svg renderer support it.
"""

from __future__ import print_function
import matplotlib
matplotlib.use("Svg")
import matplotlib.pyplot as plt
import matplotlib.transforms as mtransforms

fig1 = plt.figure()
ax = fig1.add_axes([0.1, 0.1, 0.8, 0.8])

# draw lines
l1, = ax.plot([0.1, 0.5, 0.9], [0.1, 0.9, 0.5], "bo-",
               mec="b", lw=5, ms=10, label="Line 1")

l2, = ax.plot([0.1, 0.5, 0.9], [0.5, 0.2, 0.7], "rs-",
               mec="r", lw=5, ms=10, color="r", label="Line 2")

for l in [l1, l2]:
    # draw shadows with same lines with slight offset and gray colors.
    xx = l.get_xdata()
    yy = l.get_ydata()
    shadow, = ax.plot(xx, yy)
    shadow.update_from(l)

    # adjust color
    shadow.set_color("0.2")

    # adjust zorder of the shadow lines so that it is drawn below the
    # original lines
    shadow.set_zorder(l.get_zorder() - 0.5)

    # offset transform
    ot = mtransforms.offset_copy(l.get_transform(), fig1,
                                 x=4.0, y=-6.0, units='points')

    shadow.set_transform(ot)

    # set the id for a later use
    shadow.set_gid(l.get_label() + "_shadow")

ax.set_xlim(0., 1.)
ax.set_ylim(0., 1.)

# save the figure as a bytes string in the svg format.
from io import BytesIO
f = BytesIO()
plt.savefig(f, format="svg")

import xml.etree.cElementTree as ET

# filter definition for a gaussian blur
filter_def = ""
<defs xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink">
    <filter id="dropshadow" height='1.2' width='1.2'>
```python
<feGaussianBlur result='blur' stdDeviation='3'/>
</filter>
</defs>

# read in the saved svg

tree, xmlid = ET.XMLID(f.getvalue())

# insert the filter definition in the svg dom tree.
tree.insert(0, ET.XML(filter_def))

for l1, l2 in [[l1, l2]:
    # pick up the svg element with given id
    shadow = xmlid[l1.get_label() + '_shadow']
    # apply shadow filter
    shadow.set("filter", 'url(#dropshadow)')

fn = "svg_filter_line.svg"
print("Saving '%s'" % fn)
ET.ElementTree(tree).write(fn)
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Demonstrate SVG filtering effects which might be used with mpl. The pie chart drawing code is borrowed from pie_demo.py

Note that the filtering effects are only effective if your svg renderer support it.

```python
import matplotlib
matplotlib.use("Svg")
import matplotlib.pyplot as plt
```
from matplotlib.patches import Shadow

# make a square figure and axes
fig1 = plt.figure(1, figsize=(6, 6))
ax = fig1.add_axes([0.1, 0.1, 0.8, 0.8])

labels = ['Frogs', 'Hogs', 'Dogs', 'Logs']
fracs = [15, 30, 45, 10]
explode = (0, 0.05, 0, 0)

# We want to draw the shadow for each pie but we will not use "shadow"
# option as it doesn't save the references to the shadow patches.
pies = ax.pie(fracs, explode=explode, labels=labels, autopct='%.1f%%')

for w in pies[0]:
    # set the id with the label.
    w.set_gid(w.get_label())
    # we don't want to draw the edge of the pie
    w.set_ec('none')

for w in pies[0]:
    # create shadow patch
    s = Shadow(w, -0.01, -0.01)
    s.set_gid(w.get_gid() + '_shadow')
    s.set_zorder(w.get_zorder() - 0.1)
    ax.add_patch(s)

# save
from io import BytesIO
f = BytesIO()
plt.savefig(f, format='svg')

import xml.etree.cElementTree as ET

# filter definition for shadow using a gaussian blur
# and lightening effect.
# The lightening filter is copied from http://www.w3.org/TR/SVG/filters.html
# I tested it with Inkscape and Firefox3. "Gaussian blur" is supported
# in both, but the lightening effect only in the Inkscape. Also note
# that, Inkscape's exporting also may not support it.
filter_def = ""
<defs xmlns='http://www.w3.org/2000/svg' xmlns:xlink='http://www.w3.org/1999/xlink'>
    <filter id='dropshadow' height='1.2' width='1.2'>
        <feGaussianBlur result='blur' stdDeviation='2'/>
    </filter>
    <filter id='MyFilter' filterUnits='objectBoundingBox' x='0' y='0' width='1' height='1'..."
<feGaussianBlur in='SourceAlpha' stdDeviation='4%' result='blur'/>
<feOffset in='blur' dx='4%' dy='4%' result='offsetBlur'/>
<feSpecularLighting in='blur' surfaceScale='5' specularConstant='0.75' specularExponent='20' lighting-color='#bbbbbb' result='specOut'>
    <fePointLight x='-5000%' y='-10000%' z='20000%'/>
</feSpecularLighting>
<feComposite in='specOut' in2='SourceAlpha' operator='in' result='specOut'/>
<feComposite in='SourceGraphic' in2='specOut' operator='arithmetic'
    k1='0' k2='1' k3='1' k4='0'/>
</filter>

```
# insert the filter definition in the svg dom tree.
tree.insert(0, ET.XML(filter_def))

for i, pie_name in enumerate(labels):
    pie = xmlid[pie_name]
    pie.set("filter", 'url(#MyFilter)')

    shadow = xmlid[pie_name + "_shadow"]
    shadow.set("filter", 'url(#dropshadow)')
```

fn = "svg_filter_pie.svg"
print("Saving '%s' " % fn)
ET.ElementTree(tree).write(fn)

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Matplotlib, Release 2.0.2

96.13 misc example code: tight_bbox_test.py

Check saved figures for their bboxes

My y-label

tick label

long long tick label
0.0

0.5

1.0

1.5

2.0

from __future__ import print_function
import matplotlib.pyplot as plt
import numpy as np
ax = plt.axes([0.1, 0.3, 0.5, 0.5])
ax.pcolormesh(np.array([[1, 2], [3, 4]]))
plt.yticks([0.5, 1.5], ["long long tick label",
"tick label"])
plt.ylabel("My y-label")
plt.title("Check saved figures for their bboxes")
for ext in ["png", "pdf", "svg", "svgz", "eps"]:
print("saving tight_bbox_test.%s" % (ext,))
plt.savefig("tight_bbox_test.%s" % (ext,), bbox_inches="tight")
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

96.13. misc example code: tight_bbox_test.py

2273


97.1 mplot3d example code: 2dcollections3d_demo.py

```python
# curve in (x,y)
# points in (x,z)
```
from mpl_toolkits.mplot3d import Axes3D
import numpy as np
import matplotlib.pyplot as plt

fig = plt.figure()
alx = fig.gca(projection='3d')

# Plot a sin curve using the x and y axes.
x = np.linspace(0, 1, 100)
y = np.sin(x * 2 * np.pi) / 2 + 0.5
alx.plot(x, y, zs=0, zdir='z', label='curve in (x,y)')

# Plot scatterplot data (20 2D points per colour) on the x and z axes.
colors = ('r', 'g', 'b', 'k')
x = np.random.sample(20*len(colors))
y = np.random.sample(20*len(colors))
c_list = []
for c in colors:
    c_list.append([c]*20)
# By using zdir='y', the y value of these points is fixed to the zs value 0
# and the (x,y) points are plotted on the x and z axes.
alx.scatter(x, y, zs=0, zdir='y', c=c_list, label='points in (x,z)')

# Make legend, set axes limits and labels
alx.legend()
alx.set_xlim(0, 1)
alx.set_ylim(0, 1)
alx.set_zlim(0, 1)
alx.set_xlabel('X')
alx.set_ylabel('Y')
alx.set_zlabel('Z')

# Customize the view angle so it's easier to see that the scatter points lie
# on the plane y=0
alx.view_init(elev=20., azim=-35)

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
**97.2 mplot3d example code: bars3d_demo.py**

```python
""
========================================
Create 2D bar graphs in different planes
========================================

Demonstrates making a 3D plot which has 2D bar graphs projected onto planes $y=0$, $y=1$, etc.
""

```from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
for c, z in zip(['r', 'g', 'b', 'y'], [30, 20, 10, 0]):
    xs = np.arange(20)
    ys = np.random.rand(20)

    # You can provide either a single color or an array. To demonstrate this,
    # the first bar of each set will be colored cyan.
    cs = [c] * len(xs)```
from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
from matplotlib import cm

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
X, Y, Z = axes3d.get_test_data(0.05)
cset = ax.contour(X, Y, Z, cmap=cm.coolwarm)
ax.clabel(cset, fontsize=9, inline=1)
plt.show()
plt.show()

97.4 mplot3d example code: contour3d_demo2.py

```python
from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
from matplotlib import cm

fig = plt.figure()
ax = fig.gca(projection='3d')
X, Y, Z = axes3d.get_test_data(0.05)
cset = ax.contour(X, Y, Z, extend3d=True, cmap=cm.coolwarm)
ax.clabel(cset, fontsize=9, inline=1)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
from matplotlib import cm

fig = plt.figure()
ax = fig.gca(projection='3d')
X, Y, Z = axes3d.get_test_data(0.05)
ax.plot_surface(X, Y, Z, rstride=8, cstride=8, alpha=0.3)
cset = ax.contour(X, Y, Z, zdir='z', offset=-100, cmap=cm.coolwarm)
cset = ax.contour(X, Y, Z, zdir='x', offset=-40, cmap=cm.coolwarm)
cset = ax.contour(X, Y, Z, zdir='y', offset=40, cmap=cm.coolwarm)

ax.set_xlabel('X')
ax.set_xlim(-40, 40)
ax.set_ylabel('Y')
ax.set_ylim(-40, 40)
ax.set_zlabel('Z')
ax.set_zlim(-100, 100)

plt.show()
from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
from matplotlib import cm

fig = plt.figure()
ax = fig.gca(projection='3d')
X, Y, Z = axes3d.get_test_data(0.05)
cset = ax.contourf(X, Y, Z, cmap=cm.coolwarm)
ax.clabel(cset, fontsize=9, inline=1)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
.. versionadded:: 1.1.0
   This demo depends on new features added to contourf3d.

from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
from matplotlib import cm

fig = plt.figure()
ax = fig.gca(projection='3d')
X, Y, Z = axes3d.get_test_data(0.05)
ax.plot_surface(X, Y, Z, rstride=8, cstride=8, alpha=0.3)
cset = ax.contourf(X, Y, Z, zdir='z', offset=-100, cmap=cm.coolwarm)
cset = ax.contourf(X, Y, Z, zdir='x', offset=-40, cmap=cm.coolwarm)
cset = ax.contourf(X, Y, Z, zdir='y', offset=40, cmap=cm.coolwarm)

ax.set_xlabel('X')
ax.set_xlim(-40, 40)
ax.set_ylabel('Y')
ax.set_ylim(-40, 40)
ax.set_zlabel('Z')
```python
ax.set_zlim(-100, 100)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

### 97.8 mplot3d example code: custom_shaded_3d_surface.py

```python
""
Demonstrates using custom hillshading in a 3D surface plot.
""

```python
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cbook
from matplotlib import cm
from matplotlib.colors import LightSource
import matplotlib.pyplot as plt
import numpy as np

filename = cbook.get_sample_data('jacksboro_fault_dem.npz', asfileobj=False)
with np.load(filename) as dem:
    z = dem['elevation']
    nrows, ncols = z.shape
```
x = np.linspace(dem['xmin'], dem['xmax'], ncols)
y = np.linspace(dem['ymin'], dem['ymax'], nrows)
x, y = np.meshgrid(x, y)

region = np.s_[5:50, 5:50]
x, y, z = x[region], y[region], z[region]

fig, ax = plt.subplots(subplot_kw=dict(projection='3d'))

ls = LightSource(270, 45)
# To use a custom hillshading mode, override the built-in shading and pass
# in the rgb colors of the shaded surface calculated from "shade".
rgb = ls.shade(z, cmap=cm.gist_earth, vert_exag=0.1, blend_mode='soft')
surf = ax.plot_surface(x, y, z, rstride=1, cstride=1, facecolors=rgb,
linewidth=0, antialiased=False, shade=False)

plt.show()
Create 3D histogram of 2D data

Demo of a histogram for 2 dimensional data as a bar graph in 3D.

```python
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
x, y = np.random.rand(2, 100) * 4
hist, xedges, yedges = np.histogram2d(x, y, bins=4, range=[[0, 4], [0, 4]])

# Construct arrays for the anchor positions of the 16 bars.
# Note: np.meshgrid gives arrays in (ny, nx) so we use 'F' to flatten xpos,
# ypos in column-major order. For numpy >= 1.7, we could instead call meshgrid
# with indexing='ij'.
xpos, ypos = np.meshgrid(xedges[:-1] + 0.25, yedges[:-1] + 0.25)
xpos = xpos.flatten('F')
ypos = ypos.flatten('F')
zpos = np.zeros_like(xpos)

# Construct arrays with the dimensions for the 16 bars.
dx = 0.5 * np.ones_like(zpos)
dy = dx.copy()
dz = hist.flatten()

ax.bar3d(xpos, ypos, zpos, dx, dy, dz, color='b', zsort='average')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
import matplotlib as mpl
from mpl_toolkits.mplot3d import Axes3D
import numpy as np
import matplotlib.pyplot as plt

mpl.rcParams['legend.fontsize'] = 10

fig = plt.figure()
ax = fig.gca(projection='3d')
theta = np.linspace(-4 * np.pi, 4 * np.pi, 100)
z = np.linspace(-2, 2, 100)
r = z**2 + 1
x = r * np.sin(theta)
y = r * np.cos(theta)
ax.plot(x, y, z, label='parametric curve')
ax.legend()

plt.show()
# Plot of the Lorenz Attractor based on Edward Lorenz's 1963 "Deterministic
# Nonperiodic Flow" publication.
# http://journals.ametsoc.org/doi/abs/10.1175/1520-0469%281963%29020%3C0130%3ADNF%3E2.0.
# Note: Because this is a simple non-linear ODE, it would be more easily
# done using SciPy's ode solver, but this approach depends only
# upon NumPy.

import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

def lorenz(x, y, z, s=10, r=28, b=2.667):
    x_dot = s*(y - x)
    y_dot = r*x - y - x*z
    z_dot = x*y - b*z
    return x_dot, y_dot, z_dot

dt = 0.01
stepCnt = 10000

# Need one more for the initial values
xs = np.empty((stepCnt + 1,))
ys = np.empty((stepCnt + 1,))
zs = np.empty((stepCnt + 1,))

# Setting initial values
xs[0], ys[0], zs[0] = (0., 1., 1.05)

# Stepping through "time".
for i in range(stepCnt):
    # Derivatives of the X, Y, Z state
    x_dot, y_dot, z_dot = lorenz(xs[i], ys[i], zs[i])
    xs[i + 1] = xs[i] + (x_dot * dt)
    ys[i + 1] = ys[i] + (y_dot * dt)
    zs[i + 1] = zs[i] + (z_dot * dt)

fig = plt.figure()
ax = fig.gca(projection='3d')

ax.plot(xs, ys, zs, lw=0.5)
ax.set_xlabel("X Axis")
ax.set_ylabel("Y Axis")
ax.set_zlabel("Z Axis")
ax.set_title("Lorenz Attractor")

plt.show()
A tale of 2 subplots
Demonstrate the mixing of 2d and 3d subplots

```python
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np

def f(t):
    s1 = np.cos(2*np.pi*t)
    e1 = np.exp(-t)
    return np.multiply(s1, e1)

################
# First subplot
################
t1 = np.arange(0.0, 5.0, 0.1)
t2 = np.arange(0.0, 5.0, 0.02)
t3 = np.arange(0.0, 2.0, 0.01)

# Twice as tall as it is wide.
fig = plt.figure(figsize=plt.figaspect(2.))
fig.suptitle('A tale of 2 subplots')
ax = fig.add_subplot(2, 1, 1)
l = ax.plot(t1, f(t1), 'bo',
            t2, f(t2), 'k--', markerfacecolor='green')
ax.grid(True)
ax.set_ylabel('Damped oscillation')

################
# Second subplot
################
ax = fig.add_subplot(2, 1, 2, projection='3d')
X = np.arange(-5, 5, 0.25)
xlen = len(X)
Y = np.arange(-5, 5, 0.25)
ylen = len(Y)
X, Y = np.meshgrid(X, Y)
R = np.sqrt(X**2 + Y**2)
Z = np.sin(R)

surf = ax.plot_surface(X, Y, Z, rstride=1, cstride=1,
                        linewidth=0, antialiased=False)

ax.set_zlim3d(-1, 1)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np

# This example demonstrates mplot3d's offset text display.
# As one rotates the 3D figure, the offsets should remain oriented
# same way as the axis label, and should also be located "away"
# from the center of the plot.
#
# This demo triggers the display of the offset text for the x and
# y axis by adding 1e5 to X and Y. Anything less would not
# automatically trigger it.

fig = plt.figure()
ax = fig.gca(projection='3d')
X, Y = np.mgrid[0:6*np.pi:0.25, 0:4*np.pi:0.25]
Z = np.sqrt(np.abs(np.cos(X) + np.cos(Y)))
surf = ax.plot_surface(X + 1e5, Y + 1e5, Z, cmap='autumn', cstride=2, rstride=2)
ax.set_xlabel("X-Label")
ax.set_ylabel("Y-Label")
ax.set_zlabel("Z-Label")
ax.set_zlim(0, 2)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

97.14 mplot3d example code: pathpatch3d_demo.py

```python
import matplotlib.pyplot as plt
from matplotlib.patches import Circle, PathPatch
import mpl_toolkits.mplot3d as art3d
from matplotlib.text import TextPath

# register Axes3D class with matplotlib by importing Axes3D
import mpl_toolkits.mplot3d as Axes3D
import mpl_toolkits.mplot3d.art3d as art3d
from matplotlib.text import TextPath
```

```python
Draw flat objects in 3D plot

Demonstrate using pathpatch_2d_to_3d to 'draw' shapes and text on a 3D plot.

```
from matplotlib.transforms import Affine2D

def text3d(ax, xyz, s, zdir="z", size=None, angle=0, usetex=False, **kwargs):
    x, y, z = xyz
    if zdir == "y":
        xy1, z1 = (x, z), y
    elif zdir == "y":
        xy1, z1 = (y, z), x
    else:
        xy1, z1 = (x, y), z
    text_path = TextPath((0, 0), s, size=size, usetex=usetex)
    trans = Affine2D().rotate(angle).translate(xy1[0], xy1[1])
    p1 = PathPatch(trans.transform_path(text_path), **kwargs)
    ax.add_patch(p1)
    art3d.pathpatch_2d_to_3d(p1, z=z1, zdir=zdir)

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
p = Circle((5, 5), 3)
ax.add_patch(p)
art3d.pathpatch_2d_to_3d(p, z=0, zdir="x")

text3d(ax, (4, -2, 0), "X-axis", zdir="z", size=0.5, usetex=False, ec="none", fc="k")
text3d(ax, (12, 4, 0), "Y-axis", zdir="z", size=0.5, usetex=False, angle=53.14159, ec="none", fc="k")
text3d(ax, (12, 10, 4), "Z-axis", zdir="y", size=0.5, usetex=False, angle=52.14159, ec="none", fc="k")
text3d(ax, (1, 5, 0), r"\$G_{\mu\nu} + \Lambda g_{\mu\nu} = \frac{8\pi G}{c^4} T_{\mu\nu}$", zdir="z", size=1, usetex=True, ec="none", fc="k")

ax.set_xlim3d(0, 10)
ax.set_ylim3d(0, 10)
ax.set_zlim3d(0, 10)
plt.show()
"""
============================================= 
Generate polygons to fill under 3D line graph
============================================= 

Demonstrate how to create polygons which fill the space under a line graph. In this example polygons are semi-transparent, creating a sort of 'jagged stained glass' effect.
"""

from mpl_toolkits.mplot3d import Axes3D
from matplotlib.collections import PolyCollection
import matplotlib.pyplot as plt
import matplotlib.colors as mcolors
import numpy as np

fig = plt.figure()
ax = fig.gca(projection='3d')

def cc(arg):
    """
```python
return mcolors.to_rgba(arg, alpha=0.6)

xs = np.arange(0, 10, 0.4)
verts = []
zs = [0.0, 1.0, 2.0, 3.0]
for z in zs:
    ys = np.random.rand(len(xs))
    ys[0], ys[-1] = 0, 0
    verts.append(list(zip(xs, ys)))

poly = PolyCollection(verts, facecolors=[cc('r'), cc('g'), cc('b'), cc('y')])
poly.set_alpha(0.7)
ax.add_collection3d(poly, zs=zs, zdir='y')

ax.set_xlabel('X')
ax.set_xlim3d(0, 10)
ax.set_ylabel('Y')
ax.set_ylim3d(-1, 4)
ax.set_zlabel('Z')
ax.set_zlim3d(0, 1)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
```python

from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure()
ax = fig.gca(projection='3d')

# Make the grid
x, y, z = np.meshgrid(np.arange(-0.8, 1, 0.2),
                      np.arange(-0.8, 1, 0.2),
                      np.arange(-0.8, 1, 0.8))

# Make the direction data for the arrows
u = np.sin(np.pi * x) * np.cos(np.pi * y) * np.cos(np.pi * z)
```

```
```
v = -np.cos(np.pi * x) * np.sin(np.pi * y) * np.cos(np.pi * z)
w = (np.sqrt(2.0 / 3.0) * np.cos(np.pi * x) * np.cos(np.pi * y) * np.sin(np.pi * z))

ax.quiver(x, y, z, u, v, w, length=0.1, normalize=True)

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

97.17 mplot3d example code: rotate_axes3d_demo.py

```python
# Rotate axes3d demo

import numpy as np
import matplotlib.pyplot as plt

fig = plt.figure()
ax = fig.add_subplot(projection='3d')

# Create some data
x, y, z = np.random.random((3, 100))

# Plot the data using quiver
u, v, w = np.cos(2*np.pi*x)*np.sin(2*np.pi*y), np.cos(2*np.pi*x)*np.cos(2*np.pi*y), np.sin(2*np.pi*y)
ax.quiver(x, y, z, u, v, w, length=0.1, normalize=True)

plt.show()
```

A very simple animation of a rotating 3D plot.

See wire3d_animation_demo for another simple example of animating a 3D plot.
```python
from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# load some test data for demonstration and plot a wireframe
X, Y, Z = axes3d.get_test_data(0.1)
ax.plot_wireframe(X, Y, Z, rstride=5, cstride=5)

# rotate the axes and update
for angle in range(0, 360):
    ax.view_init(30, angle)
    plt.draw()
    plt.pause(.001)
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

97.18 mplot3d example code: scatter3d_demo.py
3D scatterplot

Demonstration of a basic scatterplot in 3D.

```python
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np

def randrange(n, vmin, vmax):
    """
    Helper function to make an array of random numbers having shape (n,)
    with each number distributed Uniform(vmin, vmax).
    """
    return (vmax - vmin)*np.random.rand(n) + vmin

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

n = 100

# For each set of style and range settings, plot n random points in the box
# defined by x in [23, 32], y in [0, 100], z in [zlow, zhigh].
for c, m, zlow, zhigh in [('r', 'o', -50, -25), ('b', '^', -30, -5)]:
    xs = randrange(n, 23, 32)
    ys = randrange(n, 0, 100)
    zs = randrange(n, zlow, zhigh)
    ax.scatter(xs, ys, zs, c=c, marker=m)

ax.set_xlabel('X Label')
ax.set_ylabel('Y Label')
ax.set_zlabel('Z Label')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
3D plots as subplots

Demonstrate including 3D plots as subplots.

```python
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d.axes3d import Axes3D, get_test_data
from matplotlib import cm
import numpy as np

# set up a figure twice as wide as it is tall
fig = plt.figure(figsize=plt.figaspect(0.5))

# First subplot
# set up the axes for the first plot
ax = fig.add_subplot(1, 2, 1, projection='3d')

# plot a 3D surface like in the example mplot3d/surface3d_demo
X = np.arange(-5, 5, 0.25)
Y = np.arange(-5, 5, 0.25)
X, Y = np.meshgrid(X, Y)
R = np.sqrt(X**2 + Y**2)
Z = np.sin(R)
surf = ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap=cm.coolwarm)
```
```python
ax.set_zlim(-1.01, 1.01)
fig.colorbar(surf, shrink=0.5, aspect=10)

#*************
# Second subplot
#*************
# set up the axes for the second plot
ax = fig.add_subplot(1, 2, 2, projection='3d')

# plot a 3D wireframe like in the example mplot3d/wire3d_demo
X, Y, Z = get_test_data(0.05)
ax.plot_wireframe(X, Y, Z, rstride=10, cstride=10)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

97.20 mplot3d example code: surface3d_demo.py
3D surface (color map)
======================

Demonstrates plotting a 3D surface colored with the coolwarm color map. The surface is made opaque by using antialiased=False.

Also demonstrates using the LinearLocator and custom formatting for the z axis tick labels.

```python
from mpl_toolkits.mplot3d import Axes3D
import matplotlib as plt
import matplotlib.pyplot as plt
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter
import numpy as np

fig = plt.figure()
ax = fig.gca(projection='3d')

# Make data.
X = np.arange(-5, 5, 0.25)
Y = np.arange(-5, 5, 0.25)
X, Y = np.meshgrid(X, Y)
R = np.sqrt(X**2 + Y**2)
Z = np.sin(R)

# Plot the surface.
surf = ax.plot_surface(X, Y, Z, cmap=cm.coolwarm,
                       linewidth=0, antialiased=False)

# Customize the z axis.
ax.set_zlim(-1.01, 1.01)
ax.zaxis.set_major_locator(LinearLocator(10))
ax.zaxis.set_major_formatter(FormatStrFormatter('%.02f'))

# Add a color bar which maps values to colors.
fig.colorbar(surf, shrink=0.5, aspect=5)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
```python
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# Make data
u = np.linspace(0, 2 * np.pi, 100)
v = np.linspace(0, np.pi, 100)
x = 10 * np.outer(np.cos(u), np.sin(v))
y = 10 * np.outer(np.sin(u), np.sin(v))
z = 10 * np.outer(np.ones(np.size(u)), np.cos(v))

```
# Plot the surface
ax.plot_surface(x, y, z, color='b')
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

97.22 mplot3d example code: surface3d_demo3.py

```python
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
from matplotlib import cm
from matplotlib.ticker import LinearLocator

```

Demonstrates plotting a 3D surface colored in a checkerboard pattern.

```python
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
from matplotlib import cm
from matplotlib.ticker import LinearLocator
```
```python
import numpy as np

fig = plt.figure()
ax = fig.gca(projection='3d')

# Make data.
X = np.arange(-5, 5, 0.25)
xlen = len(X)
Y = np.arange(-5, 5, 0.25)
ylen = len(Y)
X, Y = np.meshgrid(X, Y)
R = np.sqrt(X**2 + Y**2)
Z = np.sin(R)

# Create an empty array of strings with the same shape as the meshgrid, and
# populate it with two colors in a checkerboard pattern.
colortuple = ('y', 'b')
colors = np.empty(X.shape, dtype=str)
for y in range(ylen):
    for x in range(xlen):
        colors[x, y] = colortuple[(x + y) % len(colortuple)]

# Plot the surface with face colors taken from the array we made.
surf = ax.plot_surface(X, Y, Z, facecolors=colors, linewidth=0)

# Customize the z axis.
ax.set_zlim(-1, 1)
ax.w_zaxis.set_major_locator(LinearLocator(6))

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
```python
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import pyplot as plt
import numpy as np

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# Create the mesh in polar coordinates and compute corresponding Z.
r = np.linspace(0, 1.25, 50)
```

---

**3D surface with polar coordinates**

Demonstrates plotting a surface defined in polar coordinates. Uses the reversed version of the YlGnBu color map. Also demonstrates writing axis labels with latex math mode.

*Example contributed by Armin Moser.*

```python
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import pyplot as plt
import numpy as np

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# Create the mesh in polar coordinates and compute corresponding Z.
r = np.linspace(0, 1.25, 50)
```
```python
p = np.linspace(0, 2*np.pi, 50)
R, P = np.meshgrid(r, p)
Z = ((R**2 - 1)**2)

# Express the mesh in the cartesian system.
X, Y = R*np.cos(P), R*np.sin(P)

# Plot the surface.
ax.plot_surface(X, Y, Z, cmap=plt.cm.YlGnBu_r)

# Tweak the limits and add latex math labels.
ax.set_zlim(0, 1)
ax.set_xlabel(r'$\phi_{\text{real}}$')
ax.set_ylabel(r'$\phi_{\text{im}}$')
ax.set_zlabel(r'$V(\phi)$')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

97.24 mplot3d example code: text3d_demo.py
# Text annotations in 3D

Demonstrates the placement of text annotations on a 3D plot.

Functionality shown:
- Using the text function with three types of 'zdir' values: None, an axis name (ex. 'x'), or a direction tuple (ex. (1, 1, 0)).
- Using the text function with the color keyword.
- Using the text2D function to place text on a fixed position on the ax object.

```python
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt

fig = plt.figure()
ax = fig.gca(projection='3d')

# Demo 1: zdir
zdirs = (None, 'x', 'y', 'z', (1, 1, 0), (1, 1, 1))
xs = (1, 4, 4, 9, 4, 1)
ys = (2, 5, 8, 10, 1, 2)
zs = (10, 3, 8, 9, 1, 8)

for zdir, x, y, z in zip(zdirs, xs, ys, zs):
    label = '(%d, %d, %d), dir=%s' % (x, y, z, zdir)
    ax.text(x, y, z, label, zdir)

# Demo 2: color
ax.text(9, 0, 0, "red", color='red')

# Demo 3: text2D
# Placement 0, 0 would be the bottom left, 1, 1 would be the top right.
ax.text2D(0.05, 0.95, "2D Text", transform=ax.transAxes)

# Tweaking display region and labels
ax.set_xlim(0, 10)
ax.set_ylim(0, 10)
ax.set_zlim(0, 10)
ax.set_xlabel('X axis')
ax.set_ylabel('Y axis')
ax.set_zlabel('Z axis')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Triangular 3D contour plot

Contour plots of unstructured triangular grids.

The data used is the same as in the second plot of trisurf3d_demo2. tricontourf3d_demo shows the filled version of this example.

```python
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.tri as tri
import numpy as np

n_angles = 48
n_radii = 8
min_radius = 0.25

# Create the mesh in polar coordinates and compute x, y, z.
radii = np.linspace(min_radius, 0.95, n_radii)

# Compute x, y, z.
theta = np.linspace(0.0, 2.0*np.pi, n_angles + 1)[:-1]

angles = np.broadcast_to(theta, (n_radii, n_angles)).flatten()
r = np.broadcast_to(radii, (n_radii, n_angles)).flatten()

x = r * np.cos(angles)
y = r * np.sin(angles)

plt.tricontourf(x, y, radii, 50, cmap='binary')
plt.tricontour(x, y, radii, 50, colors='k', linewidth=0.2)
```

angles = np.linspace(0, 2*np.pi, n_angles, endpoint=False)
angles = np.repeat(angles[...], np.newaxis, n_radii, axis=1)
angles[:, 1::2] += np.pi/n_angles

x = (radii*np.cos(angles)).flatten()
y = (radii*np.sin(angles)).flatten()
z = (np.cos(radii)*np.cos(angles*3.0)).flatten()

# Create a custom triangulation.
triang = tri.Triangulation(x, y)

# Mask off unwanted triangles.
xmid = x[triang.triangles].mean(axis=1)
ymid = y[triang.triangles].mean(axis=1)
mask = np.where(xmid*xmid + ymid*ymid < min_radius*min_radius, 1, 0)
triang.set_mask(mask)

fig = plt.figure()
ax = fig.gca(projection='3d')
ax.tricontour(triang, z, cmap=plt.cm.CMRmap)

# Customize the view angle so it's easier to understand the plot.
ax.view_init(elev=45.)

plt.show()
Triangular 3D filled contour plot

Filled contour plots of unstructured triangular grids.

The data used is the same as in the second plot of trisurf3d_demo2. tricontour3d_demo shows the unfilled version of this example.

```python
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.tri as tri
import numpy as np

# First create the x, y, z coordinates of the points.
    n_angles = 48
    n_radii = 8
    min_radius = 0.25

# Create the mesh in polar coordinates and compute x, y, z.
```
radii = np.linspace(min_radius, 0.95, n_radii)
angles = np.linspace(0, 2*np.pi, n_angles, endpoint=False)
angles = np.repeat(angles[..., np.newaxis], n_radii, axis=1)
angles[:, 1::2] += np.pi/n_angles

x = (radii*np.cos(angles)).flatten()
y = (radii*np.sin(angles)).flatten()
z = (np.cos(radii)*np.cos(angles*3.0)).flatten()

# Create a custom triangulation.
triang = tri.Triangulation(x, y)

# Mask off unwanted triangles.
xmid = x[triang.triangles].mean(axis=1)
ymid = y[triang.triangles].mean(axis=1)
mask = np.where(xmid*xmid + ymid*ymid < min_radius*min_radius, 1, 0)
triang.set_mask(mask)

fig = plt.figure()
ax = fig.gca(projection='3d')
ax.tricontourf(triang, z, cmap=plt.cm.cMRmap)

# Customize the view angle so it's easier to understand the plot.
ax.view_init(elev=45.)

plt.show()
Triangular 3D surfaces

Plot a 3D surface with a triangular mesh.

```python
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np

n_radii = 8
n_angles = 36

# Make radii and angles spaces (radius r=0 omitted to eliminate duplication).
radii = np.linspace(0.125, 1.0, n_radii)
angles = np.linspace(0, 2*np.pi, n_angles, endpoint=False)

# Repeat all angles for each radius.
angles = np.repeat(angles[..., np.newaxis], n_radii, axis=1)

# Now we want to repeat the points radii[n_i, ...] r times for each
# angle[j][n_i].  To do this, we first broadcast radii to match the
# dimensions of angles, then multiply.

t = np.empty([n_r, n_angles, 2], dtype=np.float64)
t[:, :, 0] = radii[:, np.newaxis] * np.cos(angles)
t[:, :, 1] = radii[:, np.newaxis] * np.sin(angles)

angles = np.repeat(angles[..., np.newaxis], n_radii, axis=2)
angles[:, :, 1] += np.pi / n_radii

t = np.empty(t.shape, dtype=t.dtype)
t[:, :, 0] = t[:, :, 0] * np.cos(angles)
t[:, :, 1] = t[:, :, 1] * np.sin(angles)
```

```bash
plt.figure()
ax = plt.subplot(111, projection='3d')
ax.plot_trisurf(t[:, :, 0], t[:, :, 1], Z,
                linewidth=0.1, antialiased=True)
plt.show()
```
# Convert polar (radii, angles) coords to cartesian (x, y) coords.  
# (0, 0) is manually added at this stage, so there will be no duplicate  
# points in the (x, y) plane.
x = np.append(0, (radii*np.cos(angles)).flatten())
y = np.append(0, (radii*np.sin(angles)).flatten())

# Compute z to make the pringle surface.
z = np.sin(-x*y)

fig = plt.figure()
ax = fig.gca(projection='3d')

ax.plot_trisurf(x, y, z, linewidth=0.2, antialiased=True)

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

97.28 mplot3d example code: trisurf3d_demo2.py

```
# More triangular 3D surfaces

Two additional examples of plotting surfaces with triangular mesh.
The first demonstrates use of plot_trisurf's triangles argument, and the 
second sets a Triangulation object's mask and passes the object directly
```
```python
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.tri as mtri

fig = plt.figure(figsize=plt.figaspect(0.5))

# First plot

# Make a mesh in the space of parameterisation variables u and v
u = np.linspace(0, 2.0 * np.pi, endpoint=True, num=50)
v = np.linspace(-0.5, 0.5, endpoint=True, num=10)
u, v = np.meshgrid(u, v)
u, v = u.flatten(), v.flatten()

# This is the Mobius mapping, taking a u, v pair and returning an x, y, z triple
x = (1 + 0.5 * v * np.cos(u / 2.0)) * np.cos(u)
y = (1 + 0.5 * v * np.cos(u / 2.0)) * np.sin(u)
z = 0.5 * v * np.sin(u / 2.0)

# Triangulate parameter space to determine the triangles
tri = mtri.Triangulation(u, v)

# Plot the surface. The triangles in parameter space determine which x, y, z points are connected by an edge.
ax = fig.add_subplot(1, 2, 1, projection='3d')
ax.plot_trisurf(x, y, z, triangles=tri.triangles, cmap=plt.cm.Spectral)
ax.set_zlim(-1, 1)

# Second plot

# Make parameter spaces radii and angles.
n_angles = 36
n_radii = 8
min_radius = 0.25
radii = np.linspace(min_radius, 0.95, n_radii)

angles = np.linspace(0, 2*np.pi, n_angles, endpoint=False)
angles = np.repeat(angles[..., np.newaxis], n_radii, axis=1)
angles[:, 1::2] += np.pi/n_angles

# Map radius, angle pairs to x, y, z points.
x = (radii*np.cos(angles)).flatten()
```

97.28. mplot3d example code: trisurf3d_demo2.py
```python
y = (radii*np.sin(angles)).flatten()
z = (np.cos(radii)*np.cos(angles**3.0)).flatten()

# Create the Triangulation; no triangles so Delaunay triangulation created.
triang = mtri.Triangulation(x, y)

# Mask off unwanted triangles.
xmid = x[triang.triangles].mean(axis=1)
ymid = y[triang.triangles].mean(axis=1)
mask = np.where((xmid**2 + ymid**2 < min_radius**2), 1, 0)
triang.set_mask(mask)

# Plot the surface.
ax = fig.add_subplot(1, 2, 2, projection='3d')
ax.plot_trisurf(triang, z, cmap=plt.cm.CMRmap)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

97.29 mplot3d example code: wire3d_animation_demo.py
A very simple 'animation' of a 3D plot. See also rotate_axes3d_demo.

from __future__ import print_function
from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
import numpy as np
import time

def generate(X, Y, phi):
    
    Generates Z data for the points in the X, Y meshgrid and parameter phi.
    
    R = 1 - np.sqrt(X**2 + Y**2)
    return np.cos(2 * np.pi * X + phi) * R

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# Make the X, Y meshgrid.
xs = np.linspace(-1, 1, 50)
ys = np.linspace(-1, 1, 50)
X, Y = np.meshgrid(xs, ys)

# Set the z axis limits so they aren't recalculated each frame.
ax.set_zlim(-1, 1)

# Begin plotting.
wframe = None
tstart = time.time()
for phi in np.linspace(0, 180. / np.pi, 100):
    # If a line collection is already remove it before drawing.
    if wframe:
        ax.collections.remove(wframe)

    Z = generate(X, Y, phi)
    wframe = ax.plot_wireframe(X, Y, Z, rstride=2, cstride=2)
    plt.pause(.001)

print('Average FPS: %f' % (100 / (time.time() - tstart)))

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
```python
from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt

fig = plt.figure()
asx = fig.add_subplot(111, projection='3d')

# Grab some test data.
X, Y, Z = axes3d.get_test_data(0.05)

# Plot a basic wireframe.
asx.plot_wireframe(X, Y, Z, rstride=10, cstride=10)

plt.show()
```
Keywords: python, matplotlib, pylab, example, codex (see Search examples)
97.31 mplot3d example code: wire3d_zero_stride.py

Column (x) stride set to 0

Row (y) stride set to 0
3D wireframe plots in one direction

Demonstrates that setting rstride or cstride to 0 causes wires to not be generated in the corresponding direction.

```python
from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt

fig, [ax1, ax2] = plt.subplots(2, 1, figsize=(8, 12), subplot_kw={'projection': '3d'})

# Get the test data
X, Y, Z = axes3d.get_test_data(0.05)

# Give the first plot only wireframes of the type y = c
ax1.plot_wireframe(X, Y, Z, rstride=10, cstride=0)
ax1.set_title("Column (x) stride set to 0")

# Give the second plot only wireframes of the type x = c
ax2.plot_wireframe(X, Y, Z, rstride=0, cstride=10)
ax2.set_title("Row (y) stride set to 0")

plt.tight_layout()
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
98.1 pie_and_polar_charts example code: pie_demo_features.py

Demonstration of a basic pie chart plus a few additional features.

In addition to the basic pie chart, this demo shows a few optional features:
* slice labels
* auto-labeling the percentage
* offsetting a slice with "explode"
* drop-shadow
* custom start angle

Note about the custom start angle:

The default `startangle` is 0, which would start the "Frogs" slice on the positive x-axis. This example sets `startangle = 90` such that everything is rotated counter-clockwise by 90 degrees, and the frog slice starts on the positive y-axis.

```python
import matplotlib.pyplot as plt

# Pie chart, where the slices will be ordered and plotted counter-clockwise:
labels = ['Frogs', 'Hogs', 'Dogs', 'Logs']
sizes = [15, 30, 45, 10]
explode = (0, 0.1, 0, 0)  # only "explode" the 2nd slice (i.e. 'Hogs')

fig1, ax1 = plt.subplots()
ax1.pie(sizes, explode=explode, labels=labels, autopct='%.1f%%',
        shadow=True, startangle=90)
ax1.axis('equal')  # Equal aspect ratio ensures that pie is drawn as a circle.

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Demo of bar plot on a polar axis.

```python
import numpy as np
import matplotlib.pyplot as plt

# Compute pie slices
N = 20
theta = np.linspace(0.0, 2 * np.pi, N, endpoint=False)
radii = 10 * np.random.rand(N)
width = np.pi / 4 * np.random.rand(N)

ax = plt.subplot(111, projection='polar')
bars = ax.bar(theta, radii, width=width, bottom=0.0)

# Use custom colors and opacity
for r, bar in zip(radii, bars):
    bar.set_facecolor(rand_color())
    bar.set_alpha(0.5)
```

```python
bar.set_facecolor(plt.cm.viridis(r / 10.))
bar.set_alpha(0.5)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

### 98.3 pie_and_polar_charts example code: polar_scatter_demo.py

```python
# Import necessary libraries
import numpy as np
import matplotlib.pyplot as plt

# Demo of scatter plot on a polar axis.
# Size increases radially in this example and color increases with angle (just to verify the symbols are being scattered correctly).

# Example code
import numpy as np
import matplotlib.pyplot as plt

# Scatter plot on polar axis
```

---

2326 Chapter 98. pie_and_polar_charts Examples
# Compute areas and colors
N = 150
r = 2 * np.random.rand(N)
theta = 2 * np.pi * np.random.rand(N)
area = 200 * r**2
colors = theta

ax = plt.subplot(111, projection='polar')
c = ax.scatter(theta, r, c=colors, s=area, cmap='hsv', alpha=0.75)

plt.show()
matplotlib supports accented characters via TeX mathtext

The following accents are provided: \hat, \breve, \grave, \bar, \acute, \tilde, \vec, \dot, \ddot. All of them have the same syntax, e.g., to make an overbar you do \bar{o} or to make an o umlaut you do \ddot{o}. The shortcuts are also provided, e.g.,: \"o \e \e \-n \.x \^y
```python
import matplotlib.pyplot as plt

plt.axes([0.1, 0.15, 0.8, 0.75])
plt.plot(range(10))

plt.title(r'$\ddot{o}\acute{e}\grave{e}\hat{O}\breve{i}\bar{A}\tilde{n}\vec{q}$', fontsize=20)
# shorthand is also supported and curly's are optional
plt.xlabel(r'''"o\ddot o \'e\`e\~n\^x\y''', fontsize=20)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

99.2 pylab_examples example code: agg_buffer.py

Use backend agg to access the figure canvas as an RGB string and then convert it to an array and pass it to Pillow for rendering.
```python
import numpy as np

import matplotlib.pyplot as plt
from matplotlib.backends.backend_agg import FigureCanvasAgg

try:
    from PIL import Image
except ImportError:
    raise SystemExit("Pillow must be installed to run this example")

plt.plot([1, 2, 3])

canvas = plt.get_current_fig_manager().canvas

agg = canvas.switch_backends(FigureCanvasAgg)
agg.draw()
s = agg.tostring_rgb()

# get the width and the height to resize the matrix
l, b, w, h = agg.figure.bbox.bounds
w, h = int(w), int(h)

X = np.fromstring(s, np.uint8)
X.shape = h, w, 3

try:
    im = Image.fromstring("RGB", (w, h), s)
except Exception:
    im = Image.frombytes("RGB", (w, h), s)

# Uncomment this line to display the image using ImageMagick's
# `display` tool.
# im.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
99.3 pylab_examples example code: agg_buffer_to_array.py

a simple figure
import matplotlib.pyplot as plt
import numpy as np

# make an agg figure
fig, ax = plt.subplots()
ax.plot([1, 2, 3])
ax.set_title('a simple figure')
fig.canvas.draw()

# grab the pixel buffer and dump it into a numpy array
X = np.array(fig.canvas.renderer._renderer)

# now display the array X as an Axes in a new figure
fig2 = plt.figure()
ax2 = fig2.add_subplot([111, frameon=False])
ax2.imshow(X)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
import matplotlib.pyplot as plt
from matplotlib.patches import Rectangle

# build a rectangle in axes coords
left, width = .25, .5
bottom, height = .25, .5
right = left + width
top = bottom + height
ax = plt.gca()
p = plt.Rectangle((left, bottom), width, height,
    fill=False,
)
p.set_transform(ax.transAxes)
p.set_clip_on(False)
ax.add_patch(p)
ax.text(left, bottom, 'left top',
    horizontalalignment='left',
    verticalalignment='top',
    transform=ax.transAxes)

ax.text(left, bottom, 'left bottom',
    horizontalalignment='left',
    verticalalignment='bottom',
    transform=ax.transAxes)

ax.text(right, top, 'right bottom',
    horizontalalignment='right',
    verticalalignment='bottom',
    transform=ax.transAxes)

ax.text(right, top, 'right top',
    horizontalalignment='right',
    verticalalignment='top',
    transform=ax.transAxes)

ax.text(right, bottom, 'center top',
    horizontalalignment='center',
    verticalalignment='top',
    transform=ax.transAxes)

ax.text(left, 0.5*(bottom + top), 'right center',
    horizontalalignment='right',
    verticalalignment='center',
    rotation='vertical',
    transform=ax.transAxes)

ax.text(left, 0.5*(bottom + top), 'left center',
    horizontalalignment='left',
    verticalalignment='center',
    rotation='vertical',
    transform=ax.transAxes)

ax.text(0.5*(left + right), 0.5*(bottom + top), 'middle',
    horizontalalignment='center',
    verticalalignment='center',
    transform=ax.transAxes)

ax.text(right, 0.5*(bottom + top), 'centered',
    horizontalalignment='center',
    verticalalignment='center',
    rotation='vertical',
    transform=ax.transAxes)

ax.text(left, top, 'rotated
with newlines',
    horizontalalignment='center',
    verticalalignment='center',
    rotation='vertical',
    transform=ax.transAxes)
rotation=45,
transform=ax.transAxes)

plt.axis('off')

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

99.5 pylab_examples example code: anchored_artists.py

```python
from matplotlib.patches import Rectangle, Ellipse
from matplotlib.offsetbox import AnchoredOffsetbox, AuxTransformBox, VPacker,
    TextArea, DrawingArea

class AnchoredText(AnchoredOffsetbox):
    def __init__(self, s, loc, pad=0.4, borderpad=0.5, prop=None, frameon=True):
        self.txt = TextArea(s, minimumdescent=False)
```

Figure 1a

0.0 0.2 0.4 0.6 0.8 1.0
0.0 0.2 0.4 0.6 0.8 1.0

Figure 1a

1′
super(AnchoredText, self).__init__(loc, pad=pad, borderpad=borderpad,
    child=self.txt,
    prop=prop,
    frameon=frameon)

class AnchoredSizeBar(AnchoredOffsetbox):
    def __init__(self, transform, size, label, loc,
        pad=0.1, borderpad=0.1, sep=2, prop=None, frameon=True):
        '''
        Draw a horizontal bar with the size in data coordinate of the given axes.
        A label will be drawn underneath (center-aligned).

        pad, borderpad in fraction of the legend font size (or prop)
        sep in points.
        '''
        self.size_bar = AuxTransformBox(transform)
        self.size_bar.add_artist(Rectangle((0, 0), size, 0, fc="none", lw=1.0))

        self.txt_label = TextArea(label, minimumdescent=False)

        self._box = VPacker(children=[self.size_bar, self.txt_label],
            align="center",
            pad=0, sep=sep)

        AnchoredOffsetbox.__init__(self, loc, pad=pad, borderpad=borderpad,
            child=self._box,
            prop=prop,
            frameon=frameon)

class AnchoredEllipse(AnchoredOffsetbox):
    def __init__(self, transform, width, height, angle, loc,
        pad=0.1, borderpad=0.1, prop=None, frameon=True):
        '''
        Draw an ellipse the size in data coordinate of the given axes.

        pad, borderpad in fraction of the legend font size (or prop)
        '''
        self._box = AuxTransformBox(transform)
        self.ellipse = Ellipse((0, 0), width, height, angle)
        self._box.add_artist(self.ellipse)

        AnchoredOffsetbox.__init__(self, loc, pad=pad, borderpad=borderpad,
            child=self._box,
            prop=prop,
            frameon=frameon)

class AnchoredDrawingArea(AnchoredOffsetbox):
    def __init__(self, width, height, xdescent, ydescent,
        loc, pad=0.4, borderpad=0.5, prop=None, frameon=True):
```python
self.da = DrawingArea(width, height, xdescent, ydescent)

super(AnchoredDrawingArea, self).__init__(loc, pad=pad, borderpad=borderpad,
child=self.da,
prop=None,
frameon=frameon)

if __name__ == "__main__":
    import matplotlib.pyplot as plt
    ax = plt.gca()
    ax.set_aspect(1.)

    at = AnchoredText("Figure 1a",
                     loc=2, frameon=True)
    at.patch.set_boxstyle("round,pad=0.,rounding_size=0.2")
    ax.add_artist(at)

    from matplotlib.patches import Circle
    ada = AnchoredDrawingArea(20, 20, 0, 0,
                               loc=1, pad=0., frameon=False)
    p = Circle((10, 10), 10)
    ada.da.add_artist(p)
    ax.add_artist(ada)

    # draw an ellipse of width=0.1, height=0.15 in the data coordinate
    ae = AnchoredEllipse(ax.transData, width=0.1, height=0.15, angle=0.,
                          loc=3, pad=0.5, borderpad=0.4, frameon=True)
    ax.add_artist(ae)

    # draw a horizontal bar with length of 0.1 in Data coordinate
    # (ax.transData) with a label underneath.
    asb = AnchoredSizeBar(ax.transData,
                          0.1,
                          r"1$\prime$",
                          loc=8,
                          pad=0.1, borderpad=0.5, sep=5,
                          frameon=False)
    ax.add_artist(asb)

    plt.draw()
    plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Pyplot animation example.

The method shown here is only for very simple, low-performance use. For more demanding applications, look at the animation module and the examples that use it.

```python
import matplotlib.pyplot as plt
import numpy as np

x = np.arange(6)
y = np.arange(5)
z = x * y[:, np.newaxis]

for i in range(5):
    if i == 0:
        p = plt.imshow(z)
        fig = plt.gcf()
        plt.clim()  # clamp the color limits
        plt.title("Boring slide show")
    else:
```

Boring slide show
```
z = z + 2
p.set_data(z)

print("step", i)
plt.pause(0.5)
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

### 99.7 pylab_examples example code: annotation_demo.py

**figure fraction**

![Diagram showing figure fraction, axes fraction, point offset from data, and pixel offset from axes fraction.](image)
a polar annotation
Some examples of how to annotate points in figures. You specify an annotation point $xy=(x,y)$ and a text point $xytext=(x,y)$ for the annotated points and text location, respectively. Optionally, you can specify the coordinate system of $xy$ and $xytext$ with one of the following strings for $xycoords$ and textcoords (default is 'data'):

- 'figure points': points from the lower left corner of the figure
- 'figure pixels': pixels from the lower left corner of the figure
- 'figure fraction': 0,0 is lower left of figure and 1,1 is upper, right
- 'axes points': points from lower left corner of axes
- 'axes pixels': pixels from lower left corner of axes
- 'axes fraction': 0,0 is lower left of axes and 1,1 is upper right
- 'offset points': Specify an offset (in points) from the $xy$ value
- 'offset pixels': Specify an offset (in pixels) from the $xy$ value
- 'data': use the axes data coordinate system

Optionally, you can specify arrow properties which draws an arrow from the text to the annotated point by giving a dictionary of arrow properties.

Valid keys are

- width : the width of the arrow in points
- frac : the fraction of the arrow length occupied by the head
headwidth : the width of the base of the arrow head in points
shrink : move the tip and base some percent away from the
        annotated point and text
any key for matplotlib.patches.polygon (e.g., facecolor)

For physical coordinate systems (points or pixels) the origin is the
(bottom, left) of the figure or axes.

""

import matplotlib.pyplot as plt
from matplotlib.patches import Ellipse
import numpy as np

# If only one location is given, the text and xypoint being
# annotated are assumed to be the same
fig = plt.figure()
ax = fig.add_subplot(111, autoscale_on=False, xlim=(-1, 5), ylim=(-3, 5))
t = np.arange(0.0, 5.0, 0.01)
s = np.cos(2*np.pi*t)
line, = ax.plot(t, s)

ax.annotate('figure pixels',
            xy=(10, 10), xycoords='figure pixels')

ax.annotate('figure points',
            xy=(80, 80), xycoords='figure points')

ax.annotate('point offset from data',
            xy=(2, 1), xycoords='data',
            xytext=(-15, 25), textcoords='offset points',
            arrowprops=dict(facecolor='black', shrink=0.05),
            horizontalalignment='right', verticalalignment='bottom')

ax.annotate('axes fraction',
            xy=(3, 1), xycoords='data',
            xytext=(0.8, 0.95), textcoords='axes fraction',
            arrowprops=dict(facecolor='black', shrink=0.05),
            horizontalalignment='right', verticalalignment='top')

ax.annotate('figure fraction',
            xy=(.025, .975), xycoords='figure fraction',
            horizontalalignment='left', verticalalignment='top',
            fontsize=20)

# use negative points or pixels to specify from right, top -10, 10
# is 10 points to the left of the right side of the axes and 10
# points above the bottom
ax.annotate('pixel offset from axes fraction',
            xy=(1, 0), xycoords='axes fraction',
            xytext=(-20, 20), textcoords='offset pixels',
            arrowprops=dict(facecolor='black', shrink=0.05),
            horizontalalignment='right', verticalalignment='top')
horizontalalignment='right',
verticalalignment='bottom')

# You can specify the xypoint and the xytext in different positions and
# coordinate systems, and optionally turn on a connecting line and mark
# the point with a marker. Annotations work on polar axes too.
# In the example below, the xy point is in native coordinates (xycoords
# defaults to 'data'). For a polar axes, this is in (theta, radius) space.
# The text in the example is placed in the fractional figure coordinate system.
# Text keyword args like horizontal and vertical alignment are respected.
fig = plt.figure()
ax = fig.add_subplot(111, projection='polar')
r = np.arange(0, 1, 0.001)
theta = 2*2*np.pi*r
line, = ax.plot(theta, r)
ind = 800
thisr, thistheta = r[ind], theta[ind]
ax.plot([thistheta], [thisr], 'o')
ax.annotate('a polar annotation',
            xy=(thistheta, thisr),  # theta, radius
            xytext=(0.05, 0.05),    # fraction, fraction
            textcoords='figure fraction',
            arrowprops=dict(facecolor='black', shrink=0.05),
            horizontalalignment='left',
            verticalalignment='bottom')

# You can also use polar notation on a cartesian axes. Here the native
# coordinate system ('data') is cartesian, so you need to specify the
# xycoords and textcoords as 'polar' if you want to use (theta, radius).
el = Ellipse((0, 0), 10, 20, facecolor='r', alpha=0.5)

fig = plt.figure()
ax = fig.add_subplot(111, aspect='equal')
ax.add_artist(el)
el.set_clip_box(ax.bbox)
ax.annotate('the top',
            xy=(np.pi/2., 10.),    # theta, radius
            xytext=(np.pi/3, 20.),  # theta, radius
            xycoords='polar',
            textcoords='polar',
            arrowprops=dict(facecolor='black', shrink=0.05),
            horizontalalignment='left',
            verticalalignment='bottom',
            clip_on=True)  # clip to the axes bounding box

ax.set_xlim([-20, 20])
ax.set_ylim([-20, 20])
plt.show()
99.8  pylab_examples example code: annotation_demo2.py
Matplotlib, Release 2.0.2

3

arrow
fancy

2
1
0
1
2
3

wedge

4
5

1>

0

1

arrow
simple
bubble
bubble,
contours
2

3

4

5

import matplotlib.pyplot as plt
from matplotlib.patches import Ellipse
import numpy as np

fig = plt.figure(1, figsize=(8, 5))
ax = fig.add_subplot(111, autoscale_on=False, xlim=(-1, 5), ylim=(-4, 3))
t = np.arange(0.0, 5.0, 0.01)
s = np.cos(2*np.pi*t)
line, = ax.plot(t, s, lw=3)
ax.annotate('straight',
xy=(0, 1), xycoords='data',
xytext=(-50, 30), textcoords='offset points',
arrowprops=dict(arrowstyle="->"))
ax.annotate('arc3,\nrad 0.2',
xy=(0.5, -1), xycoords='data',
xytext=(-80, -60), textcoords='offset points',
arrowprops=dict(arrowstyle="->",
connectionstyle="arc3,rad=.2"))
ax.annotate('arc,\nangle 50',
xy=(1., 1), xycoords='data',
xytext=(-90, 50), textcoords='offset points',

2346

Chapter 99. pylab_examples Examples


```python
arrowprops=dict(arrowstyle="->",
        connectionstyle="arc,angleA=0,armA=50,rad=10")

ax.annotate('arc,\naarms',
    xy=(1.5, -1), xycoords='data',
    xytext=(-80, -60), textcoords='offset points',
    arrowprops=dict(arrowstyle="->",
        connectionstyle="arc,angleA=0,armA=50,rad=10")
)

ax.annotate('angle,\nangle 90',
    xy=(2., 1), xycoords='data',
    xytext=(-70, 30), textcoords='offset points',
    arrowprops=dict(arrowstyle="->",
        connectionstyle="angle,angleA=0,angleB=90,rad=10")
)

ax.annotate('angle3,\nangle -90',
    xy=(2.5, -1), xycoords='data',
    xytext=(-80, -60), textcoords='offset points',
    arrowprops=dict(arrowstyle="->",
        connectionstyle="angle3,angleA=0,angleB=-90")
)

ax.annotate('angle,\nround',
    xy=(3., 1), xycoords='data',
    xytext=(-60, 30), textcoords='offset points',
    bbox=dict(boxstyle="round", fc="0.8"),
    arrowprops=dict(arrowstyle="->",
        connectionstyle="angle,angleA=0,angleB=90,rad=10")
)

ax.annotate('angle,\nround4',
    xy=(3.5, -1), xycoords='data',
    xytext=(-70, -80), textcoords='offset points',
    size=20,
    bbox=dict(boxstyle="round4, pad=.5", fc="0.8"),
    arrowprops=dict(arrowstyle="->",
        connectionstyle="angle,angleA=0,angleB=-90,rad=10")
)

ax.annotate('angle,\nshrink',
    xy=(4., 1), xycoords='data',
    xytext=(-60, 30), textcoords='offset points',
    bbox=dict(boxstyle="round", fc="0.8"),
    arrowprops=dict(arrowstyle="->",
        shrinkA=0, shrinkB=10,
        connectionstyle="angle,angleA=0,angleB=-90,rad=10")
)

# You can pass an empty string to get only annotation arrows rendered
ann = ax.annotate('', xy=(4., 1.), xycoords='data',
    xytext=(4.5, -1), textcoords='data',
    arrowprops=dict(arrowstyle="<->",
        connectionstyle="bar",
        ec="k",
        shrinkA=5, shrinkB=5")
)
```
Matplotlib, Release 2.0.2

fig = plt.figure(2)
fig.clf()
ax = fig.add_subplot(111, autoscale_on=False, xlim=(-1, 5), ylim=(-5, 3))
el = Ellipse((2, -1), 0.5, 0.5)
ax.add_patch(el)
ax.annotate('$->$',
xy=(2., -1), xycoords='data',
xytext=(-150, -140), textcoords='offset points',
bbox=dict(boxstyle="round", fc="0.8"),
arrowprops=dict(arrowstyle="->",
patchB=el,
connectionstyle="angle,angleA=90,angleB=0,rad=10"))
ax.annotate('arrow\nfancy',
xy=(2., -1), xycoords='data',
xytext=(-100, 60), textcoords='offset points',
size=20,
# bbox=dict(boxstyle="round", fc="0.8"),
arrowprops=dict(arrowstyle="fancy",
fc="0.6", ec="none",
patchB=el,
connectionstyle="angle3,angleA=0,angleB=-90"))
ax.annotate('arrow\nsimple',
xy=(2., -1), xycoords='data',
xytext=(100, 60), textcoords='offset points',
size=20,
# bbox=dict(boxstyle="round", fc="0.8"),
arrowprops=dict(arrowstyle="simple",
fc="0.6", ec="none",
patchB=el,
connectionstyle="arc3,rad=0.3"))
ax.annotate('wedge',
xy=(2., -1), xycoords='data',
xytext=(-100, -100), textcoords='offset points',
size=20,
# bbox=dict(boxstyle="round", fc="0.8"),
arrowprops=dict(arrowstyle="wedge,tail_width=0.7",
fc="0.6", ec="none",
patchB=el,
connectionstyle="arc3,rad=-0.3"))
ann = ax.annotate('bubble,\ncontours',
xy=(2., -1), xycoords='data',
xytext=(0, -70), textcoords='offset points',
size=20,
bbox=dict(boxstyle="round",
fc=(1.0, 0.7, 0.7),
ec=(1., .5, .5)),

2348

Chapter 99. pylab_examples Examples


```
arrowprops=dict(arrowstyle="wedge,tail_width=1.",
                fc=(1.0, 0.7, 0.7), ec=(1., .5, .5),
                patchA=None,
                patchB=el,
                relpos=(0.2, 0.8),
                connectionstyle="arc3,rad=-0.1")

ann = ax.annotate('bubble',
                   xy=(2., -1), xycoords='data',
                   xytext=(55, 0), textcoords='offset points',
                   size=20, va="center",
                   bbox=dict(boxstyle="round", fc=(1.0, 0.7, 0.7), ec="none"),
                   arrowprops=dict(arrowstyle="wedge,tail_width=1.",
                                   fc=(1.0, 0.7, 0.7), ec="none",
                                   patchA=None,
                                   patchB=el,
                                   relpos=(0.2, 0.5)))
```

plt.show()
import matplotlib.pyplot as plt
from matplotlib.text import OffsetFrom

fig, (ax1, ax2) = plt.subplots(1, 2)

bbox_args = dict(boxstyle="round", fc="0.8")
arrow_args = dict(arrowstyle="->")

ax1.annotate('figure fraction : 0, 0', xy=(0, 0), xycoords='figure fraction',
             xytext=(20, 20), textcoords='offset points',
             ha="left", va="bottom",
             bbox=bbox_args,
             arrowprops=arrow_args)

ax1.annotate('figure fraction : 1, 1', xy=(1, 1), xycoords='figure fraction',
             xytext=(-20, -20), textcoords='offset points',
             ha="right", va="top",
             bbox=bbox_args,
             arrowprops=arrow_args)

ax1.annotate('axes fraction : 0, 0', xy=(0, 0), xycoords='axes fraction',
             xytext=(20, 20), textcoords='offset points',
             ha="left", va="bottom",
             bbox=bbox_args,
             arrowprops=arrow_args)

ax1.annotate('axes fraction : 1, 1', xy=(1, 1), xycoords='axes fraction',
             xytext=(-20, -20), textcoords='offset points',
             ha="right", va="top",
             bbox=bbox_args,
             arrowprops=arrow_args)

an1 = ax1.annotate('Drag me 1', xy=(.5, .7), xycoords='data',
                   xytext=(.5, .7), textcoords='data',
                   ha="center", va="center",
                   bbox=bbox_args,
                   arrowprops=arrow_args)

an2 = ax1.annotate('Drag me 2', xy=(.5, .5), xycoords=an1,
                   xytext=(.5, .3), textcoords='axes fraction',
                   ha="center", va="center",
                   bbox=bbox_args,
                   arrowprops=dict(patchB=an1.get_bbox_patch(),
                                    connectionstyle="arc3,rad=0.2",
                                    **arrow_args))

an3 = ax1.annotate('', xy=(.5, .5), xycoords=an2,
                   xytext=(.5, .5), textcoords=an1,
                   ha="center", va="center",
                   bbox=bbox_args,
arrowprops=dict(patchA=an1.get_bbox_patch(),
               patchB=an2.get_bbox_patch(),
               connectionstyle="arc3,rad=0.2",
               **arrow_args))

ax2.annotate('xy=(0, 1)\nxcoords=\"data\", \"axes fraction\"',
              xy=(0, 1), xycoords=\"data\", \"axes fraction\",
              xytext=(0, -20), textcoords='offset points',
              ha=\"center\", va=\"top\",
              bbox=bbox_args,
              arrowprops=arrow_args)

ax2.annotate('xy=(0.5, 0)\nxcoords=artist',
              xy=(0.5, 0.), xycoords=t,
              xytext=(0, -20), textcoords='offset points',
              ha=\"center\", va=\"top\",
              bbox=bbox_args,
              arrowprops=arrow_args)

ax2.annotate('xy=(0.8, 0.5)\nxcoords=ax1.transData',
              xy=(0.8, 0.5), xycoords=ax1.transData,
              xytext=(10, 10),
              textcoords=OffsetFrom(ax2.bbox, (0, 0), \"points\"),
              ha=\"left\", va=\"bottom\",
              bbox=bbox_args,
              arrowprops=arrow_args)

ax2.set_xlim([-2, 2])
ax2.set_ylim([-2, 2])

an1.draggable()
an2.draggable()
 plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
import matplotlib.pyplot as plt
import numpy as np

x = np.array([10, 8, 13, 9, 11, 14, 6, 4, 12, 7, 5])
y1 = np.array([8.04, 6.95, 7.58, 8.81, 8.33, 9.96, 7.24, 4.26, 10.84, 4.82, 5.68])
y2 = np.array([9.14, 8.14, 8.74, 8.77, 9.26, 8.10, 6.13, 3.10, 9.13, 7.26, 4.74])
y3 = np.array([7.46, 6.77, 12.74, 7.11, 7.81, 8.84, 6.08, 5.39, 8.15, 6.42, 5.73])
x4 = np.array([8, 8, 8, 8, 8, 8, 19, 8, 8, 8, 8])
y4 = np.array([6.58, 5.76, 7.71, 8.84, 8.47, 7.04, 5.25, 12.50, 5.56, 7.91, 6.89])

def fit(x):
    return 3 + 0.5 * x

---

Edward Tufte uses this example from Anscombe to show 4 datasets of x and y that have the same mean, standard deviation, and regression line, but which are qualitatively different.

matplotlib fun for a rainy day

from __future__ import print_function

---

Chapter 99. *pylab_examples* Examples

2352
xfit = np.array([np.amin(x), np.amax(x)])

plt.subplot(221)
plt.plot(x, y1, 'ks', xfit, fit(xfit), 'r-', lw=2)
plt.axis([2, 20, 2, 14])
plt.setp(plt.gca(), xticklabels=[], yticks=(4, 8, 12), xticks=(0, 10, 20))
plt.text(3, 12, 'I', fontsize=20)

plt.subplot(222)
plt.plot(x, y2, 'ks', xfit, fit(xfit), 'r-', lw=2)
plt.axis([2, 20, 2, 14])
plt.setp(plt.gca(), xticklabels=[], yticks=(4, 8, 12), yticklabels=[], xticks=(0, 10, 20))
plt.text(3, 12, 'II', fontsize=20)

plt.subplot(223)
plt.plot(x, y3, 'ks', xfit, fit(xfit), 'r-', lw=2)
plt.axis([2, 20, 2, 14])
plt.setp(plt.gca(), yticks=(4, 8, 12), xticks=(0, 10, 20))
plt.text(3, 12, 'III', fontsize=20)

plt.subplot(224)

xfit = np.array([np.amin(x4), np.amax(x4)])
plt.plot(x4, y4, 'ks', xfit, fit(xfit), 'r-', lw=2)
plt.axis([2, 20, 2, 14])
plt.setp(plt.gca(), yticklabels=[], yticks=(4, 8, 12), xticks=(0, 10, 20))
plt.text(3, 12, 'IV', fontsize=20)

# verify the stats
pairs = (x, y1), (x, y2), (x, y3), (x4, y4)
for x, y in pairs:
    print('mean=%1.2f, std=%1.2f, r=%1.2f' % (np.mean(y), np.std(y), np.corrcoef(x, y)[0][1]))

plt.show()
import matplotlib.pyplot as plt
import numpy as np

def f(t):
    # a damped exponential
    s1 = np.cos(2 * np.pi * t)
    e1 = np.exp(-t)
    return s1 * e1

t1 = np.arange(0.0, 5.0, .2)

l = plt.plot(t1, f(t1), 'ro')
plt.setp(l, 'markersize', 30)
plt.setp(l, 'markerfacecolor', 'b')
plt.show()
Arrow drawing example for the new fancy_arrow facilities.

Code contributed by: Rob Knight <rob@spot.colorado.edu>

usage:

    python arrow_demo.py realistic|full|sample|extreme

import matplotlib.pyplot as plt
import numpy as np

rates_to_bases = {'r1': 'AT', 'r2': 'TA', 'r3': 'GA', 'r4': 'AG', 'r5': 'CA',
                 'r6': 'AC', 'r7': 'GT', 'r8': 'TG', 'r9': 'CT', 'r10': 'TC',
                 'r11': 'GC', 'r12': 'CG'}

numbered_bases_to_rates = dict(((v, k) for k, v in rates_to_bases.items()))
lettered_bases_to_rates = dict(((v, 'r' + v) for k, v in rates_to_bases.items()))

def add_dicts(d1, d2):
    """Adds two dicts and returns the result."""
    result = d1.copy()
    result.update(d2)
    return result
def make_arrow_plot(data, size=4, display='length', shape='right',
max_arrow_width=0.03, arrow_sep=0.02, alpha=0.5,
normalize_data=False, ec=None, labelcolor=None,
head_starts_at_zero=True, rate_labels=lettered_bases_to_rates,
**kwargs):
    """Makes an arrow plot.

Parameters:

data: dict with probabilities for the bases and pair transitions.
size: size of the graph in inches.
display: 'length', 'width', or 'alpha' for arrow property to change.
shape: 'full', 'left', or 'right' for full or half arrows.
max_arrow_width: maximum width of an arrow, data coordinates.
arrow_sep: separation between arrows in a pair, data coordinates.
alpha: maximum opacity of arrows, default 0.8.

**kwargs can be anything allowed by a Arrow object, e.g.
linewidth and edgecolor.
"""

plt.xlim(-0.5, 1.5)
plt.ylim(-0.5, 1.5)
plt.gcf().set_size_inches(size, size)
plt.xticks([])
plt.yticks([])
max_text_size = size*12
min_text_size = size
label_text_size = size*2.5
text_params = {'ha': 'center', 'va': 'center', 'family': 'sans-serif',
               'fontweight': 'bold'}

r2 = np.sqrt(2)
deltas = {
    'AT': (1, 0),
    'TA': (-1, 0),
    'GA': (0, 1),
    'AG': (0, -1),
    'CA': (-1/r2, 1/r2),
    'AC': (1/r2, -1/r2),
    'GT': (1/r2, 1/r2),
    'TG': (-1/r2, -1/r2),
    'CT': (0, 1),
    'TC': (0, -1),
    'GC': (1, 0),
    'CG': (-1, 0)
}
colors = {
    'AT': 'r',
    'TA': 'k',
}
'GA': 'g',
'AG': 'r',
'CA': 'b',
'AC': 'r',
'GT': 'g',
'TG': 'k',
'CT': 'b',
'TC': 'k',
'GC': 'g',
'CG': 'b'
}

label_positions = {
    'AT': 'center',
    'TA': 'center',
    'GA': 'center',
    'AG': 'center',
    'CA': 'left',
    'AC': 'left',
    'GT': 'left',
    'TG': 'left',
    'CT': 'center',
    'TC': 'center',
    'GC': 'center',
    'CG': 'center'
}

def do_fontsize(k):
    return float(np.clip(max_text_size*np.sqrt(data[k]),
                          min_text_size, max_text_size))

A = plt.text(0, 1, '$A_3$', color='r', size=do_fontsize('A'), **text_params)
T = plt.text(1, 1, '$T_3$', color='k', size=do_fontsize('T'), **text_params)
G = plt.text(0, 0, '$G_3$', color='g', size=do_fontsize('G'), **text_params)
C = plt.text(1, 0, '$C_3$', color='b', size=do_fontsize('C'), **text_params)

arrow_h_offset = 0.25  # data coordinates, empirically determined
max_arrow_length = 1 - 2*arrow_h_offset

max_arrow_width = max_arrow_width
max_head_width = 2.5*max_arrow_width
max_head_length = 2*max_arrow_width

arrow_params = {'length_includes_head': True, 'shape': shape,
                 'head_starts_at_zero': head_starts_at_zero}

ax = plt.gca()
sf = 0.6  # max arrow size represents this in data coords

d = (r2/2 + arrow_h_offset - 0.5)/r2  # distance for diags
r2v = arrow_sep/r2  # offset for diags

# tuple of x, y for start position
positions = {
    'AT': (arrow_h_offset, 1 + arrow_sep),
if normalize_data:
    # find maximum value for rates, i.e. where keys are 2 chars long
    max_val = 0
    for k, v in data.items():
        if len(k) == 2:
            max_val = max(max_val, v)
    # divide rates by max val, multiply by arrow scale factor
    for k, v in data.items():
        data[k] = v/max_val*sf

def draw_arrow(pair, alpha=alpha, ec=ec, labelcolor=labelcolor):
    # set the length of the arrow
    if display == 'length':
        length = max_head_length + data[pair]/sf*(max_arrow_length - max_head_length)
    else:
        length = max_arrow_length
    # set the transparency of the arrow
    if display == 'alph':
        alpha = min(data[pair]/sf, alpha)
    else:
        alpha = alpha
    # set the width of the arrow
    if display == 'width':
        scale = data[pair]/sf
        width = max_arrow_width*scale
        head_width = max_head_width*scale
        head_length = max_head_length*scale
    else:
        width = max_arrow_width
        head_width = max_head_width
        head_length = max_head_length

    fc = colors[pair]
    ec = ec or fc

    x_scale, y_scale = deltas[pair]
    x_pos, y_pos = positions[pair]
    plt.arrow(x_pos, y_pos, x_scale*length, y_scale*length,
              fc=fc, ec=ec, alpha=alpha, width=width, head_width=head_width,
head_length=head_length, **arrow_params)

# figure out coordinates for text
# if drawing relative to base: x and y are same as for arrow
# dx and dy are one arrow width left and up
# need to rotate based on direction of arrow, use x_scale and y_scale
# as sin x and cos x?
sx, cx = y_scale, x_scale

where = label_positions[pair]
if where == 'left':
    orig_position = 3*np.array([[max_arrow_width, max_arrow_width]])
elif where == 'absolute':
    orig_position = np.array([[max_arrow_length/2.0, 3*max_arrow_width]])
elif where == 'right':
    orig_position = np.array([[length - 3*max_arrow_width,
                              3*max_arrow_width]])
elif where == 'center':
    orig_position = np.array([[length/2.0, 3*max_arrow_width]])
else:
    raise ValueError("Got unknown position parameter %s" % where)

M = np.array([[cx, sx], [-sx, cx]])
coords = np.dot(orig_position, M) + [[x_pos, y_pos]]
x, y = np.ravel(coords)
orig_label = rate_labels[pair]
label = '%s_{\mathrm{\math{\%s}}}$ % (orig_label[0], orig_label[1:])

plt.text(x, y, label, size=label_text_size, ha='center', va='center',
          color=labelcolor or fc)

for p in sorted(positions):
    draw_arrow(p)

# test data
all_on_max = dict((i, 1) for i in 'TCAG') +
                 [(i + j, 0.6) for i in 'TCAG' for j in 'TCAG'])
realistic_data = {
    'A': 0.4,
    'T': 0.3,
    'G': 0.5,
    'C': 0.2,
    'AT': 0.4,
    'AC': 0.3,
    'AG': 0.2,
    'TA': 0.2,
    'TC': 0.3,
    'TG': 0.4,
    'CT': 0.2,
    'CG': 0.3,
    'CA': 0.2,
    'GA': 0.1,
extreme_data = {
    'A': 0.75,
    'T': 0.1,
    'G': 0.1,
    'C': 0.05,
    'AT': 0.6,
    'AC': 0.3,
    'AG': 0.1,
    'TA': 0.02,
    'TC': 0.3,
    'TG': 0.01,
    'CT': 0.2,
    'CG': 0.5,
    'CA': 0.2,
    'GA': 0.1,
    'GT': 0.4,
    'GC': 0.2,
}

sample_data = {
    'A': 0.2137,
    'T': 0.3541,
    'G': 0.1946,
    'C': 0.2376,
    'AT': 0.0228,
    'AC': 0.0684,
    'AG': 0.2056,
    'TA': 0.0315,
    'TC': 0.0629,
    'TG': 0.0315,
    'CT': 0.1355,
    'CG': 0.0401,
    'CA': 0.0703,
    'GA': 0.1824,
    'GT': 0.0387,
    'GC': 0.1106,
}

if __name__ == '__main__':
    from sys import argv
    d = None
    if len(argv) > 1:
        if argv[1] == 'full':
            d = all_on_max
            scaled = False
        elif argv[1] == 'extreme':
            d = extreme_data
            scaled = False
```python
elif argv[1] == 'realistic':
    d = realistic_data
    scaled = False
elif argv[1] == 'sample':
    d = sample_data
    scaled = True
if d is None:
    d = all_on_max
    scaled = False
if len(argv) > 2:
    display = argv[2]
else:
    display = 'length'

size = 4
plt.figure(figsize=(size, size))

make_arrow_plot(d, display=display, linewidth=0.001, edgecolor=None,
                normalize_data=scaled, head_starts_at_zero=True, size=size)

plt.draw()
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
import matplotlib.pyplot as plt

ax = plt.axes()
ax.arrow(0, 0, 0.5, 0.5, head_width=0.05, head_length=0.1, fc='k', ec='k')
plt.show()
import matplotlib.pyplot as plt

fig, (ax1, ax2) = plt.subplots(1, 2)
ax1.set_xscale("log")
ax1.set_yscale("log")
ax1.set_xlim(1e1, 1e3)
ax1.set_ylim(1e2, 1e3)
ax1.set_aspect(1)
ax1.set_title("adjustable = box")

ax2.set_xscale("log")
ax2.set_yscale("log")
ax2.set_adjustable("datalim")
ax2.plot([1, 3, 10], [1, 9, 100], "o-")
ax2.set_xlim(1e-1, 1e2)
ax2.set_ylim(1e-1, 1e3)
ax2.set_aspect(1)
ax2.set_title("adjustable = datalim")

plt.draw()
plt.show()
import matplotlib.pyplot as plt
import numpy as np

# create some data to use for the plot
dt = 0.001
t = np.arange(0.0, 10.0, dt)
r = np.exp(-t[:1000]/0.05)  # impulse response
x = np.random.randn(len(t))
s = np.convolve(x, r)[:len(x)]*dt  # colored noise

# the main axes is subplot(111) by default
plt.plot(t, s)
plt.axis([0, 1, 1.1*np.amin(s), 2*np.amax(s)])
plt.xlabel('time (s)')
plt.ylabel('current (nA)')
plt.title('Gaussian colored noise')

# this is an inset axes over the main axes
a = plt.axes([.65, .6, .2, .2], facecolor='y')
n, bins, patches = plt.hist(s, 400, normed=1)
plt.title('Probability')
plt.xticks([])
plt.yticks([])

# this is another inset axes over the main axes
a = plt.axes([0.2, 0.6, .2, .2], facecolor='y')
plt.plot(t[:len(r)], r)
plt.title('Impulse response')
plt.xlim(0, 0.2)
plt.xticks([])
plt.yticks([])
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

99.16  pylab_examples example code: axes_props.py

"""
You can control the axis tick and grid properties
"""
import matplotlib.pyplot as plt
import numpy as np

t = np.arange(0.0, 2.0, 0.01)
s = np.sin(2 * np.pi * t)
fig, ax = plt.subplots()
ax.plot(t, s)
ax.grid(True)

ticklines = ax.get_xticklines() + ax.get_yticklines()
gridlines = ax.get_xgridlines() + ax.get_ygridlines()
ticklabels = ax.get_xticklabels() + ax.get_yticklabels()

for line in ticklines:
    line.set_linewidth(3)

for line in gridlines:
    line.set_linestyle('-.')

for label in ticklabels:
    label.set_color('r')
    label.set_fontsize('medium')

plt.show()
```python
from matplotlib.transforms import Bbox, TransformedBbox, 
    blended_transform_factory

from mpl_toolkits.axes_grid1.inset_locator import BboxPatch, BboxConnector,
    BboxConnectorPatch

def connect_bbox(bbox1, bbox2, 
    loc1a, loc2a, loc1b, loc2b, 
    prop_lines, prop_patches=None):
    if prop_patches is None:
        prop_patches = prop_lines.copy()
        prop_patches["alpha"] = prop_patches.get("alpha", 1)*0.2

    c1 = BboxConnector(bbox1, bbox2, loc1=loc1a, loc2=loc2a, **prop_lines)
    c1.set_clip_on(False)
    c2 = BboxConnector(bbox1, bbox2, loc1=loc1b, loc2=loc2b, **prop_lines)
    c2.set_clip_on(False)
```
def zoom_effect01(ax1, ax2, xmin, xmax, **kwargs):
    ""
    ax1 : the main axes
    ax1 : the zoomed axes
    (xmin,xmax) : the limits of the colored area in both plot axes.
    
    connect ax1 & ax2. The x-range of (xmin, xmax) in both axes will be marked. The keywords parameters will be used to create patches.
    ""
    trans1 = blended_transform_factory(ax1.transData, ax1.transAxes)
    trans2 = blended_transform_factory(ax2.transData, ax2.transAxes)
    bbox = Bbox.from_extents(xmin, 0, xmax, 1)
    mybbox1 = TransformedBbox(bbox, trans1)
    mybbox2 = TransformedBbox(bbox, trans2)
    prop_patches = kwargs.copy()
    prop_patches["ec"] = "none"
    prop_patches["alpha"] = 0.2
    c1, c2, bbox_patch1, bbox_patch2, p = connect_bbox(mybbox1, mybbox2,
        loc1a=3, loc2a=2, loc1b=4, loc2b=1,
        prop_lines=kwargs, prop_patches=prop_patches)
    ax1.add_patch(bbox_patch1)
    ax2.add_patch(bbox_patch2)
    ax2.add_patch(c1)
    ax2.add_patch(c2)
    ax2.add_patch(p)
    return c1, c2, bbox_patch1, bbox_patch2, p

def zoom_effect02(ax1, ax2, **kwargs):
    ""
    ax1 : the main axes
    ""
ax1 : the zoomed axes

Similar to zoom_effect01. The xmin & xmax will be taken from the ax1.viewLim.

```python
import matplotlib.pyplot as plt
plt.figure(1, figsize=(5, 5))
ax1 = plt.subplot(221)
ax2 = plt.subplot(212)
ax2.set_xlim(0, 1)
ax2.set_xlim(0, 5)
zoom_effect01(ax1, ax2, 0.2, 0.8)
ax1 = plt.subplot(222)
ax1.set_xlim(2, 3)
ax2.set_xlim(0, 5)
zoom_effect02(ax1, ax2)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
import numpy as np
import matplotlib.pyplot as plt

t = np.arange(-1, 2, .01)
s = np.sin(2*np.pi*t)

plt.plot(t, s)
# draw a thick red hline at y=0 that spans the xrange
l = plt.axhline(linewidth=8, color='#d62728')

# draw a default hline at y=1 that spans the xrange
l = plt.axhline(y=1)

# draw a default vline at x=1 that spans the yrange
l = plt.axvline(x=1)

# draw a thick blue vline at x=0 that spans the upper quadrant of
# the yrange
l = plt.axvline(x=0, ymin=0.75, linewidth=8, color='#1f77b4')

# draw a default hline at y=.5 that spans the middle half of
# the axes
```python
l = plt.axhline(y=.5, xmin=0.25, xmax=0.75)
p = plt.axhspan(0.25, 0.75, facecolor='0.5', alpha=0.5)
p = plt.axvspan(1.25, 1.55, facecolor='#2ca02c', alpha=0.5)
plt.axis([-1, 2, -1, 2])
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

99.19 *pylab_examples* example code: `axis_equal_demo.py`

```
"This example is only interesting when ran in interactive mode"

import matplotlib.pyplot as plt
import numpy as np

# Plot circle or radius 3
```
an = np.linspace(0, 2*np.pi, 100)

plt.subplot(221)
plt.plot(3*np.cos(an), 3*np.sin(an))
plt.title('not equal, looks like ellipse', fontsize=10)

plt.subplot(222)
plt.plot(3*np.cos(an), 3*np.sin(an))
plt.axis('equal')
plt.title('equal, looks like circle', fontsize=10)

plt.subplot(223)
plt.plot(3*np.cos(an), 3*np.sin(an))
plt.axis('equal')
plt.axis([-3, 3, -3, 3])
plt.title('looks like circle, even after changing limits', fontsize=10)

plt.subplot(224)
plt.plot(3*np.cos(an), 3*np.sin(an))
plt.axis('equal')
plt.axis([-3, 3, -3, 3])
plt.plot([0, 4], [0, 4])
plt.title('still equal after adding line', fontsize=10)

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
# a stacked bar plot with errorbars
import numpy as np
import matplotlib.pyplot as plt

N = 5
menMeans = (20, 35, 30, 35, 27)  
womenMeans = (25, 32, 34, 20, 25)  
menStd = (2, 3, 4, 1, 2)  
womenStd = (3, 5, 2, 3, 3)  
ind = np.arange(N)  # the x locations for the groups
width = 0.35  # the width of the bars: can also be len(x) sequence

p1 = plt.bar(ind, menMeans, width, color='#d62728', yerr=menStd)
p2 = plt.bar(ind, womenMeans, width,  
bottom=menMeans, yerr=womenStd)

plt.ylabel('Scores')  
plt.title('Scores by group and gender')  
plt.xticks(ind, ('G1', 'G2', 'G3', 'G4', 'G5'))  
plt.yticks(np.arange(0, 81, 10))  
plt.legend((p1[0], p2[0]), ('Men', 'Women'))
plt.show()  

Keywords: python, matplotlib, pylab, example, codex (see Search examples)  

99.21 pylab_examples example code: barb_demo.py
Demonstration of wind barb plots

```python
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(-5, 5, 5)
X, Y = np.meshgrid(x, x)
U, V = 12*X, 12*Y

data = [(-1.5, .5, -6, -6),
        (1, -1, -46, 46),
        (-3, -1, 11, -11),
        (1, 1.5, 80, 80),
        (0.5, 0.25, 25, 15),
        (-1.5, -0.5, -5, 40)]

data = np.array(data, dtype=[('x', np.float32), ('y', np.float32),
                           ('u', np.float32), ('v', np.float32)])

# Default parameters, uniform grid
ax = plt.subplot(2, 2, 1)
ax.barbs(X, Y, U, V)

# Arbitrary set of vectors, make them longer and change the pivot point
#(point around which they're rotated) to be the middle
```
ax = plt.subplot(2, 2, 2)
ax.barbs(data['x'], data['y'], data['u'], data['v'], length=8, pivot='middle')

# Showing colormapping with uniform grid. Fill the circle for an empty barb, 
# don't round the values, and change some of the size parameters
ax = plt.subplot(2, 2, 3)
ax.barbs(X, Y, U, V, np.sqrt(U*U + V*V), fill_empty=True, rounding=False, 
       sizes=dict(emptybarb=0.25, spacing=0.2, height=0.3))

# Change colors as well as the increments for parts of the barbs
ax = plt.subplot(2, 2, 4)
ax.barbs(data['x'], data['y'], data['u'], data['v'], flagcolor='r', 
          barbcolor=['b', 'g'], barb_increments=dict(half=10, full=20, flag=100), 
          flip_barb=True)

# Masked arrays are also supported
masked_u = np.ma.masked_array(data['u'])
masked_u[4] = 1000  # Bad value that should not be plotted when masked
masked_u[4] = np.ma.masked

# Identical plot to panel 2 in the first figure, but with the point at 
#(0.5, 0.25) missing (masked)
fig2 = plt.figure()
ax = fig2.add_subplot(1, 1, 1)
avx.barbs(data['x'], data['y'], masked_u, data['v'], length=8, pivot='middle')

plt.show()
Bar chart demo with pairs of bars grouped for easy comparison.

```python
import numpy as np
import matplotlib.pyplot as plt

n_groups = 5
means_men = (20, 35, 30, 35, 27)
std_men = (2, 3, 4, 1, 2)
means_women = (25, 32, 34, 20, 25)
std_women = (3, 5, 2, 3, 3)

fig, ax = plt.subplots()
index = np.arange(n_groups)
bar_width = 0.35
opacity = 0.4
error_config = {'ecolor': '0.3'}
```

Scores by group and gender

<table>
<thead>
<tr>
<th>Group</th>
<th>Men</th>
<th>Women</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>20</td>
<td>25</td>
</tr>
<tr>
<td>B</td>
<td>35</td>
<td>32</td>
</tr>
<tr>
<td>C</td>
<td>30</td>
<td>34</td>
</tr>
<tr>
<td>D</td>
<td>35</td>
<td>20</td>
</tr>
<tr>
<td>E</td>
<td>27</td>
<td>25</td>
</tr>
</tbody>
</table>
rects1 = plt.bar(index, means_men, bar_width,
    alpha=opacity,
    color='b',
    yerr=std_men,
    error_kw=error_config,
    label='Men')

rects2 = plt.bar(index + bar_width, means_women, bar_width,
    alpha=opacity,
    color='r',
    yerr=std_women,
    error_kw=error_config,
    label='Women')

plt.xlabel('Group')
plt.ylabel('Scores')
plt.title('Scores by group and gender')
plt.xticks(index + bar_width / 2, ('A', 'B', 'C', 'D', 'E'))
plt.legend()
plt.tight_layout()
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Thanks Josh Hemann for the example

This example comes from an application in which grade school gym teachers wanted to be able to show parents how their child did across a handful of fitness tests, and importantly, relative to how other children did. To extract the plotting code for demo purposes, we'll just make up some data for little Johnny Doe...

```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.ticker import MaxNLocator
from collections import namedtuple

Student = namedtuple('Student', ['name', 'grade', 'gender'])
Score = namedtuple('Score', ['score', 'percentile'])

# GLOBAL CONSTANTS
```
testNames = ['Pacer Test', 'Flexed Arm
Hang', 'Mile Run', 'Agility', 'Push Ups']
testMeta = dict(zip(testNames, ['laps', 'sec', 'min:sec', 'sec', '']))

def attach_ordinal(num):
    """helper function to add ordinal string to integers

    1 -> 1st
    56 -> 56th
    """
    suffixes = dict((str(i), v) for i, v in enumerate(['th', 'st', 'nd', 'rd', 'th', 'th', 'th', 'th', 'th']))
    v = str(num)
    # special case early teens
    if v in {'11', '12', '13'}:
        return v + 'th'
    return v + suffixes[v[-1]]

def format_score(scr, test):
    """Build up the score labels for the right Y-axis by first appending a carriage return to each string and then tacking on the appropriate meta information (i.e., 'laps' vs 'seconds'). We want the labels centered on the ticks, so if there is no meta info (like for pushups) then don't add the carriage return to the string
    """
    md = testMeta[test]
    if md:
        return '{0}
{1}'.format(scr, md)
    else:
        return scr

def format_ycursor(y):
    y = int(y)
    if y < 0 or y >= len(testNames):
        return ''
    else:
        return testNames[y]

def plot_student_results(student, scores, cohort_size):
    # create the figure
    fig, ax1 = plt.subplots(figsize=(9, 7))
    fig.subplots_adjust(left=0.115, right=0.88)
    fig.canvas.set_window_title('Eldorado K-8 Fitness Chart')
    pos = np.arange(len(testNames))
rects = ax1.barh(pos, [scores[k].percentile for k in testNames],
                        align='center',
                        height=0.5, color='m',
                        tick_label=testNames)

ax1.set_title(student.name)

ax1.set_xlim([0, 100])
ax1.xaxis.set_major_locator(MaxNLocator(11))
ax1.xaxis.grid(True, linestyle='--', which='major',
                        color='grey', alpha=.25)

# Plot a solid vertical gridline to highlight the median position
ax1.axvline(50, color='grey', alpha=0.25)

# set X-axis tick marks at the deciles
cohort_label = ax1.text(.5, -.07, 'Cohort Size: {0}'.format(cohort_size),
                        horizontalalignment='center', size='small',
                        transform=ax1.transAxes)

# Set the right-hand Y-axis ticks and labels
ax2 = ax1.twinx()
scoreLabels = [format_score(scores[k].score, k) for k in testNames]

# set the tick locations
ax2.set_yticks(pos)

# make sure that the limits are set equally on both yaxis so the
ticks line up
ax2.set_ylim(ax1.get_ylim())

# set the tick labels
ax2.set_yticklabels(scoreLabels)

ax2.set_ylabel('Test Scores')

ax2.set_xlabel(('Percentile Ranking Across ' '{grade} Grade {gender}s').format(
                        grade=attach_ordinal(student.grade),
                        gender=student.gender.title()))

rect_labels = []
# Lastly, write in the ranking inside each bar to aid in interpretation
for rect in rects:
    # Rectangle widths are already integer-valued but are floating
    # type, so it helps to remove the trailing decimal point and 0 by
    # converting width to int type
    width = int(rect.get_width())

    rankStr = attach_ordinal(width)
    # The bars aren't wide enough to print the ranking inside
    if (width < 5):
        # Shift the text to the right side of the right edge
student = Student('Johnny Doe', 2, 'boy')
scores = dict(zip(testNames, (Score(v, p) for v, p in zip(['7', '48', '12:52', '17', '14'], np.round(np.random.uniform(0, 1, len(testNames))*100, 0)))))
cohort_size = 62  # The number of other 2nd grade boys

arts = plot_student_results(student, scores, cohort_size)
plt.show()
import matplotlib.pyplot as plt
import numpy as np

# the bar
x = np.where(np.random.rand(500) > 0.7, 1.0, 0.0)

axprops = dict(xticks=[], yticks=[])  # The bar is broken at present

ax = fig.add_axes([0.1, 0.3, 0.1, 0.6], **axprops)
ax.imshow(x, **barprops)

x = x.copy()

# a horizontal barcode
x.shape = 1, len(x)
ax = fig.add_axes([0.3, 0.1, 0.6, 0.1], **axprops)
ax.imshow(x, **barprops)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

99.25 pylab_examples example code: boxplot_demo.py
1

-100 -50 0 50 100 150 200
import matplotlib.pyplot as plt
import numpy as np

# fake up some data
spread = np.random.rand(50) * 100
center = np.ones(25) * 50
flier_high = np.random.rand(10) * 100 + 100
flier_low = np.random.rand(10) * -100
data = np.concatenate((spread, center, flier_high, flier_low), 0)

# basic plot
plt.boxplot(data)

# notched plot
plt.figure()
plt.boxplot(data, 1)

# change outlier point symbols
plt.figure()
plt.boxplot(data, 0, 'gD')

# don't show outlier points
plt.figure()
plt.boxplot(data, 0, '')

# horizontal boxes
plt.figure()
plt.boxplot(data, 0, 'rs', 0)

# change whisker length
plt.figure()
plt.boxplot(data, 0, 'rs', 0, 0.75)

# fake up some more data
spread = np.random.rand(50) * 100
center = np.ones(25) * 40
flier_high = np.random.rand(10) * 100 + 100
flier_low = np.random.rand(10) * -100

d2 = np.concatenate((spread, center, flier_high, flier_low), 0)
data.shape = (-1, 1)
d2.shape = (-1, 1)

# data = concatenate( (data, d2), 1 )
# Making a 2-D array only works if all the columns are the
# same length. If they are not, then use a list instead.
# This is actually more efficient because boxplot converts
# a 2-D array into a list of vectors internally anyway.
data = [data, d2, d2[::2, 0]]

# multiple box plots on one figure
plt.figure()
plt.boxplot(data)

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.patches import Polygon

# Generate some data from five different probability distributions,  
# each with different characteristics. We want to play with how an IID  
# bootstrap resample of the data preserves the distributional  
# properties of the original sample, and a boxplot is one visual tool  
# to make this assessment
numDists = 5
randomDists = ['Normal(1,1)', 'Lognormal(1,1)', 'Exp(1)', 'Gumbel(6,4)',  
               'Triangular(2,9,11)']
N = 500
np.random.seed(0)
norm = np.random.normal(1, 1, N)
logn = np.random.lognormal(1, 1, N)
expo = np.random.exponential(1, N)
gumb = np.random.gumbel(6, 4, N)
tria = np.random.triangular(2, 9, 11, N)

# Generate some random indices that we'll use to resample the original data

Thanks Josh Hemann for the example
# arrays. For code brevity, just use the same random indices for each array
bootstrapIndices = np.random.random_integers(0, N - 1, N)
normBoot = norm[bootstrapIndices]
expoBoot = expo[bootstrapIndices]
gumbBoot = gumb[bootstrapIndices]
lognBoot = logn[bootstrapIndices]
triaBoot = tria[bootstrapIndices]
data = [norm, normBoot, logn, lognBoot, expo, expoBoot, gumb, gumbBoot,
    tria, triaBoot]

fig, ax1 = plt.subplots(figsize=(10, 6))
fig.canvas.set_window_title('A Boxplot Example')
plt.subplots_adjust(left=0.075, right=0.95, top=0.9, bottom=0.25)

bp = plt.boxplot(data, notch=0, sym='+', vert=1, whis=1.5)
plt.setp(bp['boxes'], color='black')
plt.setp(bp['whiskers'], color='black')
plt.setp(bp['fliers'], color='red', marker='+')

# Add a horizontal grid to the plot, but make it very light in color
# so we can use it for reading data values but not be distracting
ax1.yaxis.grid(True, linestyle='-', which='major', color='lightgrey',
    alpha=0.5)

# Hide these grid behind plot objects
ax1.set_axisbelow(True)
ax1.set_title('Comparison of IID Bootstrap Resampling Across Five Distributions')
ax1.set_xlabel('Distribution')
ax1.set_ylabel('Value')

# Now fill the boxes with desired colors
boxColors = ['darkkhaki', 'royalblue']
numBoxes = numDists*2
medians = list(range(numBoxes))
for i in range(numBoxes):
    box = bp['boxes'][i]
    boxX = []
    boxY = []
    for j in range(5):
        boxX.append(box.get_xdata()[j])
        boxY.append(box.get_ydata()[j])
    boxCoords = list(zip(boxX, boxY))
    # Alternate between Dark Khaki and Royal Blue
    k = i % 2
    boxPolygon = Polygon(boxCoords, facecolor=boxColors[k])
    ax1.add_patch(boxPolygon)
    # Now draw the median lines back over what we just filled in
    med = bp['medians'][i]
    medianX = []
    medianY = []
    for j in range(2):
        medianX.append(med.get_xdata()[j])
        medianY.append(med.get_ydata()[j])
    ax1.plot(medianX, medianY, 'k', lw=1)
medianY.append(med.get_ydata()[j])
plt.plot(medianX, medianY, 'k')
medians[i] = medianY[0]

# Finally, overplot the sample averages, with horizontal alignment
# in the center of each box
plt.plot([np.average(med.get_xdata())], [np.average(data[i])],
        color='w', marker='*', markeredgcolor='k')

# Set the axes ranges and axes labels
ax1.set_xlim(0.5, numBoxes + 0.5)
top = 40
bottom = -5
ax1.set_ylim(bottom, top)
xtickNames = plt.setp(ax1, xticklabels=np.repeat(randomDists, 2))
plt.setp(xtickNames, rotation=45, fontsize=8)

# Due to the Y-axis scale being different across samples, it can be
# hard to compare differences in medians across the samples. Add upper
# X-axis tick labels with the sample medians to aid in comparison
# (just use two decimal places of precision)
pos = np.arange(numBoxes) + 1
upperLabels = [str(np.round(s, 2)) for s in medians]
weights = ['bold', 'semibold']
for tick, label in zip(range(numBoxes), ax1.get_xticklabels()):
    k = tick % 2
    ax1.text(pos[tick], top - (top*0.05), upperLabels[tick],
             horizontalalignment='center', size='x-small', weight=weights[k],
             color=boxColors[k])

# Finally, add a basic legend
plt.figtext(0.80, 0.08, str(N) + ' Random Numbers',
            backgroundcolor=boxColors[0], color='black', weight='roman',
            size='x-small')
plt.figtext(0.80, 0.045, 'IID Bootstrap Resample',
            backgroundcolor=boxColors[1],
            color='white', weight='roman', size='x-small')
plt.figtext(0.80, 0.015, '*', color='white', backgroundcolor='silver',
            weight='roman', size='medium')
plt.figtext(0.815, 0.013, 'Average Value', color='black', weight='roman',
            size='x-small')

plt.show()
import matplotlib.pyplot as plt
import numpy as np

def fakeBootStrapper(n):
    ""
    This is just a placeholder for the user's method of bootstrapping the median and its confidence intervals.
    Returns an arbitrary median and confidence intervals packed into a tuple
    ""
    if n == 1:
        med = 0.1
        CI = (-0.25, 0.25)
    else:
        med = 0.2
        CI = (-0.35, 0.50)
    return med, CI
```python
np.random.seed(2)
inc = 0.1
e1 = np.random.normal(0, 1, size=(500,))
e2 = np.random.normal(0, 1, size=(500,))
e3 = np.random.normal(0, 1 + inc, size=(500,))
e4 = np.random.normal(0, 1 + 2*inc, size=(500,))

treatments = [e1, e2, e3, e4]
med1, CI1 = fakeBootStrapper(1)
med2, CI2 = fakeBootStrapper(2)
medians = [None, None, med1, med2]
conf_intervals = [None, None, CI1, CI2]

fig, ax = plt.subplots()
pos = np.array(range(len(treatments))) + 1
bp = ax.boxplot(treatments, sym='k+', positions=pos,
                notch=1, bootstrap=5000,
                usermedians=medians,
                conf_intervals=conf_intervals)

ax.set_xlabel('treatment')
ax.set_ylabel('response')
plt.setp(bp['whiskers'], color='k', linestyle='--')
plt.setp(bp['fliers'], markersize=3.0)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
"""
Broken axis example, where the y-axis will have a portion cut out.
"""

import matplotlib.pyplot as plt
import numpy as np

# 30 points between [0, 0.2) originally made using np.random.rand(30)*.2
pts = np.array([
    0.015, 0.166, 0.133, 0.159, 0.041, 0.024, 0.195, 0.039, 0.161, 0.018,
    0.143, 0.056, 0.125, 0.096, 0.094, 0.051, 0.043, 0.021, 0.138, 0.075,
    0.109, 0.195, 0.050, 0.074, 0.079, 0.155, 0.020, 0.010, 0.061, 0.008])

# Now let's make two outlier points which are far away from everything.
pts[[3, 14]] += .8

f, (ax, ax2) = plt.subplots(2, 1, sharex=True)

# If we were to simply plot pts, we'd lose most of the interesting
details due to the outliers. So let's 'break' or 'cut-out' the y-axis
# into two portions - use the top (ax) for the outliers, and the bottom
# (ax2) for the details of the majority of our data
f, (ax, ax2) = plt.subplots(2, 1, sharex=True)
# plot the same data on both axes
ax.plot(pts)
ax2.plot(pts)

# zoom-in / limit the view to different portions of the data
ax.set_ylim(.78, 1.)  # outliers only
ax2.set_ylim(0, .22)  # most of the data

# hide the spines between ax and ax2
ax.spines['bottom'].set_visible(False)
ax2.spines['top'].set_visible(False)
ax.xaxis.tick_top()
ax.tick_params(labeltop='off')  # don't put tick labels at the top
ax2.xaxis.tick_bottom()

# This looks pretty good, and was fairly painless, but you can get that
# cut-out diagonal lines look with just a bit more work. The important
# thing to know here is that in axes coordinates, which are always
# between 0-1, spine endpoints are at these locations (0,0), (0,1),
# (1,0), and (1,1). Thus, we just need to put the diagonals in the
# appropriate corners of each of our axes, and so long as we use the
# right transform and disable clipping.

d = .015  # how big to make the diagonal lines in axes coordinates
# arguments to pass to plot, just so we don't keep repeating them
kwargs = dict(transform=ax.transAxes, color='k', clip_on=False)
ax.plot((-d, +d), (-d, +d), **kwargs)  # top-left diagonal
ax.plot((1 - d, 1 + d), (-d, +d), **kwargs)  # top-right diagonal
kwargs.update(transform=ax2.transAxes)  # switch to the bottom axes
ax2.plot((-d, +d), (1 - d, 1 + d), **kwargs)  # bottom-left diagonal
ax2.plot((1 - d, 1 + d), (1 - d, 1 + d), **kwargs)  # bottom-right diagonal

# What's cool about this is that now if we vary the distance between
# ax and ax2 via fig.subplots_adjust(hspace=...) or plt.subplot_tool(),
# the diagonal lines will move accordingly, and stay right at the tips
# of the spines they are 'breaking'

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Make a "broken" horizontal bar plot, i.e., one with gaps

```python
import matplotlib.pyplot as plt
fig, ax = plt.subplots()
ax.broken_barh([(110, 30), (150, 10)], (10, 9), facecolors='blue')
ax.broken_barh([(10, 50), (100, 20), (130, 10)], (20, 9),
               facecolors=('red', 'yellow', 'green'))
ax.set_ylim(5, 35)
ax.set_xlim(0, 200)
ax.set_xlabel('seconds since start')
ax.set_yticks([15, 25])
ax.set_yticklabels(['Bill', 'Jim'])
ax.grid(True)
ax.annotate('race interrupted', (61, 25),
            xytext=(0.8, 0.9), textcoords='axes fraction',
            arrowprops=dict(facecolor='black', shrink=0.05),
            fontsize=16,
            horizontalalignment='right', verticalalignment='top')
plt.show()
```
99.30 pylab_examples example code: centered_ticklabels.py

```python
# sometimes it is nice to have ticklabels centered. mpl currently
# associates a label with a tick, and the label can be aligned
# 'center', 'left', or 'right' using the horizontal alignment property:
#
# for label in ax.xaxis.get_xticklabels():
#    label.set_horizontalalignment('right')
#
# but this doesn't help center the label between ticks. One solution
# is to "face it". Use the minor ticks to place a tick centered
# between the major ticks. Here is an example that labels the months,
# centered between the ticks

import numpy as np
import matplotlib.cbook as cbook
import matplotlib.dates as dates
import matplotlib.ticker as ticker
import matplotlib.pyplot as plt
```
# load some financial data; apple's stock price

```python
fh = cbook.get_sample_data('aapl.npy.gz')

try:
    # Python3 cannot load python2 .npy files with datetime(object) arrays
    # unless the encoding is set to bytes. However this option was
    # not added until numpy 1.10 so this example will only work with
    # python 2 or with numpy 1.10 and later.
    r = np.load(fh, encoding='bytes')
except TypeError:
    r = np.load(fh)
fh.close()

r = r[-250:]  # get the last 250 days

fig, ax = plt.subplots()
ax.plot(r.date, r.adj_close)
ax.xaxis.set_major_locator(dates.MonthLocator())
ax.xaxis.set_minor_locator(dates.MonthLocator(bymonthday=15))
ax.xaxis.set_major_formatter(ticker.NullFormatter())
ax.xaxis.set_minor_formatter(dates.DateFormatter('%b'))

for tick in ax.xaxis.getMinor_ticks():
    tick.tick1line.set_markersize(0)
    tick.tick2line.set_markersize(0)
    tick.label1.set_horizontalalignment('center')

imid = len(r)//2
ax.set_xlabel(str(r.date[imid].year))
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Compute the coherence of two signals

```python
import numpy as np
import matplotlib.pyplot as plt

# make a little extra space between the subplots
plt.subplots_adjust(wspace=0.5)

dt = 0.01
t = np.arange(0, 30, dt)
nse1 = np.random.randn(len(t))  # white noise 1
nse2 = np.random.randn(len(t))  # white noise 2
r = np.exp(-t/0.05)

cnse1 = np.convolve(nse1, r, mode='same')*dt  # colored noise 1
ecnse2 = np.convolve(nse2, r, mode='same')*dt  # colored noise 2

# two signals with a coherent part and a random part
s1 = 0.01*np.sin(2*np.pi*10*t) + cnse1
s2 = 0.01*np.sin(2*np.pi*10*t) + cnse2
```
plt.subplot(211)
plt.plot(t, s1, t, s2)
plt.xlim(0, 5)
plt.xlabel('time')
plt.ylabel('s1 and s2')
plt.grid(True)

plt.subplot(212)
cxy, f = plt.cohere(s1, s2, 256, 1./dt)
plt.ylabel('coherence')
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

99.32 pylab_examples example code: color_by_yvalue.py

```python
import numpy as np
import matplotlib.pyplot as plt

# use masked arrays to plot a line with different colors by y-value

t = np.arange(0.0, 2.0, 0.01)
s = np.sin(2*np.pi*t)
plt.plot(t, s, color='blue')
plt.plot(t, 0.5*s, color='green')
plt.plot(t, 0.25*s, color='orange')
plt.show()
```
upper = 0.77
lower = -0.77

supper = np.ma.masked_where(s < upper, s)
slower = np.ma.masked_where(s > lower, s)
smiddle = np.ma.masked_where(np.logical_or(s < lower, s > upper), s)

plt.plot(t, smiddle, t, slower, t, supper)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

99.33 pylab_examples example code: color_demo.py

matplotlib gives you 5 ways to specify colors,

  1) as a single letter string, ala MATLAB

  2) as an html style hex string or html color name
3) as an R,G,B tuple, where R,G,B, range from 0-1

4) as a string representing a floating point number
   from 0 to 1, corresponding to shades of gray.

5) as a special color "Cn", where n is a number 0-9 specifying the
   nth color in the currently active color cycle.

See help(colors) for more info.

""

import matplotlib.pyplot as plt
import numpy as np

plt.subplot(111, facecolor='darkslategray')
# subplot(111, facecolor='ababab')
t = np.arange(0.0, 2.0, 0.01)
s = np.sin(2*np.pi*t)
plt.plot(t, s, 'C1')
plt.xlabel('time (s)', color='C1')
plt.ylabel('voltage (mV)', color='0.5') # grayscale color
plt.title('About as silly as it gets, folks', color='#afeeee')
plt.show()
Gaussian noise with vertical colorbar

-1
0
1

0 50 100 150 200 250
0
50
100
150
200
250

< -1
0
> 1
Produce custom labelling for a colorbar.

Contributed by Scott Sinclair

```python
import matplotlib.pyplot as plt
import numpy as np
from matplotlib import cm
from numpy.random import randn

# Make plot with vertical (default) colorbar
fig, ax = plt.subplots()
data = np.clip(randn(250, 250), -1, 1)
cax = ax.imshow(data, interpolation='nearest', cmap=cm.coolwarm)
ax.set_title('Gaussian noise with vertical colorbar')

cbar = fig.colorbar(cax, ticks=[-1, 0, 1])
cbar.ax.set_yticklabels(['< -1', '0', '> 1'])  # vertically oriented colorbar

# Make plot with horizontal colorbar
fig, ax = plt.subplots()
data = np.clip(randn(250, 250), -1, 1)
```
cax = ax.imshow(data, interpolation='nearest', cmap=cm.afmhot)
ax.set_title('Gaussian noise with horizontal colorbar')

cbar = fig.colorbar(cax, ticks=[-1, 0, 1], orientation='horizontal')
cbar.ax.set_xticklabels(['Low', 'Medium', 'High'])  # horizontal colorbar

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

99.35 *pylab_examples* example code: *colours.py*

[source code]

```python
# -*- noplot -*-

""
Some simple functions to generate colours.
""
import numpy as np
from matplotlib import colors as mcolors

def pastel(colour, weight=2.4):
    ""
    Convert colour into a nice pastel shade"
    rgb = np.asarray(mcolors.to_rgba(colour)[:3])
    # scale colour
    maxc = max(rgb)
    if maxc < 1.0 and maxc > 0:
        # scale colour
        scale = 1.0 / maxc
        rgb = rgb * scale
    # now decrease saturation
    total = rgb.sum()
    slack = 0
    for x in rgb:
        slack += 1.0 - x

    # want to increase weight from total to weight
    # pick x s.t. slack * x == weight - total
    # x = (weight - total) / slack
    x = (weight - total) / slack
    rgb = [c + (x * (1.0 - c)) for c in rgb]

    return rgb

def get_colours(n):
    ""
    Return n pastel colours."
    base = np.asarray([[1, 0, 0], [0, 1, 0], [0, 0, 1]])
```
if n <= 3:
    return base[0:n]

# how many new colours to we need to insert between
# red and green and between green and blue?
needed = (((n - 3) + 1) / 2, (n - 3) / 2)

colours = []
for start in (0, 1):
    for x in np.linspace(0, 1, needed[start] + 2):
        colours.append((base[start] * (1.0 - x)) +
                        (base[start + 1] * x))

return [pastel(c) for c in colours[0:n]]

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

99.36  pylab_examples example code: contour_corner_mask.py

Illustrate the difference between corner_mask=False and corner_mask=True
for masked contour plots.

```
import matplotlib.pyplot as plt
import numpy as np

# Data to plot.
x, y = np.meshgrid(np.arange(7), np.arange(10))
z = np.sin(0.5*x)*np.cos(0.52*y)

# Mask various z values.
mask = np.zeros_like(z, dtype=np.bool)
mask[2, 3:5] = True
mask[3:5, 4] = True
mask[7, 2] = True
mask[5, 0] = True
mask[0, 6] = True
z = np.ma.array(z, mask=mask)

corner_masks = [False, True]
for i, corner_mask in enumerate(corner_masks):
    plt.subplot(1, 2, i+1)
    cs = plt.contourf(x, y, z, corner_mask=corner_mask)
    plt.contour(cs, colors='k')
    plt.title('corner_mask = {0}'.format(corner_mask))

    # Plot grid.
    plt.grid(c='k', ls='-', alpha=0.3)

    # Indicate masked points with red circles.
    plt.plot(np.ma.array(x, mask=~mask), y, 'ro')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Simplest default with labels
Single color - negative contours dashed
Single color - negative contours solid
Illustrate simple contour plotting, contours on an image with a colorbar for the contours, and labelled contours.

See also contour_image.py.

```python
import matplotlib
import numpy as np
import matplotlib.cm as cm
import matplotlib.mlab as mlab
import matplotlib.pyplot as plt

matplotlib.rcParams['xtick.direction'] = 'out'
matplotlib.rcParams['ytick.direction'] = 'out'

delta = 0.025
x = np.arange(-3.0, 3.0, delta)
y = np.arange(-2.0, 2.0, delta)
X, Y = np.meshgrid(x, y)
Z1 = mlab.bivariate_normal(X, Y, 1.0, 1.0, 0.0, 0.0)
Z2 = mlab.bivariate_normal(X, Y, 1.5, 0.5, 1.0, 1.0)
# difference of Gaussians
Z = 10.0 * (Z2 - Z1)

# Create a simple contour plot with labels using default colors. The
```
Matplotlib, Release 2.0.2

```python
# inline argument to clabel will control whether the labels are draw
# over the line segments of the contour, removing the lines beneath
# the label
plt.figure()
CS = plt.contour(X, Y, Z)
plt.clabel(CS, inline=1, fontsize=10)
plt.title('Simplest default with labels')

# contour labels can be placed manually by providing list of positions
# (in data coordinate). See ginput_manual_clabel.py for interactive
# placement.
plt.figure()
CS = plt.contour(X, Y, Z)
manual_locations = [(-1, -1.4), (-0.62, -0.7), (-2, 0.5), (1.7, 1.2), (2.0, 1.4), (2.4, -1.7)]
plt.clabel(CS, inline=1, fontsize=10, manual=manual_locations)
plt.title('labels at selected locations')

# You can force all the contours to be the same color.
plt.figure()
CS = plt.contour(X, Y, Z, 6,
    colors='k',  # negative contours will be dashed by default
)
plt.clabel(CS, fontsize=9, inline=1)
plt.title('Single color - negative contours dashed')

# You can set negative contours to be solid instead of dashed:
matplotlib.rcParams['contour.negative_linestyle'] = 'solid'
plt.figure()
CS = plt.contour(X, Y, Z, 6,
    colors='k',  # negative contours will be dashed by default
)
plt.clabel(CS, fontsize=9, inline=1)
plt.title('Single color - negative contours solid')

# And you can manually specify the colors of the contour
plt.figure()
CS = plt.contour(X, Y, Z, 6,
    linewidths=np.arange(.5, 4, .5),
    colors=('
        r', 'green', 'blue', (1, 1, 0), '#afeeee', '
    ')
)
plt.clabel(CS, fontsize=9, inline=1)
plt.title('Crazy lines')

# Or you can use a colormap to specify the colors; the default
# colormap will be used for the contour lines
plt.figure()
im = plt.imshow(Z, interpolation='bilinear', origin='lower',
    cmap=cm.gray, extent=(-3, 3, -2, 2))
```

99.37. pylab_examples example code: contour_demo.py 2417
levels = np.arange(-1.2, 1.6, 0.2)
CS = plt.contour(Z, levels,
                origin='lower',
                linewidths=2,
                extent=(-3, 3, -2, 2))

# Thicken the zero contour.
zc = CS.collections[6]
plt.setp(zc, linewidth=4)

plt.clabel(CS, levels[1::2],
           inline=1,
           fmt='%1.1f',
           fontsize=14)

# make a colorbar for the contour lines
CB = plt.colorbar(CS, shrink=0.8, extend='both')

plt.title('Lines with colorbar')

plt.show()

# We can still add a colorbar for the image, too.
CBI = plt.colorbar(im, orientation='horizontal', shrink=0.8)

# This makes the original colorbar look a bit out of place,
# so let's improve its position.

l, b, w, h = plt.gca().get_position().bounds
ll, bb, ww, hh = CB.ax.get_position().bounds
CB.ax.set_position([ll, b + 0.1*h, ww, h*0.8])

plt.show()
Test combinations of contouring, filled contouring, and image plotting. For contour labelling, see contour_demo.py.

The emphasis in this demo is on showing how to make contours register correctly on images, and on how to get both of them oriented as desired. In particular, note the usage of the "origin" and "extent" keyword arguments to imshow and contour.

```python
import matplotlib.pyplot as plt
import numpy as np
from matplotlib import mlab, cm

# Default delta is large because that makes it fast, and it illustrates
# the correct registration between image and contours.
delta = 0.5
extent = (-3, 4, -4, 3)
x = np.arange(-3.0, 4.001, delta)
y = np.arange(-4.0, 3.001, delta)
X, Y = np.meshgrid(x, y)
```

99.38. `pylab_examples` example code: `contour_image.py`
```
Z1 = mlab.bivariate_normal(X, Y, 1.0, 1.0, 0.0, 0.0)
Z2 = mlab.bivariate_normal(X, Y, 1.5, 0.5, 1, 1)
Z = (Z1 - Z2) * 10

levels = np.arange(-2.0, 1.601, 0.4)  # Boost the upper limit to avoid truncation errors.

norm = cm.colors.Normalize(vmax=abs(Z).max(), vmin=-abs(Z).max())
cmap = cm.PRGn

fig = plt.figure()
fig.subplots_adjust(hspace=0.3)
plt.subplot(2, 2, 1)
cset1 = plt.contourf(X, Y, Z, levels,
                     cmap=cm.get_cmap(cmap, len(levels) - 1),
                     norm=norm,
                     )
# It is not necessary, but for the colormap, we need only the
# number of levels minus 1. To avoid discretization error, use
# either this number or a large number such as the default (256).

# If we want lines as well as filled regions, we need to call
# contour separately; don't try to change the edgecolor or edgewidth
# of the polygons in the collections returned by contourf.
# Use levels output from previous call to guarantee they are the same.

if cmap is None:
    # Get max and min from the first call to contourf.
    limits = cset1.levels
cmap = cm.PRGn

norm = cm.colors.Normalize(vmax=abs(Z).max(), vmin=-abs(Z).max())
cmap = cm.PRGn

fig = plt.figure()
fig.subplots_adjust(hspace=0.3)
plt.subplot(2, 2, 1)
cset1 = plt.contourf(X, Y, Z, levels,
                     cmap=cm.get_cmap(cmap, len(levels) - 1),
                     norm=norm,
                     )
# It is not necessary, but for the colormap, we need only the
# number of levels minus 1. To avoid discretization error, use
# either this number or a large number such as the default (256).

# If we want lines as well as filled regions, we need to call
# contour separately; don't try to change the edgecolor or edgewidth
# of the polygons in the collections returned by contourf.
# Use levels output from previous call to guarantee they are the same.

cset2 = plt.contour(X, Y, Z, cset1.levels, colors='k')

# We don't really need dashed contour lines to indicate negative
# regions, so let's turn them off.
for c in cset2.collections:
    c.set_linestyle('solid')

# It is easier here to make a separate call to contour than
# to set up an array of colors and linewidths.
# We are making a thick green line as a zero contour.
# Specify the zero level as a tuple with only 0 in it.

cset3 = plt.contour(X, Y, Z, (0,), colors='g', linewidths=2)
plt.title('Filled contours')
plt.colorbar(cset1)

plt.subplot(2, 2, 2)
plt.imshow(Z, extent=extent, cmap=cmap, norm=norm)
v = plt.axis()
plt.contour(Z, levels, colors='k', origin='upper', extent=extent)
plt.axis(v)
plt.title("Image, origin 'upper'")
```

plt.subplot(2, 2, 3)
plt.imshow(Z, origin='lower', extent=extent, cmap=cmap, norm=norm)
v = plt.axis()
plt.contour(Z, levels, colors='k', origin='lower', extent=extent)
plt.axis(v)
plt.title("Image, origin 'lower'")

plt.subplot(2, 2, 4)

# We will use the interpolation "nearest" here to show the actual
# image pixels.
# Note that the contour lines don't extend to the edge of the box.
# This is intentional. The Z values are defined at the center of each
# image pixel (each color block on the following subplot), so the
# domain that is contoured does not extend beyond these pixel centers.
im = plt.imshow(Z, interpolation='nearest', extent=extent, cmap=cmap, norm=norm)
v = plt.axis()
plt.contour(Z, levels, colors='k', origin='image', extent=extent)
plt.axis(v)
ylim = plt.get(plt.gca(), 'ylim')
plt.setp(plt.gca(), ylim=ylim[::-1])
plt.title("Origin from rc, reversed y-axis")
plt.colorbar(im)

plt.show()
99.39 *pylab_examples* example code: *contour_label_demo.py*
Illustrate some of the more advanced things that one can do with contour labels.

See also contour_demo.py.

```python
import matplotlib
import numpy as np
import matplotlib.cm as cm
import matplotlib.mlab as mlab
import matplotlib.ticker as ticker
import matplotlib.pyplot as plt

matplotlib.rcParams['xtick.direction'] = 'out'
matplotlib.rcParams['ytick.direction'] = 'out'

# Define our surface
delta = 0.025
x = np.arange(-3.0, 3.0, delta)
y = np.arange(-2.0, 2.0, delta)
X, Y = np.meshgrid(x, y)
Z1 = mlab.bivariate_normal(X, Y, 1.0, 1.0, 0.0, 0.0)
Z2 = mlab.bivariate_normal(X, Y, 1.5, 0.5, 1, 1)
# difference of Gaussians
```
Z = 10.0 * (Z2 - Z1)

# Make contour labels using creative float classes
# Follows suggestion of Manuel Metz

plt.figure()

# Basic contour plot
CS = plt.contour(X, Y, Z)

# Define a class that forces representation of float to look a certain way
# This remove trailing zero so '1.0' becomes '1'

class nf(float):
    def __repr__(self):
        str = '%.1f' % (self.__float__(),)
        if str[-1] == '0':
            return '%.0f' % self.__float__()
        else:
            return '%.1f' % self.__float__()

# Recast levels to new class
CS.levels = [nf(val) for val in CS.levels]

# Label levels with specially formatted floats
if plt.rcParams['text.usetex']:
    fmt = r'%.2f
else:
    fmt = '%.2f'
plt.clabel(CS, CS.levels, inline=True, fmt=fmt, fontsize=10)

# Label contours with arbitrary strings using a dictionary

plt.figure()

# Basic contour plot
CS = plt.contour(X, Y, Z)

fmt = {}
strs = ['first', 'second', 'third', 'fourth', 'fifth', 'sixth', 'seventh']
for l, s in zip(CS.levels, strs):
    fmt[l] = s

# Label every other level using strings
plt.clabel(CS, CS.levels[::2], inline=True, fmt=fmt, fontsize=10)

# Use a Formatter

plt.figure()
Matplotlib, Release 2.0.2

```python
CS = plt.contour(X, Y, 100**Z, locator=plt.LogLocator())
fmt = ticker.LogFormatterMathText()
fmt.create_dummy_axis()
plt.clabel(CS, CS.levels, fmt=fmt)
plt.title("$100^Z$")
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

99.40 `pylab_examples` example code: `contourf_demo.py`

![Contour plot example](image_url)
Listed colors (3 masked regions)
import numpy as np
import matplotlib.pyplot as plt

origin = 'lower'
#origin = 'upper'

delta = 0.025

x = y = np.arange(-3.0, 3.01, delta)
X, Y = np.meshgrid(x, y)
Z1 = plt.mlab.bivariate_normal(X, Y, 1.0, 1.0, 0.0, 0.0)
Z2 = plt.mlab.bivariate_normal(X, Y, 1.5, 0.5, 1, 1)
Z = 10 * (Z1 - Z2)

nr, nc = Z.shape

# put Nans in one corner:
Z[-nr//6:, -nc//6:] = np.nan
# contourf will convert these to masked

Z = np.ma.array(Z)
# mask another corner:
Z[:nr//6, :nc//6] = np.ma.masked

# mask a circle in the middle:
interior = np.sqrt((X**2) + (Y**2)) < 0.5
Z[interior] = np.ma.masked

# We are using automatic selection of contour levels;
# this is usually not such a good idea, because they don't
# occur on nice boundaries, but we do it here for purposes
# of illustration.
CS = plt.contourf(X, Y, Z, 10,
                 #[-1, -0.1, 0, 0.1],
                 #alpha=0.5,
                 cmap=plt.cm.bone,
                 origin=origin)

# Note that in the following, we explicitly pass in a subset of
# the contour levels used for the filled contours. Alternatively,
# We could pass in additional levels to provide extra resolution,
# or leave out the levels kwarg to use all of the original levels.
CS2 = plt.contour(CS, levels=CS.levels[::2],
                colors='r',
                origin=origin)

plt.title('Nonsense (3 masked regions)')
plt.xlabel('word length anomaly')
plt.ylabel('sentence length anomaly')

# Make a colorbar for the ContourSet returned by the contourf call.
cbar = plt.colorbar(CS)
cbar.ax.set_ylabel('verbosity coefficient')
# Add the contour line levels to the colorbar
   cbar.add_lines(CS2)

plt.figure()

# Now make a contour plot with the levels specified,
# and with the colormap generated automatically from a list
# of colors.
levels = [-1.5, -1, -0.5, 0, 0.5, 1]
CS3 = plt.contourf(X, Y, Z, levels,
                  colors=('r', 'g', 'b'),
                  origin=origin,
                  extend='both')

# Our data range extends outside the range of levels; make
# data below the lowest contour level yellow, and above the
# highest level cyan:
CS3.cmap.set_under('yellow')
CS3.cmap.set_over('cyan')

CS4 = plt.contour(X, Y, Z, levels,
                  colors=('k',),
                  linewidens=(3,),
                  origin=origin)

plt.title('Listed colors (3 masked regions)')
plt.clabel(CS4, fmt='%.2f', colors='w', fontsize=14)

# Notice that the colorbar command gets all the information it
# needs from the ContourSet object, CS3.
plt.colorbar(CS3)

# Illustrate all 4 possible "extend" settings:
extends = ['neither', 'both', 'min', 'max']
cmap = plt.cm.get_cmap("winter")
cmap.set_under("magenta")
cmap.set_over("yellow")
# Note: contouring simply excludes masked or nan regions, so
# instead of using the "bad" colormap value for them, it draws
# nothing at all in them. Therefore the following would have
# no effect:
# cmap.set_bad("red")

fig, axs = plt.subplots(2, 2)
fig.subplots_adjust(hspace=0.3)

for ax, extend in zip(axs.ravel(), extends):
    cs = ax.contourf(X, Y, Z, levels, cmap=cmap, extend=extend, origin=origin)
    fig.colorbar(cs, ax=ax, shrink=0.9)
    ax.set_title("extend = %s" % extend)
    ax.locator_params(nbins=4)

plt.show()
99.41  *pylab_examples* example code: *contourf_hatching.py*
import matplotlib.pyplot as plt
import numpy as np

# invent some numbers, turning the x and y arrays into simple
# 2d arrays, which make combining them together easier.
x = np.linspace(-3, 5, 150).reshape(1, -1)
y = np.linspace(-3, 5, 120).reshape(-1, 1)
z = np.cos(x) + np.sin(y)

# we no longer need x and y to be 2 dimensional, so flatten them.
x, y = x.flatten(), y.flatten()

# ---------------------------------------------  Plot #1  ---------------------------------------------
fig = plt.figure()

cs = plt.contourf(x, y, z, hatches=['-', '/', '\\', '//'],
                  cmap=plt.get_cmap('gray'),
                  extend='both', alpha=0.5)
plt.colorbar()
# a plot of hatches without color with a legend
plt.figure()
n_levels = 6
plt.contour(x, y, z, n_levels, colors='black', linestyles='--')
cs = plt.contourf(x, y, z, n_levels, colors='none',
                hatches=['.', '/', '\', None, '\\', '*'],
                extend='lower')

# create a legend for the contour set
artists, labels = cs.legend_elements()
plt.legend(artists, labels, handleheight=2)

plt.show()
An example of how to interact with the plotting canvas by connecting to move and click events

```python
from __future__ import print_function
import sys
import matplotlib.pyplot as plt
import numpy as np

t = np.arange(0.0, 1.0, 0.01)
s = np.sin(2*np.pi*t)
fig, ax = plt.subplots()
ax.plot(t, s)

def on_move(event):
    x, y = event.x, event.y
    if event.inaxes:
        ax = event.inaxes  # the axes instance
        print('data coords %f %f' % (event.xdata, event.ydata))

def on_click(event):
    x, y = event.x, event.y
    if event.button == 1:
        if event.inaxes is not None:
            print('data coords %f %f' % (event.xdata, event.ydata))

binding_id = plt.connect('motion_notify_event', on_move)
plt.connect('button_press_event', on_click)

if "test_disconnect" in sys.argv:
    print("disconnecting console coordinate printout...")
    plt.disconnect(binding_id)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
# override the default reporting of coords

```python
import matplotlib.pyplot as plt
import numpy as np

def millions(x):
    return r'$%1.1fM$' % (x*1e-6)

x = np.random.rand(20)
y = 1e7*np.random.rand(20)

fig, ax = plt.subplots()
ax.fmt_ydata = millions
plt.plot(x, y, 'o')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Compute the cross spectral density of two signals

```python
import numpy as np
import matplotlib.pyplot as plt

fig, (ax1, ax2) = plt.subplots(2, 1)
# make a little extra space between the subplots
fig.subplots_adjust(hspace=0.5)

dt = 0.01
T = np.arange(0, 30, dt)
nse1 = np.random.randn(len(t)) # white noise 1
nse2 = np.random.randn(len(t)) # white noise 2
r = np.exp(-t/0.05)

# colored noise
nc1 = np.convolve(nse1, r, mode='same')*dt # colored noise 1
nc2 = np.convolve(nse2, r, mode='same')*dt # colored noise 2
	s1 = 0.01*np.sin(2*np.pi*10*t) + nc1
```

```python
# two signals with a coherent part and a random part
```

```python
```
s2 = 0.01*np.sin(2*np.pi*10*t) + cnse2

ax1.plot(t, s1, t, s2)
ax1.set_xlim(0, 5)
ax1.set_xlabel('time')
ax1.set_ylabel('s1 and s2')
ax1.grid(True)

cxy, f = ax2.csd(s1, s2, 256, 1./dt)
ax2.set_ylabel('CSD (db)')
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

99.45 pylab_examples example code: cursor_demo.py

[source code]

```python
# -*- noplot -*-

""
This example shows how to use matplotlib to provide a data cursor. It
uses matplotlib to draw the cursor and may be a slow since this
requires redrawing the figure with every mouse move.

Faster cursoring is possible using native GUI drawing, as in
wxcursor_demo.py.

The mpldatacursor and mplcursors third-party packages can be used to achieve a
similar effect. See
   https://github.com/joferkington/mpldatacursor
   https://github.com/anntzer/mplcursors
""

from __future__ import print_function
import matplotlib.pyplot as plt
import numpy as np

class Cursor(object):
    def __init__(self, ax):
        self.ax = ax
        self.lx = ax.axhline(color='k')  # the horiz line
        self.ly = ax.axvline(color='k')  # the vert line

        # text location in axes coords
        self.txt = ax.text(0.7, 0.9, '', transform=ax.transAxes)

    def mouse_move(self, event):
        if not event.inaxes:
            return
```

99.45. pylab_examples example code: cursor_demo.py 2437
x, y = event.xdata, event.ydata
# update the line positions
self.lx.set_ydata(y)
self.ly.set_xdata(x)

self.txt.set_text('x=%1.2f, y=%1.2f' % (x, y))
plt.draw()

class SnaptoCursor(object):
    
    Like Cursor but the crosshair snaps to the nearest x,y point
    For simplicity, I'm assuming x is sorted

    def __init__(self, ax, x, y):
        self.ax = ax
        self.lx = ax.axhline(color='k')  # the horiz line
        self.ly = ax.axvline(color='k')  # the vert line
        self.x = x
        self.y = y
        # text location in axes coords
        self.txt = ax.text(0.7, 0.9, '', transform=ax.transAxes)

    def mouse_move(self, event):
        if not event.inaxes:
            return
        x, y = event.xdata, event.ydata

        indx = np.searchsorted(self.x, [x])[0]
x = self.x[indx]
y = self.y[indx]
# update the line positions
self.lx.set_ydata(y)
self.ly.set_xdata(x)

self.txt.set_text('x=%1.2f, y=%1.2f' % (x, y))
print('x=%1.2f, y=%1.2f' % (x, y))
plt.draw()

t = np.arange(0.0, 1.0, 0.01)
s = np.sin(2*2*np.pi*t)
fig, ax = plt.subplots()

#cursor = Cursor(ax)
cursor = SnaptoCursor(ax, t, s)
plt.connect('motion_notify_event', cursor.mouse_move)

ax.plot(t, s, 'o')
plt.axis([0, 1, -1, 1])
plt.show()
99.46  pylab_examples example code: custom_cmap.py

N bins: 3

N bins: 6

N bins: 10

N bins: 100
Custom Blue-Red colormaps

Varying alpha
Creating a colormap from a list of colors
-----------------------------------------
Creating a colormap from a list of colors can be done with the `from_list` method of `LinearSegmentedColormap`. You must pass a list of RGB tuples that define the mixture of colors from 0 to 1.

Creating custom colormaps
-------------------------
It is also possible to create a custom mapping for a colormap. This is accomplished by creating dictionary that specifies how the RGB channels change from one end of the cmap to the other.

Example: suppose you want red to increase from 0 to 1 over the bottom half, green to do the same over the middle half, and blue over the top half. Then you would use:

cdict = {'red': ((0.0, 0.0, 0.0),
                 (0.5, 1.0, 1.0),
                 (1.0, 1.0, 1.0)),

           'green': ((0.0, 0.0, 0.0),
                    (0.25, 0.0, 0.0),
                    (0.75, 1.0, 1.0),
                    (1.0, 1.0, 1.0)),

           'blue': ((0.0, 0.0, 0.0),
                    (0.5, 0.0, 0.0),
                    (1.0, 1.0, 1.0))}

If, as in this example, there are no discontinuities in the r, g, and b components, then it is quite simple: the second and third element of each tuple, above, is the same--call it "y". The first element ("x") defines interpolation intervals over the full range of 0 to 1, and it must span that whole range. In other words, the values of x divide the 0-to-1 range into a set of segments, and y gives the end-point color values for each segment.

Now consider the green. cdict['green'] is saying that for
0 <= x <= 0.25, y is zero; no green.
0.25 < x <= 0.75, y varies linearly from 0 to 1.
x > 0.75, y remains at 1, full green.

If there are discontinuities, then it is a little more complicated. Label the 3 elements in each row in the cdict entry for a given color as (x, y0, y1). Then for values of x between x[i] and x[i+1] the color value is interpolated between y1[i] and y0[i+1].
Going back to the cookbook example, look at `cdict['red']`; because `y0` \(!= y1`, it is saying that for `x` from `0` to `0.5`, red increases from `0` to `1`, but then it jumps down, so that for `x` from `0.5` to `1`, red increases from `0.7` to `1`. Green ramps from `0` to `1` as `x` goes from `0` to `0.5`, then jumps back to `0`, and ramps back to `1` as `x` goes from `0.5` to `1`.

\[
\begin{array}{ccc}
\text{row } i: & x & y0 & y1 \\
\text{row } i+1: & x & y0 & y1 \\
\end{array}
\]

Above is an attempt to show that for `x` in the range `x[i]` to `x[i+1]`, the interpolation is between `y1[i]` and `y0[i+1]`. So, `y0[0]` and `y1[-1]` are never used.

```python
# Make some illustrative fake data:

x = np.arange(0, np.pi, 0.1)
y = np.arange(0, 2*np.pi, 0.1)
X, Y = np.meshgrid(x, y)
Z = np.cos(X) * np.sin(Y) * 10

# --- Colormaps from a list ---

colors = [(1, 0, 0), (0, 1, 0), (0, 0, 1)]  # R -> G -> B
n_bins = [3, 6, 10, 100]  # Discretizes the interpolation into bins
cmap_name = 'my_list'
fig, axs = plt.subplots(2, 2, figsize=(6, 9))
fig.subplots_adjust(left=0.02, bottom=0.06, right=0.95, top=0.94, wspace=0.05)
for n_bin, ax in zip(n_bins, axs.ravel()):
    # Create the colormap
    cm = LinearSegmentedColormap.from_list(
        cmap_name, colors, N=n_bin)
    # Fewer bins will result in "coarser" colormap interpolation
    im = ax.imshow(Z, interpolation='nearest', origin='lower', cmap=cm)
    ax.set_title("N bins: %s" % n_bin)
    fig.colorbar(im, ax=ax)

# --- Custom colormaps ---

cdict1 = {'red': ((0.0, 0.0, 0.0),
                   (0.5, 0.0, 0.1),
                   (1.0, 1.0, 1.0)),
           'green': ((0.0, 0.0, 0.0),
                     (1.0, 0.0, 0.0)),
           'blue': ((0.0, 0.0, 1.0),
                    (0.5, 0.1, 0.0),
                    (1.0, 0.0, 0.0))}
```


cdict2 = {'red': ((0.0, 0.0, 0.0),
                  (0.5, 0.0, 1.0),
                  (1.0, 0.1, 1.0)),

           'green': ((0.0, 0.0, 0.0),
                     (1.0, 0.0, 0.0)),

           'blue': ((0.0, 0.0, 0.1),
                    (0.5, 1.0, 0.0),
                    (1.0, 0.0, 0.0))
        }

cdict3 = {'red': ((0.0, 0.0, 0.0),
                  (0.25, 0.0, 0.0),
                  (0.5, 0.8, 1.0),
                  (0.75, 1.0, 1.0),
                  (1.0, 0.4, 1.0)),

           'green': ((0.0, 0.0, 0.0),
                     (0.25, 0.0, 0.0),
                     (0.5, 0.9, 0.9),
                     (0.75, 0.0, 0.0),
                     (1.0, 0.0, 0.0)),

           'blue': ((0.0, 0.0, 0.4),
                    (0.25, 1.0, 1.0),
                    (0.5, 1.0, 0.8),
                    (0.75, 0.0, 0.0),
                    (1.0, 0.0, 0.0))
        }

# Make a modified version of cdict3 with some transparency
# in the middle of the range.
cdict4 = cdict3.copy()
cdict4['alpha'] = ((0.0, 1.0, 1.0),
                    # (0.25,1.0, 1.0),
                    # (0.5, 0.3, 0.3),
                    # (0.75,1.0, 1.0),
                    (1.0, 1.0, 1.0))

# Now we will use this example to illustrate 3 ways of
# handling custom colormaps.
# First, the most direct and explicit:
blue_red1 = LinearSegmentedColormap('BlueRed1', cdict1)

# Second, create the map explicitly and register it.
# Like the first method, this method works with any kind
# of Colormap, not just
# a LinearSegmentedColormap:
blue_red2 = LinearSegmentedColormap('BlueRed2', cdict2)
plt.register_cmap(cmap=blue_red2)

# Third, for LinearSegmentedColormap only,
# leave everything to register_cmap:
plt.register_cmap(name='BlueRed3', data=cdict3)  # optional lut kwarg
plt.register_cmap(name='BlueRedAlpha', data=cdict4)

# Make the figure:
fig, axs = plt.subplots(2, 2, figsize=(6, 9))
fig.subplots_adjust(left=0.02, bottom=0.06, right=0.95, top=0.94, wspace=0.05)

# Make 4 subplots:
im1 = axs[0, 0].imshow(Z, interpolation='nearest', cmap=blue_red1)
fig.colorbar(im1, ax=axs[0, 0])

cmap = plt.get_cmap('BlueRed2')
im2 = axs[1, 0].imshow(Z, interpolation='nearest', cmap=cmap)
fig.colorbar(im2, ax=axs[1, 0])

# Now we will set the third cmap as the default. One would
# not normally do this in the middle of a script like this;
# it is done here just to illustrate the method.
plt.rcParams['image.cmap'] = 'BlueRed3'
im3 = axs[0, 1].imshow(Z, interpolation='nearest')
fig.colorbar(im3, ax=axs[0, 1])
axs[0, 1].set_title("Alpha = 1")

# Or as yet another variation, we can replace the rcParams
# specification *before* the imshow with the following *after*
# imshow.
# This sets the new default *and* sets the colormap of the last
# image-like item plotted via pyplot, if any.
#
# Draw a line with low zorder so it will be behind the image.
axs[1, 1].plot([0, 10*np.pi], [0, 20*np.pi], color='c', lw=20, zorder=-1)
im4 = axs[1, 1].imshow(Z, interpolation='nearest')
fig.colorbar(im4, ax=axs[1, 1])

# Here it is: changing the colormap for the current image and its
# colorbar after they have been plotted.
im4.set_cmap('BlueRedAlpha')
axs[1, 1].set_title("Varying alpha")
```python
fig.suptitle('Custom Blue-Red colormaps', fontsize=16)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

99.47  `pylab_examples` example code: `custom_figure_class.py`

```
The new ticker code was designed to explicitly support user customized
ticking. The documentation
http://matplotlib.org/matplotlib.ticker.html details this
process. That code defines a lot of preset tickers but was primarily
designed to be user extensible.

In this example a user defined function is used to format the ticks in
millions of dollars on the y axis
```python
from matplotlib.ticker import FuncFormatter
import matplotlib.pyplot as plt
import numpy as np

x = np.arange(4)
money = [1.5e5, 2.5e6, 5.5e6, 2.0e7]

def millions(x, pos):
    'The two args are the value and tick position'
    return '${%.1fM}' % (x*1e-6)

formatter = FuncFormatter(millions)

fig, ax = plt.subplots()
ax.yaxis.set_major_formatter(formatter)
plt.bar(x, money)
plt.xticks(x, ('Bill', 'Fred', 'Mary', 'Sue'))
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
I'm not trying to make a good looking figure here, but just to show some examples of customizing rc params on the fly.

If you like to work interactively, and need to create different sets of defaults for figures (e.g., one set of defaults for publication, one set for interactive exploration), you may want to define some functions in a custom module that set the defaults, e.g.,

def set_pub():
    rc('font', weight='bold')     # bold fonts are easier to see
    rc('tick', labelsize=15)     # tick labels bigger
    rc('lines', lw=1, color='k') # thicker black lines (no budget for color!)
    rc('grid', c='0.5', ls='-', lw=0.5) # solid gray grid lines
    rc('savefig', dpi=300)       # higher res outputs

Then as you are working interactively, you just need to do

```python
>>> set_pub()
>>> subplot(111)
```
>>> plot([1, 2, 3])
>>> savefig('myfig')
>>> rcdefaults() # restore the defaults

""

import matplotlib.pyplot as plt

plt.subplot(311)
plt.plot([1, 2, 3])

# the axes attributes need to be set before the call to subplot
plt.rc('font', weight='bold')
plt.rc('xtick.major', size=5, pad=7)
plt.rc('xtick', labelsize=15)

# using aliases for color, linestyle and linewidth; gray, solid, thick
plt.rc('grid', c='0.5', ls='-', lw=5)
plt.rc('lines', lw=2, color='g')
plt.subplot(312)

plt.plot([1, 2, 3])
plt.grid(True)

plt.rcdefaults()
plt.subplot(313)
plt.plot([1, 2, 3])
plt.grid(True)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
```python
import matplotlib.pyplot as plt

DATA = ((1, 3),
        (2, 4),
        (3, 1),
        (4, 2))

dash_style = (
    (0, 20, -15, 30, 10),
    (1, 30, 0, 15, 10),
    (0, 40, 15, 15, 10),
    (1, 20, 30, 60, 10),
)

fig, ax = plt.subplots()

(x, y) = zip(*DATA)
ax.plot(x, y, marker='o')
for i in range(len(DATA)):
(x, y) = DATA[i]
(dd, dl, r, dr, dp) = dash_style[i]
# print('dashlen call %d % d')
t = ax.text(x, y, str((x, y)), withdash=True,
dashdirection=dd,
dashlength=dl,
rotation=r,
dashrotation=dr,
dashpush=dp,
)

ax.set_xlim((0.0, 5.0))
ax.set_ylim((0.0, 5.0))

plt.show()
from matplotlib.dates import DayLocator, HourLocator, DateFormatter, drange
from numpy import arange

date1 = datetime.datetime(2000, 3, 2)
date2 = datetime.datetime(2000, 3, 6)
delta = datetime.timedelta(hours=6)
dates = drange(date1, date2, delta)

y = arange(len(dates)*1.0)

fig, ax = plt.subplots()
ax.plot_date(dates, y*y)

# this is superfluous, since the autoscaler should get it right, but
# use date2num and num2date to convert between dates and floats if
# you want; both date2num and num2date convert an instance or sequence
# to a float or sequence of floats, respectively.
ax.set_xlim(dates[0], dates[-1])

# The hour locator takes the hour or sequence of hours you want to
# tick, not the base multiple.
ax.xaxis.set_major_locator(DayLocator())
ax.xaxis.set_minor_locator(HourLocator(arange(0, 25, 6)))
ax.xaxis.set_major_formatter(DateFormatter('%Y-%m-%d'))
ax.fmt_xdata = DateFormatter('%Y-%m-%d %H:%M:%S')
fig.autofmt_xdate()

plt.show()
Show how to use an rrule instance to make a custom date ticker - here we put a tick mark on every 5th easter

See https://moin.conectiva.com.br/DateUtil for help with rrules

```python
import matplotlib.pyplot as plt
from matplotlib.dates import YEARLY, DateFormatter, rrulewrapper, RRuleLocator, drange
import numpy as np
import datetime

# tick every 5th easter
rule = rrulewrapper(YEARLY, byeaster=1, interval=5)
loc = RRuleLocator(rule)
formatter = DateFormatter('%m/%d/%Y')
date1 = datetime.date(1952, 1, 1)
date2 = datetime.date(2004, 4, 12)
delta = datetime.timedelta(days=100)

dates = drange(date1, date2, delta)
s = np.random.rand(len(dates))  # make up some random y values
```

When plotting daily data, a frequent request is to plot the data ignoring skips, e.g., no extra spaces for weekends. This is particularly common in financial time series, when you may have data for M-F and not Sat, Sun and you don't want gaps in the x axis. The approach is to simply use the integer index for the xdata and a custom tick formatter to get the appropriate date string for a given index.
```python
from __future__ import print_function
import numpy as np
from matplotlib.mlab import csv2rec
import matplotlib.pyplot as plt
import matplotlib.cbook as cbook
from matplotlib.ticker import Formatter

datafile = cbook.get_sample_data('msft.csv', asfileobj=False)
print('loading %s' % datafile)
r = csv2rec(datafile)[-40:]

class MyFormatter(Formatter):
    def __init__(self, dates, fmt='%Y-%m-%d'):
        self.dates = dates
        self.fmt = fmt

    def __call__(self, x, pos=0):
        'Return the label for time x at position pos'
        ind = int(np.round(x))
        if ind >= len(self.dates) or ind < 0:
            return ''

        return self.dates[ind].strftime(self.fmt)

formatter = MyFormatter(r.date)

fig, ax = plt.subplots()
ax.xaxis.set_major_formatter(formatter)
ax.plot(np.arange(len(r)), r.close, 'o-')
fig.autofmt_xdate()
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
```python
import matplotlib.pyplot as plt
import numpy as np
import matplotlib.cm as cm
import matplotlib.mlab as mlab

def smooth1d(x, window_len):
    # copied from http://www.scipy.org/Cookbook/SignalSmooth

    s = np.r_[2*x[0] - x[window_len:1:-1], x, 2*x[-1] - x[-1:-window_len:-1]]
    w = np.hanning(window_len)
    y = np.convolve(w/w.sum(), s, mode='same')
```
```python
return y[window_len-1:-window_len+1]

def smooth2d(A, sigma=3):
    window_len = max(int(sigma), 3)*2 + 1
    A1 = np.array([smooth1d(x, window_len) for x in np.asarray(A)])
    A2 = np.transpose(A1)
    A3 = np.array([smooth1d(x, window_len) for x in A2])
    A4 = np.transpose(A3)

    return A4

class BaseFilter(object):
    def prepare_image(self, src_image, dpi, pad):
        ny, nx, depth = src_image.shape
        # tgt_image = np.zeros([pad*2+ny, pad*2+nx, depth], dtype="d")
        padded_src = np.zeros([pad*2 + ny, pad*2 + nx, depth], dtype="d")
        padded_src[pad:-pad, pad:-pad, :] = src_image[:, :, :]

        return padded_src  # , tgt_image

def get_pad(self, dpi):
    return 0

def __call__(self, im, dpi):
    pad = self.get_pad(dpi)
    padded_src = self.prepare_image(im, dpi, pad)
    tgt_image = self.process_image(padded_src, dpi)
    return tgt_image, -pad, -pad

class OffsetFilter(BaseFilter):
    def __init__(self, offsets=None):
        if offsets is None:
            self.offsets = (0, 0)
        else:
            self.offsets = offsets

def get_pad(self, dpi):
    return int(max(*self.offsets)/72.*dpi)

def process_image(self, padded_src, dpi):
    ox, oy = self.offsets
    a1 = np.roll(padded_src, int(ox/72.*dpi), axis=1)
    a2 = np.roll(a1, -int(oy/72.*dpi), axis=0)
    return a2

class GaussianFilter(BaseFilter):
    "simple gauss filter"
```
def __init__(self, sigma, alpha=0.5, color=None):
    self.sigma = sigma
    self.alpha = alpha
    if color is None:
        self.color = (0, 0, 0)
    else:
        self.color = color

def get_pad(self, dpi):
    return int(self.sigma*3/72.*dpi)

def process_image(self, padded_src, dpi):
    tgt_image = np.zeros_like(padded_src)
    aa = smooth2d(padded_src[:, :, -1]*self.alpha,
                  self.sigma/72.*dpi)
    tgt_image[:, :, -1] = aa
    tgt_image[:, :, :-1] = self.color
    return tgt_image

class DropShadowFilter(BaseFilter):
    def __init__(self, sigma, alpha=0.3, color=None, offsets=None):
        self.gauss_filter = GaussianFilter(sigma, alpha, color)
        self.offset_filter = OffsetFilter(offsets)

def get_pad(self, dpi):
    return max(self.gauss_filter.get_pad(dpi),
               self.offset_filter.get_pad(dpi))

def process_image(self, padded_src, dpi):
    t1 = self.gauss_filter.process_image(padded_src, dpi)
    t2 = self.offset_filter.process_image(t1, dpi)
    return t2

from matplotlib.colors import LightSource

class LightFilter(BaseFilter):
    "simple gauss filter"

    def __init__(self, sigma, fraction=0.5):
        self.gauss_filter = GaussianFilter(sigma, alpha=1)
        self.light_source = LightSource()
        self.fraction = fraction

    def get_pad(self, dpi):
        return self.gauss_filter.get_pad(dpi)

    def process_image(self, padded_src, dpi):
        tl = self.gauss_filter.process_image(padded_src, dpi)
        elevation = tl[:, :, 3]
rgb = padded_src[:, :, :3]

rgb2 = self.light_source.shade_rgb(rgb, elevation,
    fraction=self.fraction)

tgt = np.empty_like(padded_src)
tgt[:, :, :3] = rgb2
tgt[:, :, 3] = padded_src[:, :, 3]

return tgt

class GrowFilter(BaseFilter):
    "enlarge the area"

    def __init__(self, pixels, color=None):
        self.pixels = pixels
        if color is None:
            self.color = (1, 1, 1)
        else:
            self.color = color

        def __call__(self, im, dpi):
            pad = self.pixels
            ny, nx, depth = im.shape
            new_im = np.empty([pad*2 + ny, pad*2 + nx, depth], dtype="d")
            alpha = new_im[:, :, 3]
            alpha.fill(0)
            alpha[pad:-pad, pad:-pad] = im[:, :, -1]
            alpha2 = np.clip(smooth2d(alpha, self.pixels/72.*dpi) * 5, 0, 1)
            new_im[:, :, -1] = alpha2
            new_im[:, :, :-1] = self.color
            offsetx, offsety = -pad, -pad

            return new_im, offsetx, offsety

from matplotlib.artist import Artist

class FilteredArtistList(Artist):
    """
    A simple container to draw filtered artist.
    """

    def __init__(self, artist_list, filter):
        self._artist_list = artist_list
        self._filter = filter
        Artist.__init__(self)

    def draw(self, renderer):
        renderer.start_rasterizing()
        renderer.start_filter()
for a in self._artist_list:
    a.draw(renderer)
    renderer.stop_filter(self._filter)
    renderer.stop_rasterizing()

import matplotlib.transforms as mtransforms

def filtered_text(ax):
    # mostly copied from contour_demo.py

    # prepare image
    delta = 0.025
    x = np.arange(-3.0, 3.0, delta)
    y = np.arange(-2.0, 2.0, delta)
    X, Y = np.meshgrid(x, y)
    Z1 = mlab.bivariate_normal(X, Y, 1.0, 1.0, 0.0, 0.0)
    Z2 = mlab.bivariate_normal(X, Y, 1.5, 0.5, 1, 1)
    Z = 10.0 * (Z2 - Z1)

    # draw
    im = ax.imshow(Z, interpolation='bilinear', origin='lower',
                   cmap=cm.gray, extent=(-3, 3, -2, 2))
    levels = np.arange(-1.2, 1.6, 0.2)
    CS = ax.contour(Z, levels,
                    origin='lower',
                    linewidths=2,
                    extent=(-3, 3, -2, 2))

    ax.set_aspect("auto")

    # contour label
    cl = ax.clabel(CS, levels[1::2],
                   inline=1,
                   fmt='%1.1f',
                   fontsize=11)

    # change clable color to black
    from matplotlib.patheffects import Normal
    for t in cl:
        t.set_color("k")
        # to force TextPath (i.e., same font in all backends)
        t.set_path_effects([Normal()])

    # Add white glows to improve visibility of labels.
    white_glows = FilteredArtistList(cl, GrowFilter(3))
    ax.add_artist(white_glows)
    white_glows.set_zorder(cl[0].get_zorder() - 0.1)

    ax.xaxis.set_visible(False)
    ax.yaxis.set_visible(False)
def drop_shadow_line(ax):
    # copied from examples/misc/svg_filter_line.py

    # draw lines
    l1, = ax.plot([0.1, 0.5, 0.9], [0.1, 0.9, 0.5], "bo-",
                  mec="b", mfc="w", lw=5, mew=3, ms=10, label="Line 1")
    l2, = ax.plot([0.1, 0.5, 0.9], [0.5, 0.2, 0.7], "ro-",
                  mec="r", mfc="w", lw=5, mew=3, ms=10, label="Line 1")

    gauss = DropShadowFilter(4)

    for l in [l1, l2]:
        # draw shadows with same lines with slight offset.
        xx = l.get_xdata()
        yy = l.get_ydata()
        shadow, = ax.plot(xx, yy)
        shadow.update_from(l)

        # offset transform
        ot = mtransforms.offset_copy(l.get_transform(), ax.figure,
                                      x=4.0, y=-6.0, units='points')
        shadow.set_transform(ot)

        # adjust zorder of the shadow lines so that it is drawn below the
        # original lines
        shadow.set_zorder(l.get_zorder() - 0.5)
        shadow.set_agg_filter(gauss)
        shadow.set_rasterized(True)  # to support mixed-mode renderers

    ax.set_xlim(0., 1.)
    ax.set_ylim(0., 1.)
    ax.xaxis.set_visible(False)
    ax.yaxis.set_visible(False)

def drop_shadow_patches(ax):
    # Copied from barchart_demo.py

    N = 5
    menMeans = (20, 35, 30, 35, 27)

    ind = np.arange(N)  # the x locations for the groups
    width = 0.35        # the width of the bars

    rects1 = ax.bar(ind, menMeans, width, color='r', ec="w", lw=2)

    womenMeans = (25, 32, 34, 20, 25)
    rects2 = ax.bar(ind + width + 0.1, womenMeans, width, color='y', ec="w", lw=2)
```python
# gauss = GaussianFilter(1.5, offsets=(1,1), )
gauss = DropShadowFilter(5, offsets=(1, 1), )
shadow = FilteredArtistList(rects1 + rects2, gauss)
ax.add_artist(shadow)
shadow.set_zorder(rects1[0].get_zorder() - 0.1)

ax.set_ylim(0, 40)
ax.xaxis.set_visible(False)
ax.yaxis.set_visible(False)

def light_filter_pie(ax):
    fracs = [15, 30, 45, 10]
    explode = (0, 0.05, 0, 0)
    pies = ax.pie(fracs, explode=explode)
    ax.patch.set_visible(True)
    light_filter = LightFilter(9)
    for p in pies[0]:
        p.set_agg_filter(light_filter)
        p.set_rasterized(True)  # to support mixed-mode renderers
        p.set(ec="none",
              lw=2)

    gauss = DropShadowFilter(9, offsets=(3, 4), alpha=0.7)
    shadow = FilteredArtistList(pies[0], gauss)
    ax.add_artist(shadow)
    shadow.set_zorder(pies[0][0].get_zorder() - 0.1)

if 1:

    plt.figure(1, figsize=(6, 6))
    plt.subplots_adjust(left=0.05, right=0.95)

    ax = plt.subplot(221)
    filtered_text(ax)

    ax = plt.subplot(222)
    drop_shadow_line(ax)

    ax = plt.subplot(223)
    drop_shadow_patches(ax)

    ax = plt.subplot(224)
    ax.set_aspect(1)
    light_filter_pie(ax)
    ax.set_frame_on(True)

    plt.show()
```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.patches import Circle
from matplotlib.offsetbox import (TextArea, DrawingArea, OffsetImage, AnnotationBbox)
from matplotlib.cbook import get_sample_data

if 1:
    fig, ax = plt.subplots()

    # Define 1st position to annotate (display it with a marker)
    xy = (0.5, 0.7)
    ax.plot(xy[0], xy[1], ".r")

    # Annotate the 1st position with a text box ('Test 1')
    offsetbox = TextArea("Test 1", minimumdescent=False)
ab = AnnotationBbox(offsetbox, xy,
        xybox=(-20, 40),
        xycoords='data',
        boxcoords="offset points",
        arrowprops=dict(arrowstyle="->"))
ax.add_artist(ab)

# Annotate the 1st position with another text box ('Test')
offsetbox = TextArea("Test", minimumdescent=False)
ab = AnnotationBbox(offsetbox, xy,
        xybox=(1.02, xy[1]),
        xycoords='data',
        boxcoords=("axes fraction", "data"),
        box_alignment=(0., 0.5),
        arrowprops=dict(arrowstyle="->"))
ax.add_artist(ab)

# Define a 2nd position to annotate (don't display with a marker this time)
xy = [0.3, 0.55]

# Annotate the 2nd position with a circle patch
da = DrawingArea(20, 20, 0, 0)
p = Circle((10, 10), 10)
da.add_artist(p)

ab = AnnotationBbox(da, xy,
        xybox=(1.02, xy[1]),
        xycoords='data',
        boxcoords=("axes fraction", "data"),
        box_alignment=(0., 0.5),
        arrowprops=dict(arrowstyle="->"))
ax.add_artist(ab)

# Annotate the 2nd position with an image (a generated array of pixels)
arr = np.arange(100).reshape((10, 10))
im = OffsetImage(arr, zoom=2)
im.image.axes = ax

ab = AnnotationBbox(im, xy,
        xybox=(-50., 50.),
        xycoords='data',
        boxcoords="offset points",
        pad=0.3,
        arrowprops=dict(arrowstyle="->"))
ax.add_artist(ab)

# Annotate the 2nd position with another image (a Grace Hopper portrait)
fn = get_sample_data("grace_hopper.png", asfileobj=False)
arr_img = plt.imread(fn, format='png')
imagebox = OffsetImage(arr_img, zoom=0.2)
imagebox.image.axes = ax

ab = AnnotationBbox(imagebox, xy,
                      xybox=(120, -80),
                      xycoords='data',
                      boxcoords="offset points",
                      pad=0.5,
                      arrowprops=dict(
                          arrowstyle="->",
                          connectionstyle="angle,angleA=0,angleB=90,rad=3"
                      )
)

ax.add_artist(ab)

# Fix the display limits to see everything
ax.set_xlim(0, 1)
ax.set_ylim(0, 1)

plt.show()
import matplotlib.pyplot as plt
import numpy as np
from matplotlib.image import BboxImage
from matplotlib.transforms import Bbox, TransformedBbox

if __name__ == "__main__":
    fig = plt.figure(1)
    ax = plt.subplot(121)

    txt = ax.text(0.5, 0.5, "test", size=30, ha="center", color="w")
    kwargs = dict()

    bbox_image = BboxImage(txt.get_window_extent, norm=None, origin=None, clip_on=False, **kwargs)

    a = np.arange(256).reshape(1, 256)/256.
    bbox_image.set_data(a)
    ax.add_artist(bbox_image)
```python
ax = plt.subplot(122)
a = np.linspace(0, 1, 256).reshape(1, -1)
a = np.vstack((a, a))

maps = sorted(m
    for m in plt.cm.cmap_d
    if not m.endswith("_r") and  # Skip reversed colormaps.
        not m.startswith(('spectral', 'Vega'))  # Skip deprecated colormaps.
)

#fig.subplots_adjust(top=0.99, bottom=0.01, left=0.2, right=0.99)

ncol = 2
nrow = len(maps)//ncol + 1

xpad_fraction = 0.3
dx = 1./(ncol + xpad_fraction*(ncol - 1))

ypad_fraction = 0.3
dy = 1./(nrow + ypad_fraction*(nrow - 1))

for i, m in enumerate(maps):
    ix, iy = divmod(i, nrow)
    #plt.figimage(a, 10, i*10, cmap=plt.get_cmap(m), origin='lower')
bbox0 = Bbox.from_bounds(ix*dx*(1 + xpad_fraction),
                           1. - iy*dy*(1 + ypad_fraction) - dy,
                           dx, dy)
bbox = TransformedBbox(bbox0, ax.transAxes)

bbox_image = BboxImage(bbox,
                        cmap=plt.get_cmap(m),
                        norm=None,
                        origin=None,
                        **kwargs)

bbox_image.set_data(a)
ax.add_artist(bbox_image)

plt.draw()
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
import matplotlib.pyplot as plt
import numpy as np
from matplotlib.image import BboxImage

from matplotlib._png import read_png
import matplotlib.colors
from matplotlib.cbook import get_sample_data

class RibbonBox(object):
    original_image = read_png(get_sample_data("Minduka_Present_Blue_Pack.png", asfileobj=False))

cut_location = 70
b_and_h = original_image[:, :, 2]
color = original_image[:, :, 2] - original_image[:, :, 0]
alpha = original_image[:, :, 3]
xn = original_image.shape[1]

    def __init__(self, color):
        rgb = matplotlib.colors.to_rgba(color)[:3]
im = np.empty(self.original_image.shape,
               self.original_image.dtype)

im[:, :, :3] = self.b_and_h[:, :, np.newaxis]
im[:, :, :3] -= self.color[:, :, np.newaxis]*(1. - np.array(rgb))
im[:, :, 3] = self.alpha

self.im = im

def get_stretched_image(self, stretch_factor):
    stretch_factor = max(stretch_factor, 1)
    ny, nx, nch = self.im.shape
    ny2 = int(ny*stretch_factor)

    stretched_image = np.empty((ny2, nx, nch),
                                self.im.dtype)
    cut = self.im[self.cut_location, :, :]
    stretched_image[:, :, :] = cut

    stretched_image[:, self.cut_location, :, :] = 
        self.im[:, self.cut_location, :, :]
    stretched_image[-(ny - self.cut_location):, :, :] = 
        self.im[-(ny - self.cut_location):, :, :]

    self._cached_im = stretched_image
    return stretched_image

class RibbonBoxImage(BboxImage):
    zorder = 1

    def __init__(self, bbox, color, 
                  cmap=None, 
                  norm=None, 
                  interpolation=None, 
                  origin=None, 
                  filternorm=1, 
                  filterrad=4.0, 
                  resample=False, 
                  **kwargs):

        BboxImage.__init__(self, bbox, 
                           cmap=cmap, 
                           norm=norm, 
                           interpolation=interpolation, 
                           origin=origin, 
                           filternorm=filternorm, 
                           filterrad=filterrad, 
                           resample=resample, 
                           **kwargs)

        self._ribbonbox = RibbonBox(color)
```python
self._cached_ny = None

def draw(self, renderer, *args, **kwargs):
    bbox = self.get_window_extent(renderer)
    stretch_factor = bbox.height / bbox.width

    ny = int(stretch_factor * self._ribbonbox.nx)
    if self._cached_ny != ny:
        arr = self._ribbonbox.get_stretched_image(stretch_factor)
        self.set_array(arr)
        self._cached_ny = ny
        BboxImage.draw(self, renderer, *args, **kwargs)

if 1:
    from matplotlib.transforms import Bbox, TransformedBbox
    from matplotlib.ticker import ScalarFormatter

fig, ax = plt.subplots()

years = np.arange(2004, 2009)
box_colors = [(0.8, 0.2, 0.2),
              (0.2, 0.8, 0.2),
              (0.2, 0.2, 0.8),
              (0.7, 0.5, 0.8),
              (0.3, 0.8, 0.7),
             ]

heights = np.random.random(years.shape) * 7000 + 3000

fmt = ScalarFormatter(useOffset=False)
ax.xaxis.set_major_formatter(fmt)

for year, h, bc in zip(years, heights, box_colors):
    bbox0 = Bbox.from_extents(year - 0.4, 0., year + 0.4, h)
    bbox = TransformedBbox(bbox0, ax.transData)
    rb_patch = RibbonBoxImage(bbox, bc, interpolation="bicubic")
    ax.add_artist(rb_patch)

    ax.annotate(r"%d" % (int(h/100.)*100),
                (year, h), va="bottom", ha="center")

patch_gradient = BboxImage(ax.bbox,
                           interpolation="bicubic",
                           zorder=0.1,
                          )
gradient = np.zeros((2, 2, 4), dtype=np.float)
gradient[:,:,0] = [1, 1, 0.]
gradient[:,:,3] = [0.1, 0.3, [0.3, 0.5]] # alpha channel
patch_gradient.set_array(gradient)
ax.add_artist(patch_gradient)
```
ax.set_xlim(years[0] - 0.5, years[-1] + 0.5)
ax.set_ylim(0, 10000)

fig.savefig('ribbon_box.png')
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

99.58  pylab_examples example code: demo_text_path.py

# -*- coding: utf-8 -*-

import matplotlib.pyplot as plt
from matplotlib.image import BboxImage
import numpy as np
from matplotlib.transforms import IdentityTransform
import matplotlib.patches as mpatches

from matplotlib.offsetbox import AnnotationBbox,
     AnchoredOffsetbox, AuxTransformBox

# pylab_examples example code: demo_text_path.py 2473
from matplotlib.cbook import get_sample_data
from matplotlib.text import Path

class PathClippedImagePatch(mpatches.PathPatch):
    """
    The given image is used to draw the face of the patch. Internally, it uses BboxImage whose clippath set to the path of the patch.
    """
    def __init__(self, path, bbox_image, **kwargs):
        mpatches.PathPatch.__init__(self, path, **kwargs)
        self._init_bbox_image(bbox_image)

    def set_facecolor(self, color):
        """simply ignore facecolor"
        mpatches.PathPatch.set_facecolor(self, "none")

    def _init_bbox_image(self, im):
        bbox_image = BboxImage(self.get_window_extent,
                               norm=None,
                               origin=None,
                               )
        bbox_image.set_transform(IdentityTransform())
        bbox_image.set_data(im)
        self.bbox_image = bbox_image

    def draw(self, renderer=None):
        # the clip path must be updated every draw. any solution? -JJ
        self.bbox_image.set_clip_path(self._path, self.get_transform())
        self.bbox_image.draw(renderer)
        mpatches.PathPatch.draw(self, renderer)

if 1:
    usetex = plt.rcParams["text.usetex"]
    fig = plt.figure(1)
    # EXAMPLE 1
    ax = plt.subplot(211)
    from matplotlib._png import read_png
fn = get_sample_data("grace_hopper.png", asfileobj=False)
arr = read_png(fn)

text_path = TextPath((0, 0), "!?", size=150)
p = PathClippedImagePatch(text_path, arr, ec="k",
                          transform=IdentityTransform())

#p.set_clip_on(False)

# make offset box
offsetbox = AuxTransformBox(IdentityTransform())
offsetbox.add_artist(p)

# make anchored offset box
ao = AnchoredOffsetbox(loc=2, child=offsetbox, frameon=True, borderpad=0.2)
ax.add_artist(ao)

# another text
from matplotlib.patches import PathPatch
if usetex:
    r = r"\mbox{textpath supports mathtext & \TeX}"
else:
    r = r"textpath supports mathtext & TeX"

text_path = TextPath((0, 0), r,
                     size=20, usetex=usetex)

p1 = PathPatch(text_path, ec="w", lw=3, fc="w", alpha=0.9,
               transform=IdentityTransform())
p2 = PathPatch(text_path, ec="none", fc="k",
               transform=IdentityTransform())

offsetbox2 = AuxTransformBox(IdentityTransform())
offsetbox2.add_artist(p1)
offsetbox2.add_artist(p2)

ab = AnnotationBbox(offsetbox2, (0.95, 0.05), xycoords='axes fraction',
                     boxcoords="offset points",
                     box_alignment=(1., 0.),
                     frameon=False)
ax.add_artist(ab)

ax.imshow([[0, 1, 2], [1, 2, 3]], cmap=plt.cm.gist_gray_r,
           interpolation="bilinear",
           aspect="auto")

# EXAMPLE 2

ax = plt.subplot(212)
arr = np.arange(256).reshape(1, 256)/256.
```python
if usetex:
    s = r"$\displaystyle\left\{\sum_{n=1}^{\infty}\frac{-e^{i\pi}}{2^n}\right\}$!"
else:
    s = r"$\left\{\sum_{n=1}^{\infty}\frac{-e^{i\pi}}{2^n}\right\}$!"

text_path = TextPath((0, 0), s, size=40, usetex=usetex)
text_patch = PathClippedImagePatch(text_path, arr, ec="none",
    transform=IdentityTransform())

shadow1 = mpatches.Shadow(text_patch, 1, -1, props=dict(fc="none", ec="0.6", lw=3))
shadow2 = mpatches.Shadow(text_patch, 1, -1, props=dict(fc="0.3", ec="none"))

# make offset box
offsetbox = AuxTransformBox(IdentityTransform())
offsetbox.add_artist(shadow1)
offsetbox.add_artist(shadow2)
offsetbox.add_artist(text_patch)

# place the anchored offset box using AnnotationBbox
ab = AnnotationBbox(offsetbox, (0.5, 0.5),
    xycoords='data',
    boxcoords="offset points",
    box_alignment=(0.5, 0.5),
)

#text_path.set_size(10)

ax.add_artist(ab)

ax.set_xlim(0, 1)
ax.set_ylim(0, 1)

plt.draw()
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
from mpl_toolkits.axes_grid1.axes_grid import ImageGrid

def test_rotation_mode(fig, mode, subplot_location):
    ha_list = "left center right".split()
    va_list = "top center baseline bottom".split()
    grid = ImageGrid(fig, subplot_location,
                     nrows_ncols=(len(va_list), len(ha_list)),
                     share_all=True, aspect=True,
                     #label_mode='1',
                     cbar_mode=None)

    for ha, ax in zip(ha_list, grid.axes_row[-1]):
        ax.axis["bottom"].label.set_text(ha)

    grid.axes_row[0][1].set_title(mode, size="large")

    for va, ax in zip(va_list, grid.axes_column[0]):
        ax.axis["left"].label.set_text(va)

    i = 0
    for va in va_list:
        for ha in ha_list:
            ax = grid[i]
            for axis in ax.axis.values():
axis.toggle(ticks=False, ticklabels=False)
ax.text(0.5, 0.5, "Tpg",
    size="large", rotation=40,
    bbox=dict(boxstyle="square,pad=0.",
        ec="none", fc="0.5", alpha=0.5),
    ha=ha, va=va,
    rotation_mode=mode)
ax.axvline(0.5)
ax.axhline(0.5)
i += 1

if 1:
    import matplotlib.pyplot as plt
    fig = plt.figure(1, figsize=(5.5, 4))
    fig.clf()

    test_rotation_mode(fig, "default", 121)
    test_rotation_mode(fig, "anchor", 122)
    plt.show()
99.60  pylab_examples example code: demo_tight_layout.py

Title

y-label

x-label
Matplotlib, Release 2.0.2

pylab_examples example code: demo_tight_layout.py

99.60. pylab_examples example code: demo_tight_layout.py
Matplotlib, Release 2.0.2

Chapter 99.  *pylab_examples* Examples
Title

y-label

x-label

y-label

x-label

Title

y-label

x-label

y-label

x-label
import matplotlib.pyplot as plt
import itertools
import warnings

fontsizes = itertools.cycle([8, 16, 24, 32])

def example_plot(ax):
    ax.plot([1, 2])
    ax.set_xlabel('x-label', fontsize=next(fontsizes))
    ax.set_ylabel('y-label', fontsize=next(fontsizes))
    ax.set_title('Title', fontsize=next(fontsizes))

fig, ax = plt.subplots()
example_plot(ax)
plt.tight_layout()

fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(nrows=2, ncols=2)
example_plot(ax1)
example_plot(ax2)
example_plot(ax3)
example_plot(ax4)
plt.tight_layout()
```python
def example_plot(ax):
    pass

def main():
    fig, (ax1, ax2) = plt.subplots(nrows=2, ncols=1)
    example_plot(ax1)
    example_plot(ax2)
    plt.tight_layout()

    fig, (ax1, ax2) = plt.subplots(nrows=1, ncols=2)
    example_plot(ax1)
    example_plot(ax2)
    plt.tight_layout()

    fig, axes = plt.subplots(nrows=3, ncols=3)
    for row in axes:
        for ax in row:
            example_plot(ax)
    plt.tight_layout()

    fig = plt.figure()
    ax1 = plt.subplot(221)
    ax2 = plt.subplot(223)
    ax3 = plt.subplot(122)
    example_plot(ax1)
    example_plot(ax2)
    example_plot(ax3)
    plt.tight_layout()

    fig = plt.figure()
    ax1 = plt.subplot2grid((3, 3), (0, 0))
    ax2 = plt.subplot2grid((3, 3), (0, 1), colspan=2)
    ax3 = plt.subplot2grid((3, 3), (1, 0), colspan=2, rowspan=2)
    ax4 = plt.subplot2grid((3, 3), (1, 2), rowspan=2)
    example_plot(ax1)
    example_plot(ax2)
    example_plot(ax3)
    example_plot(ax4)
    plt.tight_layout()

    plt.show()

    fig = plt.figure()

    import matplotlib.gridspec as gridspec
    gs1 = gridspec.GridSpec(3, 1)
    ax1 = fig.add_subplot(gs1[0])
```
ax2 = fig.add_subplot(gs1[1])
ax3 = fig.add_subplot(gs1[2])

example_plot(ax1)
example_plot(ax2)
example_plot(ax3)

with warnings.catch_warnings():
    warnings.simplefilter("ignore", UserWarning)
    # This raises warnings since tight layout cannot
    # handle gridspec automatically. We are going to
    # do that manually so we can filter the warning.
    gs1.tight_layout(fig, rect=[None, None, 0.45, None])

gs2 = gridspec.GridSpec(2, 1)
ax4 = fig.add_subplot(gs2[0])
ax5 = fig.add_subplot(gs2[1])

example_plot(ax4)
example_plot(ax5)

with warnings.catch_warnings():
    # This raises warnings since tight layout cannot
    # handle gridspec automatically. We are going to
    # do that manually so we can filter the warning.
    warnings.simplefilter("ignore", UserWarning)
    gs2.tight_layout(fig, rect=[0.45, None, None, None])

# now match the top and bottom of two gridspecs.
top = min(gs1.top, gs2.top)
bottom = max(gs1.bottom, gs2.bottom)

gs1.update(top=top, bottom=bottom)
gs2.update(top=top, bottom=bottom)

plt.show()
```python
import matplotlib.cm as cm
import matplotlib.pyplot as plt
from matplotlib.patches import Circle, PathPatch
from matplotlib.path import Path
from matplotlib.transforms import Affine2D
import numpy as np

r = np.random.rand(50)
t = np.random.rand(50) * np.pi * 2.0
x = r * np.cos(t)
y = r * np.sin(t)
```
```python
fig, ax = plt.subplots(figsize=(6, 6))
circle = Circle((0, 0), 1, facecolor='none',
                edgecolor=(0, 0.8, 0.8), linewidth=3, alpha=0.5)
ax.add_patch(circle)

im = plt.imshow(np.random.random((100, 100)),
                 origin='lower', cmap=cm.winter,
                 interpolation='spline36',
                 extent=(-1, 1, -1, 1))

im.set_clip_path(circle)

plt.plot(x, y, 'o', color=(0.9, 0.9, 1.0), alpha=0.8)
```

---

Dolphin from OpenClipart library by Andy Fitzsimon

```text
# <cc:License rdf:about="http://web.resource.org/cc/PublicDomain">
# <cc:permits rdf:resource="http://web.resource.org/cc/Reproduction"/>
# <cc:permits rdf:resource="http://web.resource.org/cc/Distribution"/>
# </cc:License>

```

```text
dolphin = """
M -0.59739425,160.18173 C -0.62740401,160.18885 -0.57867129,160.11183
-0.57867129,160.11183 C -0.5438361,159.89315
-0.3951468,159.81496 C -0.24645668,159.73678 -0.18316813,159.71981
-0.18316813,159.71981 C -0.10322971,159.58124
-0.057804323,159.58725 C -0.029723983,159.58913 -0.01841603,159.60356
-0.017265183,159.62815 C -0.000250183,159.63252 -0.002918513,159.70554
-0.006184293,159.71248 C -0.0040763903,159.7194 -0.0066711426,159.71091
0.077336307,159.73612 C 0.16879567,159.76377 0.28380306,159.86448
0.31516668,159.91533 C 0.3463503,159.96618 0.5011127,160.1771
0.5011127,160.1771 C 0.63668998,160.19238 0.67763022,160.31259
0.66556395,160.32668 C 0.65339985,160.34212 0.66350443,160.33642
0.64907908,160.33088 C 0.63463742,160.32533 0.61309688,160.2974
0.5789627,160.29339 C 0.54348657,160.28968 0.52329693,160.27674
0.50728856,160.27737 C 0.49060916,160.27975 0.48965803,160.31565
0.4614204,160.33673 C 0.43329696,160.35786 0.4570711,160.39871
0.43309565,160.40685 C 0.4105108,160.41442 0.39416631,160.33027
0.3954995,160.2935 C 0.3963269,160.25672 0.43807996,160.21522
0.44567915,160.19734 C 0.45327833,160.17946 0.27948689,159.9424
-0.061852613,159.99845 C -0.083965233,160.0427 -0.26176109,160.06683
-0.26176109,160.06683 C -0.30127962,160.07028 -0.21167141,160.09731
-0.24649368,160.1011 C -0.32642366,160.11569 -0.34521187,160.06895
-0.40622293,160.0819 C -0.467234,160.09485 -0.56738444,160.17461
-0.59739425,160.18173
"""
```

```python
vertices = []
codes = []
parts = dolphin.split()
i = 0
code_map = {
    'M': (Path.MOVETO, 1),
    'C': (Path.CURVE4, 3),
}```
'L': (Path.LINETO, 1)
}

while i < len(parts):
    code = parts[i]
    path_code, npoints = code_map[code]
    codes.extend([path_code] * npoints)
    vertices.extend([[float(x) for x in y.split(',')] for y in parts[i + 1:i + npoints + 1]])
    i += npoints + 1
vertices = np.array(vertices, np.float)
vertices[::, 1] -= 160

dolphin_path = Path(vertices, codes)
dolphin_patch = PathPatch(dolphin_path, facecolor=(0.6, 0.6, 0.6),
edgecolor=(0.0, 0.0, 0.0))
ax.add_patch(dolphins_path)

vertices = Affine2D().rotate_deg(60).transform(vertices)
dolphin_path2 = Path(vertices, codes)
dolphin_patch2 = PathPatch(dolphin_path2, facecolor=(0.5, 0.5, 0.5),
edgecolor=(0.0, 0.0, 0.0))
ax.add_patch(dolphins_path2)

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
import matplotlib.pyplot as plt
import numpy as np
from matplotlib.collections import EllipseCollection

x = np.arange(10)
y = np.arange(15)
X, Y = np.meshgrid(x, y)

XY = np.hstack((X.ravel()[:, np.newaxis], Y.ravel()[:, np.newaxis]))

ww = X/10.0
hh = Y/15.0
aa = X*9

fig, ax = plt.subplots()
ec = EllipseCollection(ww, hh, aa, units='x', offsets=XY,
                       transOffset=ax.transData)
ect.set_array((X + Y).ravel())
ax.add_collection(ec)
ax.autoscale_view()
```python
import matplotlib.pyplot as plt
import numpy.random as rnd
from matplotlib.patches import Ellipse

NUM = 250

ells = [Ellipse(xy=rnd.rand(2)*10, width=rnd.rand(), height=rnd.rand(), angle=rnd.rand()*360)
        for i in range(NUM)]

fig = plt.figure(0)
ax = fig.add_subplot(111, aspect='equal')

ax.set_xlabel('X')
ax.set_ylabel('y')
cbar = plt.colorbar(ec)
cbar.set_label('X+Y')
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
```python
for e in ells:
    ax.add_artist(e)
    e.set_clip_box(ax.bbox)
    e.set_alpha(rnd.rand())
    e.set_facecolor(rnd.rand(3))

ax.set_xlim(0, 10)
ax.set_ylim(0, 10)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

99.64 `pylab_examples` example code: ellipse_rotated.py

```python
import matplotlib.pyplot as plt
import numpy as np
from matplotlib.patches import Ellipse

delta = 45.0  # degrees

angles = np.arange(0, 360 + delta, delta)
```
```python
ells = [Ellipse((1, 1), 4, 2, a) for a in angles]

a = plt.subplot(111, aspect='equal')

for e in ells:
    e.set_clip_box(a.bbox)
    e.set_alpha(0.1)
    a.add_artist(e)

plt.xlim(-2, 4)
plt.ylim(-1, 3)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

99.65 `pylab_examples` example code: `equal_aspect_ratio.py`

```
Example: simple line plot.
Show how to make a plot that has equal aspect ratio
```
import matplotlib.pyplot as plt
import numpy as np

t = np.arange(0.0, 1.0 + 0.01, 0.01)
s = np.cos(2*2*np.pi*t)
plt.plot(t, s, '-', lw=2)

plt.xlabel('time (s)')
plt.ylabel('voltage (mV)')
plt.title('About as simple as it gets, folks')
plt.grid(True)

plt.axes().set_aspect('equal', 'datalim')

plt.show()
Illustration of upper and lower limit symbols on errorbars

```python
import numpy as np
import matplotlib.pyplot as plt

fig = plt.figure(0)
x = np.arange(10.0)
y = np.sin(np.arange(10.0)/20.0*np.pi)
plt.errorbar(x, y, yerr=0.1)

y = np.sin(np.arange(10.0)/20.0*np.pi) + 1
plt.errorbar(x, y, yerr=0.1, uplims=True)

y = np.sin(np.arange(10.0)/20.0*np.pi) + 2
upperlimits = np.array([1, 0]*5)
lowerlimits = np.array([0, 1]*5)
plt.errorbar(x, y, yerr=0.1, uplims=upperlimits, lolims=lowerlimits)

plt.xlim(-1, 10)

fig = plt.figure(1)
x = np.arange(10.0)/10.0
y = (x + 0.1)**2
```

99.66. `pylab_examples` example code: `errorbar_limits.py`
```python
plt.errorbar(x, y, xerr=0.1, xlolims=True)
y = (x + 0.1)**3
plt.errorbar(x + 0.6, y, xerr=0.1, xuplims=upperlimits, xlolims=lowerlimits)
y = (x + 0.1)**4
plt.errorbar(x + 1.2, y, xerr=0.1, xuplims=True)
plt.xlim(-0.2, 2.4)
plt.ylim(-0.1, 1.3)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

99.67 pylab_examples example code: errorbar_subsample.py

```

Demo for the errorevery keyword to show data full accuracy data plots with few errorbars.

```
```python
import numpy as np
import matplotlib.pyplot as plt

# example data
x = np.arange(0.1, 4, 0.1)
y = np.exp(-x)

# example variable error bar values
yerr = 0.1 + 0.1*np.sqrt(x)

# Now switch to a more OO interface to exercise more features.
fig, axs = plt.subplots(nrows=1, ncols=2, sharex=True)
ax = axs[0]
ax.errorbar(x, y, yerr=yerr)
ax.set_title('all errorbars')

ax = axs[1]
ax.errorbar(x, y, yerr=yerr, errorevery=5)
ax.set_title('only every 5th errorbar')

fig.suptitle('Errorbar subsampling for better appearance')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
"Plot two curves, then use EventCollections to mark the locations of the x and y data points on the respective axes for each curve"

```python
import matplotlib.pyplot as plt
from matplotlib.collections import EventCollection
import numpy as np

# create random data
np.random.seed(50)
xdata = np.random.random([2, 10])

# split the data into two parts
xdata1 = xdata[0, :]
xdata2 = xdata[1, :]

# sort the data so it makes clean curves
xdata1.sort()
xdata2.sort()

# create some y data points
```
ydata1 = xdata1 ** 2  
ydata2 = 1 - xdata2 ** 3

# plot the data
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)
ax.plot(xdata1, ydata1, 'r', xdata2, ydata2, 'b')

# create the events marking the x data points
xevents1 = EventCollection(xdata1, color=[1, 0, 0], linelength=0.05)
xevents2 = EventCollection(xdata2, color=[0, 0, 1], linelength=0.05)

# create the events marking the y data points
yevents1 = EventCollection(ydata1, color=[1, 0, 0], linelength=0.05, orientation='vertical')
yevents2 = EventCollection(ydata2, color=[0, 0, 1], linelength=0.05, orientation='vertical')

# add the events to the axis
ax.add_collection(xevents1)
ax.add_collection(xevents2)
ax.add_collection(yevents1)
ax.add_collection(yevents2)

# set the limits
ax.set_xlim([0, 1])
ax.set_ylim([0, 1])

ax.set_title('line plot with data points')

# display the plot
plt.show()
An eventplot showing sequences of events with various line properties. The plot is shown in both horizontal and vertical orientations.

```python
import matplotlib.pyplot as plt
import numpy as np
import matplotlib
matplotlib.rcParams['font.size'] = 8.0

# set the random seed
np.random.seed(0)

# create random data
data1 = np.random.random([6, 50])

# set different colors for each set of positions
colors1 = np.array([[1, 0, 0],
                    [0, 1, 0],
                    [0, 0, 1],
                    [1, 1, 0],
                    [1, 0, 1]],
                   dtype=np.float)
```

# set different line properties for each set of positions
# note that some overlap
lineoffsets1 = np.array([-15, -3, 1, 1.5, 6, 10])
linelengths1 = [5, 2, 1, 1, 3, 1.5]

fig = plt.figure()

# create a horizontal plot
ax1 = fig.add_subplot(221)
ax1.eventplot(data1, colors=colors1, lineoffsets=lineoffsets1,
              linelengths=linelengths1)

# create a vertical plot
ax2 = fig.add_subplot(223)
ax2.eventplot(data1, colors=colors1, lineoffsets=lineoffsets1,
              linelengths=linelengths1, orientation='vertical')

# create another set of random data.
# the gamma distribution is only used for aesthetic purposes
data2 = np.random.gamma(4, size=[60, 50])

# use individual values for the parameters this time
# these values will be used for all data sets (except lineoffsets2, which
# sets the increment between each data set in this usage)
colors2 = [[[0, 0, 0]]
lineoffsets2 = 1
linelengths2 = 1

# create a horizontal plot
ax1 = fig.add_subplot(222)
ax1.eventplot(data2, colors=colors2, lineoffsets=lineoffsets2,
              linelengths=linelengths2)

# create a vertical plot
ax2 = fig.add_subplot(224)
ax2.eventplot(data2, colors=colors2, lineoffsets=lineoffsets2,
              linelengths=linelengths2, orientation='vertical')

plt.show()
import matplotlib.patches as mpatches
import matplotlib.pyplot as plt

styles = mpatches.ArrowStyle.get_styles()

ncol = 2
nrow = (len(styles) + 1) // ncol
figheight = (nrow + 0.5)
fig1 = plt.figure(1, (4.*ncol/1.5, figheight/1.5))

def to_texstring(s):

ax = fig1.add_axes([0, 0, 1, 1], frameon=False, aspect=1.)
ax.set_xlim(0, 4*ncol)
ax.set_ylim(0, figheight)
ax.set_xlim(0, 4*ncol)
ax.set_ylim(0, figheight)
Matplotlib, Release 2.0.2

s = s.replace("<", r"$<$")
s = s.replace(">", r"$>$")
s = s.replace("|", r"$|$")
return s
for i, (stylename, styleclass) in enumerate(sorted(styles.items())):
x = 3.2 + (i//nrow)*4
y = (figheight - 0.7 - i % nrow) # /figheight
p = mpatches.Circle((x, y), 0.2)
ax.add_patch(p)
ax.annotate(to_texstring(stylename), (x, y),
(x - 1.2, y),
#xycoords="figure fraction", textcoords="figure fraction",
ha="right", va="center",
size=fontsize,
arrowprops=dict(arrowstyle=stylename,
patchB=p,
shrinkA=5,
shrinkB=5,
fc="k", ec="k",
connectionstyle="arc3,rad=-0.05",
),
bbox=dict(boxstyle="square", fc="w"))
ax.xaxis.set_visible(False)
ax.yaxis.set_visible(False)

plt.draw()
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

99.70. pylab_examples example code: fancyarrow_demo.py

2505


import matplotlib.pyplot as plt
import matplotlib.transforms as mtransforms
from matplotlib.patches import FancyBboxPatch

# Bbox object around which the fancy box will be drawn.
bb = mtransforms.Bbox([[0.3, 0.4], [0.7, 0.6]])

def draw_bbox(ax, bb):
    # boxstyle=square with pad=0, i.e. bbox itself.
    p_bbox = FancyBboxPatch((bb.xmin, bb.ymin),
                            abs(bb.width), abs(bb.height),
                            boxstyle="square, pad=0.0",
                            ec="k", fc="none", zorder=10.,
                            )
    ax.add_patch(p_bbox)

def test1(ax):
    # a fancy box with round corners. pad=0.1

import matplotlib.pyplot as plt
import matplotlib.transforms as mtransforms
from matplotlib.patches import FancyBboxPatch

# Bbox object around which the fancy box will be drawn.
bb = mtransforms.Bbox([[0.3, 0.4], [0.7, 0.6]])

def draw_bbox(ax, bb):
    # boxstyle=square with pad=0, i.e. bbox itself.
    p_bbox = FancyBboxPatch((bb.xmin, bb.ymin),
                            abs(bb.width), abs(bb.height),
                            boxstyle="square, pad=0.0",
                            ec="k", fc="none", zorder=10.,
                            )
    ax.add_patch(p_bbox)

def test1(ax):
    # a fancy box with round corners. pad=0.1
```python
p_fancy = FancyBboxPatch((bb.xmin, bb.ymin),
    abs(bb.width), abs(bb.height),
    boxstyle="round, pad=0.1",
    fc=(1., .8, 1.),
    ec=(1., 0.5, 1.))

ax.add_patch(p_fancy)

ax.text(0.1, 0.8,
    r'
    boxstyle="round, pad=0.1"
    rounding_size=0.2",
    size=10, transform=ax.transAxes)

# draws control points for the fancy box.
#l = p_fancy.get_path().vertices
#ax.plot(l[:,0], l[:,1], ".")

draw_bbox(ax, bb)

def test2(ax):

    # bbox=round has two optional argument. pad and rounding_size.
    # They can be set during the initialization.
    p_fancy = FancyBboxPatch((bb.xmin, bb.ymin),
        abs(bb.width), abs(bb.height),
        boxstyle="round, pad=0.1",
        fc=(1., .8, 1.),
        ec=(1., 0.5, 1.))

    ax.add_patch(p_fancy)

    # boxstyle and its argument can be later modified with
    # set_boxstyle method. Note that the old attributes are simply
    # forgotten even if the boxstyle name is same.

    p_fancy.set_boxstyle("round, pad=0.1, rounding_size=0.2")
    # or
    #p_fancy.set_boxstyle("round", pad=0.1, rounding_size=0.2)

    ax.text(0.1, 0.8,
        ' boxstyle="round, pad=0.1\\a rounding\\_size=0.2"",
        size=10, transform=ax.transAxes)

    # draws control points for the fancy box.
    #l = p_fancy.get_path().vertices
    #ax.plot(l[:,0], l[:,1], ".")

draw_bbox(ax, bb)

def test3(ax):
```

*99.71. pylab_examples example code: fancybox_demo.py*
# mutation_scale determine overall scale of the mutation,  
# i.e. both pad and rounding_size is scaled according to this  
# value.

```
p_fancy = FancyBboxPatch((bb.xmin, bb.ymin),  
abs(bb.width), abs(bb.height),  
boxstyle="round,pad=0.1",  
mutation_scale=2.,  
fc=(1., .8, 1.),  
ec=(1., 0.5, 1.))
```

```
ax.add_patch(p_fancy)
```

```
ax.text(0.1, 0.8,  
' boxstyle="round,pad=0.1" \n mutation\_scale=2',  
size=10, transform=ax.transAxes)
```

```
# draws control points for the fancy box.  
#l = p_fancy.get_path().vertices  
#ax.plot(l[:,0], l[:,1], ".")

draw_bbox(ax, bb)
```

```python
def test4(ax):
    # When the aspect ratio of the axes is not 1, the fancy box may  
    # not be what you expected (green)

    p_fancy = FancyBboxPatch((bb.xmin, bb.ymin),  
abs(bb.width), abs(bb.height),  
boxstyle="round,pad=0.2",  
fc="none",  
ec=(0., .5, 0.), zorder=4)

    ax.add_patch(p_fancy)

    # You can compensate this by setting the mutation_aspect (pink).
    p_fancy = FancyBboxPatch((bb.xmin, bb.ymin),  
abs(bb.width), abs(bb.height),  
boxstyle="round,pad=0.3",  
mutation_aspect=.5,  
fc=(1., .8, 1.),  
ec=(1., 0.5, 1.))

    ax.add_patch(p_fancy)

    ax.text(0.1, 0.8,  
' boxstyle="round,pad=0.3" \n mutation\_aspect=.5',  
size=10, transform=ax.transAxes)

draw_bbox(ax, bb)
```
```python
def test_all():
    plt.clf()
    ax = plt.subplot(2, 2, 1)
test1(ax)
    ax.set_xlim(0., 1.)
    ax.set_ylim(0., 1.)
    ax.set_title("test1")
    ax.set_aspect(1.)

    ax = plt.subplot(2, 2, 2)
    ax.set_title("test2")
test2(ax)
    ax.set_xlim(0., 1.)
    ax.set_ylim(0., 1.)
    ax.set_aspect(1.)

    ax = plt.subplot(2, 2, 3)
    ax.set_title("test3")
test3(ax)
    ax.set_xlim(0., 1.)
    ax.set_ylim(0., 1.)
    ax.set_aspect(1)

    ax = plt.subplot(2, 2, 4)
    ax.set_title("test4")
test4(ax)
    ax.set_xlim(-0.5, 1.5)
    ax.set_ylim(0., 1.)
    ax.set_aspect(2.)

    plt.draw()
    plt.show()

test_all()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
import matplotlib.patches as mpatch
import matplotlib.pyplot as plt

styles = mpatch.BoxStyle.get_styles()
spacing = 1.2

figheight = (spacing * len(styles) + .5)
fig1 = plt.figure(1, (4/1.5, figheight/1.5))
fontsize = 0.3 * 72

for i, stylen in enumerate(sorted(styles.keys())):
    fig1.text(0.5, (spacing * (float(len(styles)) - i) - 0.5)/figheight, stylen,
    ha="center",
    size=fontsize,
    transform=fig1.transFigure,
    bbox=dict(boxstyle=stylename, fc="w", ec="k")

plt.draw()
plt.show()
ha="center", va="center",
bbox=\text{dict(boxstyle="round",
    ec=(1., 0.5, 0.5),
    fc=(1., 0.8, 0.8),
    )
}

plt.text(0.5, 0.4, "test", size=50, rotation=-30.,
    ha="right", va="top",
bbox=\text{dict(boxstyle="square",
    ec=(1., 0.5, 0.5),
    fc=(1., 0.8, 0.8),
    )
    }\n)

plt.draw()
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

99.74 pylab_examples example code: figimage_demo.py
This illustrates placing images directly in the figure, with no axes.

```python
import numpy as np
import matplotlib
import matplotlib.cm as cm
import matplotlib.pyplot as plt

fig = plt.figure()
Z = np.arange(10000.0)
Z.shape = 100, 100
Z[:, 50:] = 1.
im1 = plt.figimage(Z, xo=50, yo=0, origin='lower')
im2 = plt.figimage(Z, xo=100, yo=100, alpha=.8, origin='lower')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
import numpy as np
import matplotlib.pyplot as plt

fig = plt.figure()
ax1 = fig.add_axes([0.1, 0.1, 0.4, 0.7])
ax2 = fig.add_axes([0.55, 0.1, 0.4, 0.7])

x = np.arange(0.0, 2.0, 0.02)
y1 = np.sin(2*np.pi*x)
y2 = np.exp(-x)
l1, l2 = ax1.plot(x, y1, 'rs-', x, y2, 'go')

y3 = np.sin(4*np.pi*x)
y4 = np.exp(-2*x)
l3, l4 = ax2.plot(x, y3, 'yd-', x, y4, 'k^')

fig.legend((l1, l2), ('Line 1', 'Line 2'), 'upper left')
fig.legend((l3, l4), ('Line 3', 'Line 4'), 'upper right')
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

99.76  

This is a somewhat long figure title
from matplotlib.font_manager import FontProperties
import matplotlib.pyplot as plt
import numpy as np

def f(t):
    s1 = np.cos(2*np.pi*t)
    e1 = np.exp(-t)
    return s1 * e1

t1 = np.arange(0.0, 5.0, 0.1)
t2 = np.arange(0.0, 5.0, 0.02)
t3 = np.arange(0.0, 2.0, 0.01)

plt.subplot(121)
plt.plot(t1, f(t1), 'o', t2, f(t2), '-')
plt.title('subplot 1')
plt.ylabel('Damped oscillation')
plt.suptitle('This is a somewhat long figure title', fontsize=16)

plt.subplot(122)
plt.plot(t3, np.cos(2*np.pi*t3), '--')
plt.xlabel('time (s)')
plt.title('subplot 2')
plt.ylabel('Undamped')

plt.subplots_adjust(left=0.2, wspace=0.8, top=0.8)
plt.show()
99.77  pylab_examples example code: fill_between_demo.py

between y1 and 0
between y1 and 1
between y1 and y2

0.00 0.25 0.50 0.75 1.00 1.25 1.50 1.75 2.00
x
fill between where

Now regions with y2>1 are masked

pylab_examples example code: fill_between_demo.py
import matplotlib.pyplot as plt
import numpy as np

x = np.arange(0.0, 2, 0.01)
y1 = np.sin(2*np.pi*x)
y2 = 1.2*np.sin(4*np.pi*x)

fig, (ax1, ax2, ax3) = plt.subplots(3, 1, sharex=True)
ax1.fill_between(x, 0, y1)
ax1.set_ylabel('between y1 and 0')

ax2.fill_between(x, y1, 1)
ax2.set_ylabel('between y1 and 1')

ax3.fill_between(x, y1, y2)
ax3.set_ylabel('between y1 and y2')
ax3.set_xlabel('x')

# now fill between y1 and y2 where a logical condition is met. Note
# this is different than calling
# fill_between(x[where], y1[where], y2[where])
# because of edge effects over multiple contiguous regions.
fig, (ax, ax1) = plt.subplots(2, 1, sharex=True)
ax.plot(x, y1, x, y2, color='black')
ax.fill_between(x, y1, y2, where=y2 >= y1, facecolor='green', interpolate=True)
ax.fill_between(x, y1, y2, where=y2 <= y1, facecolor='red', interpolate=True)
ax.set_title('fill between where')

# Test support for masked arrays.
y2 = np.ma.masked_greater(y2, 1.0)
ax1.plot(x, y1, x, y2, color='black')
ax1.fill_between(x, y1, y2, where=y2 >= y1, facecolor='green', interpolate=True)
ax1.fill_between(x, y1, y2, where=y2 <= y1, facecolor='red', interpolate=True)
ax1.set_title('Now regions with y2>1 are masked')

# This example illustrates a problem; because of the data
# gridding, there are undesired unfilled triangles at the crossover
# points. A brute-force solution would be to interpolate all
# arrays to a very fine grid before plotting.

# show how to use transforms to create axes spans where a certain condition is satisfied
fig, ax = plt.subplots()
y = np.sin(4*np.pi*x)
ax.plot(x, y, color='black')

# use the data coordinates for the x-axis and the axes coordinates for the y-axis
import matplotlib.transforms as mtransforms
trans = mtransforms.blended_transform_factory(ax.transData, ax.transAxes)
theta = 0.9
ax.axhline(theta, color='green', lw=2, alpha=0.5)
ax.axhline(-theta, color='red', lw=2, alpha=0.5)
ax.fill_between(x, 0, 1, where=y > theta, facecolor='green', alpha=0.5, transform=trans)
ax.fill_between(x, 0, 1, where=y < -theta, facecolor='red', alpha=0.5, transform=trans)

plt.show()
99.78  

pylab_examples example code: fill_betweenx_demo.py

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()

y1 = [0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 2, 2, 2]

ax.plot(y1)
ax.fill_betweenx([-1.0, -0.5, 0.0, 0.5, 1.0], y1, y1 + 1, color='blue', alpha=0.5)

ax.set_xlabel('x')
ax.set_ylabel('y')

plt.show()
```
Copy of fill_between.py but using fill_betweenx() instead.

```python
import matplotlib.mlab as mlab
from matplotlib.pyplot import figure, show
import numpy as np

x = np.arange(0.0, 2, 0.01)
y1 = np.sin(2*np.pi*x)
y2 = 1.2*np.sin(4*np.pi*x)

fig = figure()
ax1 = fig.add_subplot(311)
ax2 = fig.add_subplot(312, sharex=ax1)
ax3 = fig.add_subplot(313, sharex=ax1)

ax1.fill_betweenx(x, 0, y1)
ax1.set_ylabel('(y1, 0)')

ax2.fill_betweenx(x, y1, 1)
ax2.set_ylabel('(y1, 1)')

ax3.fill_betweenx(x, y1, y2)
ax3.set_ylabel('(y1, y2)')
ax3.set_xlabel('x')
```
# now fill between y1 and y2 where a logical condition is met. Note
# this is different than calling
#   fill_between(x[where], y1[where], y2[where])
# because of edge effects over multiple contiguous regions.
fig = figure()
ax = fig.add_subplot(211)
ax.plot(y1, x, y2, x, color='black')
ax.fill_betweenx(x, y1, y2, where=y2 >= y1, facecolor='green')
ax.fill_betweenx(x, y1, y2, where=y2 <= y1, facecolor='red')
ax.set_title('fill between where')

# Test support for masked arrays.
y2 = np.ma.masked_greater(y2, 1.0)
ax1 = fig.add_subplot(212, sharex=ax)
ax1.plot(y1, x, y2, x, color='black')
ax1.fill_betweenx(x, y1, y2, where=y2 >= y1, facecolor='green')
ax1.fill_betweenx(x, y1, y2, where=y2 <= y1, facecolor='red')
ax1.set_title('Now regions with y2 > 1 are masked')

# This example illustrates a problem; because of the data
# gridding, there are undesired unfilled triangles at the crossover
# points. A brute-force solution would be to interpolate all
# arrays to a very fine grid before plotting.

show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
import matplotlib.pyplot as plt
import numpy as np

theta = np.arange(0, 8*np.pi, 0.1)
a = 1
b = .2

for dt in np.arange(0, 2*np.pi, np.pi/2.0):
    x = a*np.cos(theta + dt)*np.exp(b*theta)
y = a*np.sin(theta + dt)*np.exp(b*theta)

    dt = dt + np.pi/4.0

    x2 = a*np.cos(theta + dt)*np.exp(b*theta)
y2 = a*np.sin(theta + dt)*np.exp(b*theta)

    xf = np.concatenate((x, x2[::-1]))
yf = np.concatenate((y, y2[::-1]))

    p1 = plt.fill(xf, yf)
Recursively find all objects that match some criteria

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.text as text

a = np.arange(0, 3, .02)
b = np.arange(0, 3, .02)
c = np.exp(a)
d = c[::-1]

fig, ax = plt.subplots()
plt.plot(a, c, 'k--', a, d, 'k:', a, c + d, 'k')
plt.legend(('Model length', 'Data length', 'Total message length'),
           loc='upper center', shadow=True)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
plt.ylim([-1, 20])
plt.grid(False)
plt.xlabel('Model complexity ---->')
plt.ylabel('Message length ---->')
plt.title('Minimum Message Length')

# match on arbitrary function
def myfunc(x):
    return hasattr(x, 'set_color') and not hasattr(x, 'set_facecolor')

for o in fig.findobj(myfunc):
    o.set_color('blue')

# match on class instances
for o in fig.findobj(text.Text):
    o.set_fontstyle('italic')

plt.show()
labelr = ['00', '10', '20', '30', '40', '50', '60', '70', '80', '90', 'A0', 'B0', 'C0', 'D0', 'E0', 'F0']

if len(sys.argv) > 1:
    fontname = sys.argv[1]
else:
    fontname = os.path.join(matplotlib.get_data_path(),
                           'fonts', 'ttf', 'DejaVuSans.ttf')

font = FT2Font(fontname)
codes = list(font.get_charmap().items())
codes.sort()

# a 16,16 array of character strings
chars = [['' for c in range(16)] for r in range(16)]

colors = [[(0.95, 0.95, 0.95) for c in range(16)] for r in range(16)]

plt.figure(figsize=(8, 4), dpi=120)
for ccode, glyphind in codes:
    if ccode >= 256:
        continue
    r, c = divmod(ccode, 16)
    s = unichr(ccode)
    chars[r][c] = s

lightgrn = (0.5, 0.8, 0.5)
plt.title(fontname)

tab = plt.table(cellText=chars,
                 rowLabels=labelr,
                 colLabels=labelc,
                 rowColours=[lightgrn]*16,
                 colColours=[lightgrn]*16,
                 cellColours=colors,
                 cellLoc='center',
                 loc='upper left')

for key, cell in tab.get_celld().items():
    row, col = key
    if row > 0 and col > 0:
        cell.set_text_props(fontproperties=FontProperties(fname=fontname))

plt.axis('off')
plt.show()
Show how to set custom font properties.

For interactive users, you can also use kwargs to the text command, which requires less typing. See examples/fonts_demo_kw.py

```python
from matplotlib.font_manager import FontProperties
import matplotlib.pyplot as plt

plt.subplot(111, facecolor='w')

font0 = FontProperties()
alignment = {'horizontalalignment': 'center', 'verticalalignment': 'baseline'}
# Show family options

families = ['serif', 'sans-serif', 'cursive', 'fantasy', 'monospace']

font1 = font0.copy()
font1.set_size('large')

# Using alignment and size

plt.text(-0.8, 0.9, 'family', fontproperties=font1, **alignment)
```
yp = [0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2]

for k, family in enumerate(families):
    font = font0.copy()
    font.set_family(family)
    t = plt.text(-0.8, yp[k], family, fontproperties=font,
                  **alignment)

# Show style options
styles = ['normal', 'italic', 'oblique']

for k, style in enumerate(styles):
    font = font0.copy()
    font.set_family('sans-serif')
    font.set_style(style)
    t = plt.text(-0.4, yp[k], style, fontproperties=font,
                 **alignment)

# Show variant options
variants = ['normal', 'small-caps']

for k, variant in enumerate(variants):
    font = font0.copy()
    font.set_family('serif')
    font.set_variant(variant)
    t = plt.text(0.0, yp[k], variant, fontproperties=font,
                 **alignment)

# Show weight options
weights = ['light', 'normal', 'medium', 'semibold', 'bold', 'heavy', 'black']

for k, weight in enumerate(weights):
    font = font0.copy()
    font.set_weight(weight)
    t = plt.text(0.4, yp[k], weight, fontproperties=font,
                 **alignment)

# Show size options
sizes = ['xx-small', 'x-small', 'small', 'medium', 'large',
         'large', 'larger', 'largest']

for k, size in enumerate(sizes):
    font = font0.copy()
    font.set_size(size)
    t = plt.text(0.4, yp[k], size, fontproperties=font,
                 **alignment)
'x-large', 'xx-large']

t = plt.text(0.8, 0.9, 'size', fontproperties=font1,
             **alignment)

for k, size in enumerate(sizes):
    font = font0.copy()
    font.set_size(size)
    t = plt.text(0.8, yp[k], size, fontproperties=font,
                 **alignment)

# Show bold italic

font = font0.copy()
font.set_style('italic')
font.set_weight('bold')
t = plt.text(-0.4, 0.1, 'bold italic', fontproperties=font,
             **alignment)

font = font0.copy()
font.set_style('italic')
font.set_weight('bold')
t = plt.text(-0.4, 0.2, 'bold italic', fontproperties=font,
             **alignment)

font = font0.copy()
font.set_style('italic')
font.set_weight('bold')
t = plt.text(-0.4, 0.3, 'bold italic', fontproperties=font,
             **alignment)

plt.axis([-1, 1, 0, 1])

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Same as fonts_demo using kwargs. If you prefer a more pythonic, OO style of coding, see examples/fonts_demo.py.

```python
from matplotlib.font_manager import FontProperties
import matplotlib.pyplot as plt
import numpy as np

plt.subplot(111, facecolor='w')
alignment = {'horizontalalignment': 'center', 'verticalalignment': 'baseline'}

# Show family options
families = ['serif', 'sans-serif', 'cursive', 'fantasy', 'monospace']

t = plt.text(-0.8, 0.9, 'family', size='large', **alignment)

yp = [0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2]

for k, family in enumerate(families):
    t = plt.text(-0.8, yp[k], family, family=family, **alignment)
```

```python

```
# Show style options

```python
styles = ['normal', 'italic', 'oblique']

for k, style in enumerate(styles):
    t = plt.text(-0.4, yp[k], style, family='sans-serif', style=style, **alignment)
```

# Show variant options

```python
variants = ['normal', 'small-caps']

for k, variant in enumerate(variants):
    t = plt.text(0.0, yp[k], variant, family='serif', variant=variant, **alignment)
```

# Show weight options

```python
weights = ['light', 'normal', 'medium', 'semibold', 'bold', 'heavy', 'black']

for k, weight in enumerate(weights):
    t = plt.text(0.4, yp[k], weight, weight=weight, **alignment)
```

# Show size options

```python
sizes = ['xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large']

for k, size in enumerate(sizes):
    t = plt.text(0.8, yp[k], size, size=size, **alignment)
```

# Show bold italic

```python
x = -0.4

for i in range(3):
    t = plt.text(x, 0.1 + i / 3, 'bold italic', style='italic', weight='bold', size='x-small', **alignment)
```

```python
t = plt.text(x, 0.2, 'bold italic', style='italic', weight='bold', size='medium', **alignment)
```

```python
t = plt.text(x, 0.3, 'bold italic', **alignment)
```
To create plots that share a common axes (visually) you can set the hspace between the subplots close to zero (do not use zero itself). Normally you’ll want to turn off the tick labels on all but one of the axes.

In this example the plots share a common xaxis but you can follow the same logic to supply a common y axis.

```python
import matplotlib.pyplot as plt
import numpy as np
```
t = np.arange(0.0, 2.0, 0.01)
s1 = np.sin(2*np.pi*t)
s2 = np.exp(-t)
s3 = s1*s2

# axes rect in relative 0,1 coords left, bottom, width, height. Turn
# off xtick labels on all but the lower plot

f = plt.figure()
plt.subplots_adjust(hspace=0.001)

ax1 = plt.subplot(311)
ax1.plot(t, s1)
plt.yticks(np.arange(-0.9, 1.0, 0.4))
plt.ylim(-1, 1)

ax2 = plt.subplot(312, sharex=ax1)
ax2.plot(t, s2)
plt.yticks(np.arange(0.1, 1.0, 0.2))
plt.ylim(0, 1)

ax3 = plt.subplot(313, sharex=ax1)
ax3.plot(t, s3)
plt.yticks(np.arange(-0.9, 1.0, 0.4))
plt.ylim(-1, 1)

xticklabels = ax1.get_xticklabels() + ax2.get_xticklabels()
plt.setp(xticklabels, visible=False)

plt.show()
99.85 pylab_examples example code: geo_demo.py
```python
import matplotlib.pyplot as plt

plt.figure()
plt.subplot(111, projection="aitoff")
plt.title("Aitoff")
plt.grid(True)

plt.figure()
plt.subplot(111, projection="hammer")
plt.title("Hammer")
plt.grid(True)

plt.figure()
plt.subplot(111, projection="lambert")
plt.title("Lambert")
plt.grid(True)

plt.figure()
plt.subplot(111, projection="mollweide")
plt.title("Mollweide")
plt.grid(True)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
99.86 pylab_examples example code: ginput_demo.py

[source code]

```python
# -*- noplot -*-

from __future__ import print_function

import matplotlib.pyplot as plt
import numpy as np
t = np.arange(10)
plt.plot(t, np.sin(t))
print("Please click")
x = plt.ginput(3)
print("clicked", x)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

99.87 pylab_examples example code: ginput_manual_clabel.py

[source code]

```python
# -*- noplot -*-

from __future__ import print_function

""
This provides examples of uses of interactive functions, such as ginput, waitforbuttonpress and manual clabel placement.

This script must be run interactively using a backend that has a graphical user interface (for example, using GTKAgg backend, but not PS backend).

See also ginput_demo.py
""

import time
import matplotlib
import numpy as np
import matplotlib.cm as cm
import matplotlib.mlab as mlab
import matplotlib.pyplot as plt

def tellme(s):
    print(s)
    plt.title(s, fontsize=16)
    plt.draw()

# Define a triangle by clicking three points
```
```python
plt.clf()
plt.axis([-1., 1., -1., 1.])
plt.setp(plt.gca(), autoscale_on=False)

tellme('You will define a triangle, click to begin')

plt.waitforbuttonpress()

happy = False
while not happy:
    pts = []
    while len(pts) < 3:
        tellme('Select 3 corners with mouse')
        pts = np.asarray(plt.ginput(3, timeout=-1))
        if len(pts) < 3:
            tellme('Too few points, starting over')
            time.sleep(1)  # Wait a second

    ph = plt.fill(pts[:, 0], pts[:, 1], 'r', lw=2)

tellme('Happy? Key click for yes, mouse click for no')

    happy = plt.waitforbuttonpress()

    # Get rid of fill
    if not happy:
        for p in ph:
            p.remove()

# Now contour according to distance from triangle
# corners - just an example

# Define a nice function of distance from individual pts
def f(x, y, pts):
    z = np.zeros_like(x)
    for p in pts:
        z = z + 1/(np.sqrt((x - p[0])**2 + (y - p[1])**2))
    return 1/z

X, Y = np.meshgrid(np.linspace(-1, 1, 51), np.linspace(-1, 1, 51))
Z = f(X, Y, pts)
CS = plt.contour(X, Y, Z, 20)
tellme('Use mouse to select contour label locations, middle button to finish')
CL = plt.clabel(CS, manual=True)
```

# Now do a zoom
```python
###
tellme('Now do a nested zoom, click to begin')
plt.waitforbuttonpress()

happy = False
while not happy:
    tellme('Select two corners of zoom, middle mouse button to finish')
    pts = np.asarray(plt.ginput(2, timeout=-1))

    happy = len(pts) < 2
    if happy:
        break

    pts = np.sort(pts, axis=0)
    plt.axis(pts.T.ravel())

tellme('All Done!')
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

99.88 pylab_examples example code: gradient_bar.py
```python
from matplotlib.pyplot import figure, show, cm
from numpy import arange
from numpy.random import rand

def gbar(ax, x, y, width=0.5, bottom=0):
    X = [[.6, .6], [.7, .7]]
    for left, top in zip(x, y):
        right = left + width
        ax.imshow(X, interpolation='bicubic', cmap=cm.Blues,
                  extent=(left, right, bottom, top), alpha=1)

fig = figure()
xmin, xmax = xlim = 0, 10
ymin, ymax = ylim = 0, 1
ax = fig.add_subplot(111, xlim=xlim, ylim=ylim,
                     autoscale_on=False)
X = [[.6, .6], [.7, .7]]
ax.imshow(X, interpolation='bicubic', cmap=cm.copper,
          extent=(xmin, xmax, ymin, ymax), alpha=1)

N = 10
x = arange(N) + 0.25
y = rand(N)
gbar(ax, x, y, width=0.7)
ax.set_aspect('auto')
show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
from numpy.random import uniform, seed
from matplotlib.mlab import griddata
import matplotlib.pyplot as plt
import numpy as np

# make up data.
# npts = int (raw_input ('enter # of random points to plot: '))
seed (0)
npts = 200
x = uniform (-2, 2, npts)
y = uniform (-2, 2, npts)
z = x*np.exp(-x**2 - y**2)

# define grid.
xi = np.linspace (-2.1, 2.1, 100)
yi = np.linspace (-2.1, 2.1, 200)

# grid the data.
zi = griddata (x, y, z, xi, yi, interp='linear')

# contour the gridded data, plotting dots at the nonuniform data points.
CS = plt.contour (xi, yi, zi, 15, linewitdhs=0.5, colors='k')

CS = plt.contourf (xi, yi, zi, 15,
                   vmax=abs(zi).max(), vmin=-abs(zi).max())

plt.colorbar ()  # draw colorbar
# plot data points.
import matplotlib.pyplot as plt
from matplotlib.patches import Ellipse, Polygon

fig = plt.figure()
ax1 = fig.add_subplot(131)
ax1.bar(range(1, 5), range(1, 5), color='red', edgecolor='black', hatch='//')
ax1.bar(range(1, 5), [6] * 4, bottom=range(1, 5), color='blue', edgecolor='black', hatch='x')
"Hatching (pattern filled polygons) is supported currently in the PS, PDF, SVG and Agg backends only.

import matplotlib.pyplot as plt
from matplotlib.patches import Ellipse, Polygon

fig = plt.figure()
ax1 = fig.add_subplot(131)
ax1.bar(range(1, 5), range(1, 5), color='red', edgecolor='black', hatch='//')
ax1.bar(range(1, 5), [6] * 4, bottom=range(1, 5), color='blue', edgecolor='black', hatch='x')

plt.scatter(x, y, marker='o', s=5, zorder=10)
plt.xlim(-2, 2)
plt.ylim(-2, 2)
plt.title('griddata test (%d points)' % npts)
plt.show()
ax1.set_xticks([1.5, 2.5, 3.5, 4.5])

ax2 = fig.add_subplot(132)
bars = ax2.bar(range(1, 5), range(1, 5), color='yellow', ecolor='black') +
    ax2.bar(range(1, 5), [6] * 4, bottom=range(1, 5), color='green', ecolor='black')
ax2.set_xticks([1.5, 2.5, 3.5, 4.5])

patterns = ('-', '+', 'x', '\\', '*', 'o', '0', '.')
for bar, pattern in zip(bars, patterns):
    bar.set_hatch(pattern)

tax3 = fig.add_subplot(133)
tax3.fill([1, 3, 3, 1], [1, 1, 2, 2], fill=False, hatch='\\')
tax3.add_patch(Ellipse((4, 1.5), 4, 0.5, fill=False, hatch='*'))
tax3.add_patch(Polygon([[0, 0], [4, 1.1], [6, 2.5], [2, 1.4]], closed=True, fill=False, hatch='/'))
tax3.set_xlim((0, 6))
tax3.set_ylim((0, 2.5))

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

99.91 pylab_examples example code: hexbin_demo.py

hexbin is an axes method or pyplot function that is essentially
**a pcolor of a 2-D histogram with hexagonal cells. It can be much more informative than a scatter plot; in the first subplot below, try substituting 'scatter' for 'hexbin'.

```python
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(0)
n = 100000
x = np.random.standard_normal(n)
y = 2.0 + 3.0 * x + 4.0 * np.random.standard_normal(n)
xmin = x.min()
ymax = x.max()
ymin = y.min()
ymax = y.max()

fig, axs = plt.subplots(ncols=2, sharey=True, figsize=(7, 4))
fig.subplots_adjust(hspace=0.5, left=0.07, right=0.93)
ax = axs[0]
hb = ax.hexbin(x, y, gridsize=50, cmap='inferno')
ax.axis([xmin, xmax, ymin, ymax])
ax.set_title("Hexagon binning")
ax = axs[1]
hb = ax.hexbin(x, y, gridsize=50, bins='log', cmap='inferno')
ax.axis([xmin, xmax, ymin, ymax])
ax.set_title("With a log color scale")
cb = fig.colorbar(hb, ax=ax)
cb.set_label('log10(N)')
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)
"""
hexbin is an axes method or pyplot function that is essentially a
pcolor of a 2-D histogram with hexagonal cells.
"""

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.mlab as mlab

delta = 0.025
x = y = np.arange(-3.0, 3.0, delta)
X, Y = np.meshgrid(x, y)
Z1 = mlab.bivariate_normal(X, Y, 1.0, 1.0, 0.0, 0.0)
Z2 = mlab.bivariate_normal(X, Y, 1.5, 0.5, 1, 1)
Z = Z2 - Z1  # difference of Gaussians
x = X.ravel()
y = Y.ravel()
z = Z.ravel()

if 1:
    # make some points 20 times more common than others, but same mean
xcond = (-1 < x) & (x < 1)
ycond = (-2 < y) & (y < 0)
cond = xcond & ycond
xnew = x[cond]
ynew = y[cond]
znew = z[cond]

for i in range(20):
    x = np.hstack((x, xnew))
y = np.hstack((y, ynew))
z = np.hstack((z, znew))

xmin = x.min()
xmax = x.max()
ymin = y.min()
ymax = y.max()

gridsize = 30

plt.subplot(211)
plt.hexbin(x, y, C=z, gridsize=gridsize, marginals=True, cmap=plt.cm.RdBu,
          vmax=abs(z).max(), vmin=-abs(z).max())
plt.axis([xmin, xmax, ymin, ymax])
ax = plt.colorbar()
ax.set_label('mean value')

plt.subplot(212)
plt.hexbin(x, y, gridsize=gridsize, cmap=plt.cm.Blues_r)
plt.axis([xmin, xmax, ymin, ymax])
ax = plt.colorbar()
ax.set_label('N observations')

plt.show()
import matplotlib.pyplot as plt
import numpy as np
x = np.random.randn(1000)
y = np.random.randn(1000) + 5

# normal distribution center at x=0 and y=5
plt.hist2d(x, y, bins=40)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
from matplotlib.colors import LogNorm
import matplotlib.pyplot as plt
import numpy as np

# normal distribution center at x=0 and y=5
x = np.random.randn(100000)
y = np.random.randn(100000) + 5

plt.hist2d(x, y, bins=40, norm=LogNorm())
plt.colorbar()
plt.show()
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import matplotlib.colors as colors

fig, ax = plt.subplots()
Ntotal = 1000
N, bins, patches = ax.hist(np.random.rand(Ntotal), 20)

# I'll color code by height, but you could use any scalar

# we need to normalize the data to 0..1 for the full range of the colormap
fracs = N.astype(float)/N.max()
norm = colors.Normalize(fracs.min(), fracs.max())

for thisfrac, thispatch in zip(fracs, patches):
    color = cm.viridis(norm(thisfrac))
    thispatch.set_facecolor(color)
plt.show()  

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

99.96  
pylab_examples example code: histogram_percent_demo.py

![Histogram with percentage labels](image)

```python
import matplotlib
from numpy.random import randn
import matplotlib.pyplot as plt
from matplotlib.ticker import FuncFormatter

def to_percent(y, position):
    # Ignore the passed in position. This has the effect of scaling the default
    # tick locations.
    s = str(100 * y)
    # The percent symbol needs escaping in latex
    if matplotlib.rcParams['text.usetex'] is True:
        return s + r'$\%$'
    else:
        return s + '%'

import matplotlib
from numpy.random import randn
import matplotlib.pyplot as plt
from matplotlib.ticker import FuncFormatter

def to_percent(y, position):
    # Ignore the passed in position. This has the effect of scaling the default
    # tick locations.
    s = str(100 * y)
    # The percent symbol needs escaping in latex
    if matplotlib.rcParams['text.usetex'] is True:
        return s + r'$\%$'
    else:
        return s + '%'

import matplotlib
from numpy.random import randn
import matplotlib.pyplot as plt
from matplotlib.ticker import FuncFormatter

def to_percent(y, position):
    # Ignore the passed in position. This has the effect of scaling the default
    # tick locations.
    s = str(100 * y)
    # The percent symbol needs escaping in latex
    if matplotlib.rcParams['text.usetex'] is True:
        return s + r'$\%$'
    else:
        return s + '%'
```

99.96  
pylab_examples example code: histogram_percent_demo.py 2551
```python
x = randn(5000)

# Make a normed histogram. It'll be multiplied by 100 later.
plt.hist(x, bins=50, normed=True)

# Create the formatter using the function to_percent. This multiplies all the
# default labels by 100, making them all percentages
formatter = FuncFormatter(to_percent)

# Set the formatter
plt.gca().yaxis.set_major_formatter(formatter)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

---

**99.97  pylab_examples example code: hyperlinks.py**

[source code]

```python
# -*- noplot -*-

""
This example demonstrates how to set a hyperlinks on various kinds of elements.
""

import numpy as np
import matplotlib.cm as cm
import matplotlib.mlab as mlab
import matplotlib.pyplot as plt

f = plt.figure()
s = plt.scatter([1, 2, 3], [4, 5, 6])
s.set_urls(['http://www.bbc.co.uk/news', 'http://www.google.com', None])
f.canvas.print_figure('scatter.svg')

f = plt.figure()
delta = 0.025
x = y = np.arange(-3.0, 3.0, delta)
X, Y = np.meshgrid(x, y)
Z1 = mlab.bivariate_normal(X, Y, 1.0, 1.0, 0.0, 0.0)
Z2 = mlab.bivariate_normal(X, Y, 1.5, 0.5, 1, 1)
Z = Z2 - Z1  # difference of Gaussians

im = plt.imshow(Z, interpolation='bilinear', cmap=cm.gray,
                 origin='lower', extent=[-3, 3, -3, 3])
im.set_url('http://www.google.com')
```
import numpy as np
import matplotlib.cm as cm
import matplotlib.mlab as mlab
import matplotlib.pyplot as plt
from matplotlib.path import Path
from matplotlib.patches import PathPatch

delta = 0.025
x = y = np.arange(-3.0, 3.0, delta)
X, Y = np.meshgrid(x, y)
Z1 = mlab.bivariate_normal(X, Y, 1.0, 1.0, 0.0, 0.0)
Z2 = mlab.bivariate_normal(X, Y, 1.5, 0.5, 1, 1)
Z = Z2 - Z1  # difference of Gaussians

path = Path([[0, 1], [1, 0], [0, -1], [-1, 0], [0, 1]])
patch = PathPatch(path, facecolor='none')
plt.gca().add_patch(patch)

im = plt.imshow(Z, interpolation='bilinear', cmap=cm.gray,
                 origin='lower', extent=[-3, 3, -3, 3],
                 clip_path=patch, clip_on=True)
im.set_clip_path(patch)

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

99.99  pylab_examples example code: image_demo.py

import numpy as np
import matplotlib.cm as cm
import matplotlib.mlab as mlab
import matplotlib.pyplot as plt

delta = 0.025
x = y = np.arange(-3.0, 3.0, delta)
X, Y = np.meshgrid(x, y)
Z1 = mlab.bivariate_normal(X, Y, 1.0, 1.0, 0.0, 0.0)
```python
Z2 = mlab.bivariate_normal(X, Y, 1.5, 0.5, 1, 1)
Z = Z2 - Z1  # difference of Gaussians

im = plt.imshow(Z, interpolation='bilinear', cmap=cm.RdYlGn,
                 origin='lower', extent=[-3, 3, -3, 3],
                 vmax=abs(Z).max(), vmin=-abs(Z).max())

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

99.100  pylab_examples example code: image_demo2.py

```python
from __future__ import print_function
import matplotlib.pyplot as plt
import numpy as np
import matplotlib.cbook as cbook

w, h = 512, 512

datafile = cbook.get_sample_data('ct.raw.gz', asfileobj=True)
s = datafile.read()
```
A = np.fromstring(s, np.uint16).astype(float)
A *= 1.0 / max(A)
A.shape = w, h

extent = (0, 25, 0, 25)
im = plt.imshow(A, cmap=plt.cm.hot, origin='upper', extent=extent)

markers = [(15.9, 14.5), (16.8, 15)]
x, y = zip(*markers)
plt.plot(x, y, 'o')

plt.title('CT density')
plt.show()
pylab_examples example code: image_interp.py
The same (small) array, interpolated with three different interpolation methods.

The center of the pixel at $A[i,j]$ is plotted at $i+0.5$, $i+0.5$. If you are using interpolation='nearest', the region bounded by $(i,j)$ and $(i+1,j+1)$ will have the same color. If you are using interpolation, the pixel center will have the same color as it does with nearest, but other pixels will be interpolated between the neighboring pixels.

Earlier versions of matplotlib (<0.63) tried to hide the edge effects from you by setting the view limits so that they would not be visible. A recent bugfix in antigrain, and a new implementation in the matplotlib._image module which takes advantage of this fix, no longer makes this necessary. To prevent edge effects, when doing interpolation, the matplotlib._image module now pads the input array with identical pixels around the edge. e.g., if you have a 5x5 array with colors a-y as below

```
a b c d e
f g h i j
k l m n o
p q r s t
u v w x y
```
the _image module creates the padded array,

```
 a a b c d e e
 a a b c d e e
 f f g h i j j
 k k l m n o o
 p p q r s t t
 o u v w x y y
 o u v w x y y
```
does the interpolation/resizing, and then extracts the central region. This allows you to plot the full range of your array w/o edge effects, and for example to layer multiple images of different sizes over one another with different interpolation methods - see examples/layer_images.py. It also implies a performance hit, as this new temporary, padded array must be created. Sophisticated interpolation also implies a performance hit, so if you need maximal performance or have very large images, interpolation='nearest' is suggested.

```
import matplotlib.pyplot as plt
import numpy as np

A = np.random.rand(5, 5)
plt.figure(1)
plt.imshow(A, interpolation='nearest')
plt.grid(True)

plt.figure(2)
plt.imshow(A, interpolation='bilinear')
plt.grid(True)

plt.figure(3)
plt.imshow(A, interpolation='bicubic')
plt.grid(True)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
```python
from copy import copy
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.colors as colors
import matplotlib.mlab as mlab

# compute some interesting data
x0, x1 = -5, 5
y0, y1 = -3, 3
x = np.linspace(x0, x1, 500)

# imshow with masked array input and out-of-range colors.
# The second subplot illustrates the use of BoundaryNorm to get a filled contour effect.
```

Green=low, Red=high, Blue=masked
With BoundaryNorm

```
y = np.linspace(y0, y1, 500)
X, Y = np.meshgrid(x, y)
Z1 = mlab.bivariate_normal(X, Y, 1.0, 1.0, 0.0, 0.0)
Z2 = mlab.bivariate_normal(X, Y, 1.5, 0.5, 1, 1)
Z = 10*(Z2 - Z1)  # difference of Gaussians

# Set up a colormap:
# use copy so that we do not mutate the global colormap instance
palette = copy(plt.cm.gray)
palette.set_over('r', 1.0)
palette.set_under('g', 1.0)
palette.set_bad('b', 1.0)
# Alternatively, we could use
# palette.set_bad(alpha = 0.0)
# to make the bad region transparent. This is the default.
# If you comment out all the palette.set* lines, you will see
# all the defaults; under and over will be colored with the
# first and last colors in the palette, respectively.
Zm = np.ma.masked_where(Z > 1.2, Z)

# By setting vmin and vmax in the norm, we establish the
# range to which the regular palette color scale is applied.
# Anything above that range is colored based on palette.set_over, etc.

# set up the axes
fig, (ax1, ax2) = plt.subplots(nrows=2, figsize=(6, 5.4))

# plot using 'continuous' color map
im = ax1.imshow(Zm, interpolation='bilinear',
                 cmap=palette,
                 norm=colors.Normalize(vmin=-1.0, vmax=1.0),
                 aspect='auto',
                 origin='lower',
                 extent=[x0, x1, y0, y1])
ax1.set_title('Green=low, Red=high, Blue=masked')
cbar = fig.colorbar(im, extend='both', shrink=0.9, ax=ax1)
cbar.set_label('uniform')
for ticklabel in ax1.xaxis.get_ticklabels():
ticklabel.set_visible(False)

# Plot using a small number of colors, with unevenly spaced boundaries.
im = ax2.imshow(Zm, interpolation='nearest',
                 cmap=palette,
                 norm=colors.BoundaryNorm([-1, -0.5, -0.2, 0, 0.2, 0.5, 1],
                                          ncolors=palette.N),
                 aspect='auto',
                 origin='lower',
                 extent=[x0, x1, y0, y1])
ax2.set_title('With BoundaryNorm')
cbar = fig.colorbar(im, extend='both', spacing='proportional',
                                          shrink=0.9, ax=ax2)
cbar.set_label('proportional')
fig.suptitle('imshow, with out-of-range and masked data')
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

99.103 pylab_examples example code: image_nonuniform.py

```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.image import NonUniformImage
from matplotlib import cm

interp = 'nearest'

# Linear x array for cell centers:
x = np.linspace(-4, 4, 9)
```

This illustrates the NonUniformImage class. It is not available via an Axes method but it is easily added to an Axes instance as shown here.
# Highly nonlinear x array:
x2 = x**3

y = np.linspace(-4, 4, 9)

z = np.sqrt(x[np.newaxis, :]**2 + y[:, np.newaxis]**2)

fig, axs = plt.subplots(nrows=2, ncols=2)
fig.subplots_adjust(bottom=0.07, hspace=0.3)
fig.suptitle('NonUniformImage class', fontsize='large')
ax = axs[0, 0]
im = NonUniformImage(ax, interpolation='interp', extent=(-4, 4, -4, 4),
cmap=cm.Purples)
im.set_data(x, y, z)
ax.images.append(im)
ax.set_xlim(-4, 4)
ax.set_ylim(-4, 4)
ax.set_title('interp')

ax = axs[0, 1]
im = NonUniformImage(ax, interpolation='interp', extent=(-64, 64, -4, 4),
cmap=cm.Purples)
im.set_data(x2, y, z)
ax.images.append(im)
ax.set_xlim(-64, 64)
ax.set_ylim(-4, 4)
ax.set_title('interp')

interp = 'bilinear'

ax = axs[1, 0]
im = NonUniformImage(ax, interpolation='interp', extent=(-4, 4, -4, 4),
cmap=cm.Purples)
im.set_data(x, y, z)
ax.images.append(im)
ax.set_xlim(-4, 4)
ax.set_ylim(-4, 4)
ax.set_title(interp)

ax = axs[1, 1]
im = NonUniformImage(ax, interpolation='interp', extent=(-64, 64, -4, 4),
cmap=cm.Purples)
im.set_data(x2, y, z)
ax.images.append(im)
ax.set_xlim(-64, 64)
ax.set_ylim(-4, 4)
ax.set_title(interp)

plt.show()
You can specify whether images should be plotted with the array origin $x[0,0]$ in the upper left or upper right by using the origin parameter. You can also control the default by setting `image.origin` in your matplotlibrc file; see http://matplotlib.org/matplotlibrc

```python
import matplotlib.pyplot as plt
import numpy as np

x = np.arange(120)
x.shape = (10, 12)

interp = 'bilinear'
fig, axs = plt.subplots(nrows=2, sharex=True, figsize=(3, 5))
axs[0].set_title('blue should be up')
axs[0].imshow(x, origin='upper', interpolation=interp)

axs[1].set_title('blue should be down')
axs[1].imshow(x, origin='lower', interpolation=interp)
```
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

99.105  pylab_examples example code: image_slices_viewer.py

```python
from __future__ import print_function
import numpy
import matplotlib.pyplot as plt

class IndexTracker(object):
    def __init__(self, ax, X):
        self.ax = ax
        ax.set_title('use scroll wheel to navigate images

        self.X = X
        rows, cols, self.slices = X.shape
        self.ind = self.slices//2

        self.im = ax.imshow(self.X[:, :, self.ind])
        self.update()
```

99.105.  pylab_examples example code: image_slices_viewer.py  2565
```python
def onscroll(self, event):
    print("%s %s" % (event.button, event.step))
    if event.button == 'up':
        self.ind = numpy.clip(self.ind + 1, 0, self.slices - 1)
    else:
        self.ind = numpy.clip(self.ind - 1, 0, self.slices - 1)
    self.update()

def update(self):
    self.im.set_data(self.X[:, :, self.ind])
    ax.set_ylabel('slice %s' % self.ind)
    self.im.axes.figure.canvas.draw()

fig, ax = plt.subplots(1, 1)

X = numpy.random.rand(20, 20, 40)

tracker = IndexTracker(ax, X)

fig.canvas.mpl_connect('scroll_event', tracker.onscroll)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
99.106 pylab_examples example code: interp_demo.py

```python
import matplotlib.pyplot as plt
from numpy import pi, sin, linspace
from matplotlib.mlab import stineman_interp

x = linspace(0, 2*pi, 20)
y = sin(x)
yp = None
xi = linspace(x[0], x[-1], 100)
yi = stineman_interp(xi, x, y, yp)

fig, ax = plt.subplots()
ax.plot(x, y, 'o', xi, yi, '.')
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
You can use decreasing axes by flipping the normal order of the axis limits.

```python
import matplotlib.pyplot as plt
import numpy as np

t = np.arange(0.01, 5.0, 0.01)
s = np.exp(-t)
plt.plot(t, s)
plt.xlim(5, 0)  # decreasing time
plt.xlabel('decreasing time (s)')
plt.ylabel('voltage (mV)')
plt.title('Should be growing...')
plt.grid(True)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Layer images above one another using alpha blending

from __future__ import division
import matplotlib.pyplot as plt
import numpy as np

def func3(x, y):
    return (1 - x/2 + x**5 + y**3) * np.exp(-(x**2 + y**2))

# make these smaller to increase the resolution
dx, dy = 0.05, 0.05
x = np.arange(-3.0, 3.0, dx)
y = np.arange(-3.0, 3.0, dy)
X, Y = np.meshgrid(x, y)

# when layering multiple images, the images need to have the same
# extent. This does not mean they need to have the same shape, but
# they both need to render to the same coordinate system determined by
# xmin, xmax, ymin, ymax. Note if you use different interpolations
# for the images their apparent extent could be different due to
# interpolation edge effects

xmin, xmax, ymin, ymax = np.amin(x), np.amax(x), np.amin(y), np.amax(y)
extent = xmin, xmax, ymin, ymax
fig = plt.figure(frameon=False)

Z1 = np.array((\[0, 1\]^4 + [1, 0]^4)^4)
Z1.shape = (8, 8)  # chessboard
im1 = plt.imshow(Z1, cmap=plt.cm.gray, interpolation='nearest',
    extent=extent)

Z2 = func3(X, Y)
im2 = plt.imshow(Z2, cmap=plt.cm.viridis, alpha=.9, interpolation='bilinear',
    extent=extent)

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

99.109 pylab_examples example code: leftventricle_bulleye.py
This example demonstrates how to create the 17 segment model for the left ventricle recommended by the American Heart Association (AHA).

```python
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt

def bullseye_plot(ax, data, segBold=None, cmap=None, norm=None):
    
    Bullseye representation for the left ventricle.

    Parameters
    ----------
    ax : axes
    data : list of int and float
        The intensity values for each of the 17 segments
    segBold : list of int, optional
        A list with the segments to highlight
    cmap : ColorMap or None, optional
        Optional argument to set the desired colormap
    norm : Normalize or None, optional
        Optional argument to normalize data into the [0.0, 1.0] range

    Notes
    ------
    This function create the 17 segment model for the left ventricle according to the American Heart Association (AHA) [1].

    References
    ----------
    .. [1] M. D. Cerqueira, N. J. Weissman, V. Dilsizian, A. K. Jacobs, 
        S. Kaul, W. K. Laskey, D. J. Pennell, J. A. Rumberger, T. Ryan, 
        and M. S. Verani, "Standardized myocardial segmentation and 
        nomenclature for tomographic imaging of the heart", 

    if segBold is None:
        segBold = []

    linewidth = 2
    data = np.array(data).ravel()

    if cmap is None:
        cmap = plt.cm.viridis

    if norm is None:
        norm = mpl.colors.Normalize(vmin=data.min(), vmax=data.max())

    theta = np.linspace(0, 2*np.pi, 768)
```

99.109. pylab_examples example code: leftventricle_bulleye.py 2571
```python
r = np.linspace(0.2, 1, 4)

# Create the bound for the segment 17
for i in range(r.shape[0]):
    ax.plot(theta, np.repeat(r[i], theta.shape), '-k', lw=linewidth)

# Create the bounds for the segments 1-12
for i in range(6):
    theta_i = i*60*np.pi/180
    ax.plot([theta_i, theta_i], [r[1], 1], '-k', lw=linewidth)

# Create the bounds for the segments 13-16
for i in range(4):
    theta_i = i*90*np.pi/180 - 45*np.pi/180
    ax.plot([theta_i, theta_i], [r[0], r[1]], '-k', lw=linewidth)

# Fill the segments 1-6
r0 = r[2:4]
r0 = np.repeat(r0[:, np.newaxis], 128, axis=1).T
for i in range(6):
    # First segment start at 60 degrees
    theta0 = theta[i*128:i*128+128] + 60*np.pi/180
    theta0 = np.repeat(theta0[:, np.newaxis], 2, axis=1)
    z = np.ones((128, 2))*data[i]
    ax.pcolormesh(theta0, r0, z, cmap=cmap, norm=norm)
    if i+1 in segBold:
        ax.plot(theta0, r0, '-k', lw=linewidth+2)
        ax.plot(theta0[0], [r[2], r[3]], '-k', lw=linewidth+1)
        ax.plot(theta0[-1], [r[2], r[3]], '-k', lw=linewidth+1)

# Fill the segments 7-12
r0 = r[1:3]
r0 = np.repeat(r0[:, np.newaxis], 128, axis=1).T
for i in range(6):
    # First segment start at 60 degrees
    theta0 = theta[i*128:i*128+128] + 60*np.pi/180
    theta0 = np.repeat(theta0[:, np.newaxis], 2, axis=1)
    z = np.ones((128, 2))*data[i+6]
    ax.pcolormesh(theta0, r0, z, cmap=cmap, norm=norm)
    if i+7 in segBold:
        ax.plot(theta0, r0, '-k', lw=linewidth+2)
        ax.plot(theta0[0], [r[2], r[3]], '-k', lw=linewidth+1)
        ax.plot(theta0[-1], [r[2], r[3]], '-k', lw=linewidth+1)

# Fill the segments 13-16
r0 = r[0:2]
r0 = np.repeat(r0[:, np.newaxis], 192, axis=1).T
for i in range(4):
    # First segment start at 45 degrees
    theta0 = theta[i*192:i*192+192] + 45*np.pi/180
    theta0 = np.repeat(theta0[:, np.newaxis], 2, axis=1)
    z = np.ones((192, 2))*data[i+12]
    ax.pcolormesh(theta0, r0, z, cmap=cmap, norm=norm)
```

Chapter 99. `pylab_examples` Examples
if i+13 in segBold:
    ax.plot(theta0, r0, '-k', lw=linewidth+2)
    ax.plot(theta0[0], [r[0], r[1]], '-k', lw=linewidth+1)
    ax.plot(theta0[-1], [r[0], r[1]], '-k', lw=linewidth+1)

# Fill the segments 17
if data.size == 17:
    r0 = np.array([[0, r[0]])
    r0 = np.repeat(r0[:, np.newaxis], theta.size, axis=1).T
    theta0 = np.repeat(theta[:, np.newaxis], 2, axis=1)
    z = np.ones((theta.size, 2)) * data[16]
    ax.pcolormesh(theta0, r0, z, cmap=cmap, norm=norm)
    if 17 in segBold:
        ax.plot(theta0, r0, '-k', lw=linewidth+2)

ax.set_ylim([0, 1])
ax.set_yticklabels([])
ax.set_xticklabels([])

# Create the fake data
data = np.array(range(17)) + 1

# Make a figure and axes with dimensions as desired.
fig, ax = plt.subplots(figsize=(12, 8), nrows=1, ncols=3,
                      subplot_kw=dict(projection='polar'))
fig.canvas.set_window_title('Left Ventricle Bulls Eyes (AHA)')

# Create the axis for the colorbars
axl = fig.add_axes([0.14, 0.15, 0.2, 0.05])
axl2 = fig.add_axes([0.41, 0.15, 0.2, 0.05])
axl3 = fig.add_axes([0.69, 0.15, 0.2, 0.05])

# Set the colormap and norm to correspond to the data for which
# the colorbar will be used.
cmap = mpl.cm.viridis
norm = mpl.colors.Normalize(vmin=1, vmax=17)

# ColorbarBase derives from ScalarMappable and puts a colorbar
# in a specified axes, so it has everything needed for a
# standalone colorbar. There are many more kwargs, but the
# following gives a basic continuous colorbar with ticks
# and labels.
cb1 = mpl.colorbar.ColorbarBase(axl, cmap=cmap, norm=norm,
                                  orientation='horizontal')

# Set the colormap and norm to correspond to the data for which
# the colorbar will be used.
cmap2 = mpl.cm.cool
**Matplotlib, Release 2.0.2**

```python
norm2 = mpl.colors.Normalize(vmin=1, vmax=17)

# ColorbarBase derives from ScalarMappable and puts a colorbar
# in a specified axes, so it has everything needed for a
# standalone colorbar. There are many more kwargs, but the
# following gives a basic continuous colorbar with ticks
# and labels.

cb2 = mpl.colorbar.ColorbarBase(axl2, cmap=cmap2, norm=norm2,
                                 orientation='horizontal')

cb2.set_label('Some other units')

# The second example illustrates the use of a ListedColormap, a
# BoundaryNorm, and extended ends to show the "over" and "under"
# value colors.

cmap3 = mpl.colors.ListedColormap(['r', 'g', 'b', 'c'])
cmap3.set_over('0.35')
cmap3.set_under('0.75')

# If a ListedColormap is used, the length of the bounds array must be
# one greater than the length of the color list. The bounds must be
# monotonically increasing.

bounds = [2, 3, 7, 9, 15]
norm3 = mpl.colors.BoundaryNorm(bounds, cmap3.N)

cb3 = mpl.colorbar.ColorbarBase(axl3, cmap=cmap3, norm=norm3,
                                 extend='both',
                                 ticks=bounds,
                                 orientation='horizontal')

cb3.set_label('Discrete intervals, some other units')

# Create the 17 segment model
bullseye_plot(ax[0], data, cmap=cmap, norm=norm)
ax[0].set_title('Bulls Eye (AHA)')

bullseye_plot(ax[1], data, cmap=cmap2, norm=norm2)
ax[1].set_title('Bulls Eye (AHA)')

bullseye_plot(ax[2], data, segBold=[3, 5, 6, 11, 12, 16],
              cmap=cmap3, norm=norm3)
ax[2].set_title('Segments [3,5,6,11,12,16] in bold')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)
# Make a legend for specific lines.
import matplotlib.pyplot as plt
import numpy as np
t1 = np.arange(0.0, 2.0, 0.1)
t2 = np.arange(0.0, 2.0, 0.01)

# note that plot returns a list of lines. The "l1, = plot" usage
# extracts the first element of the list into l1 using tuple
# unpacking. So l1 is a Line2D instance, not a sequence of lines
l1, = plt.plot(t2, np.exp(-t2))
l2, l3 = plt.plot(t2, np.sin(2 * np.pi * t2), '--o', t1, np.log(1 + t1), '.')
l4, = plt.plot(t2, np.exp(-t2) * np.sin(2 * np.pi * t2), 's-.'

plt.legend((l2, l4), ('oscillatory', 'damped'), loc='upper right', shadow=True)
plt.xlabel('time')
plt.ylabel('volts')
plt.title('Damped oscillation')
plt.show()
```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 1)

# Plot the lines y=x**n for n=1..4.
ax = plt.subplot(2, 1, 1)
for n in range(1, 5):
    plt.plot(x, x**n, label="n={0}".format(n))
plt.legend(loc="upper left", bbox_to_anchor=[0, 1],
    ncol=2, shadow=True, title="Legend", fancybox=True)
ax.get_legend().get_title().set_color("red")

# Demonstrate some more complex labels.
ax = plt.subplot(2, 1, 2)
plt.plot(x, x**2, label="multi\nline")
half_pi = np.linspace(0, np.pi / 2)
plt.plot(np.sin(half_pi), np.cos(half_pi), label=r"\$\frac{1}{2}\pi$"
    )
plt.plot(x, 2**(x**2), label="$2^{x^2}$")
plt.legend(shadow=True, fancybox=True)

plt.show()
```
import matplotlib.pyplot as plt

fig, axes = plt.subplots(3, 1)
top_ax, middle_ax, bottom_ax = axes

top_ax.bar([0, 1, 2], [0.2, 0.3, 0.1], width=0.4, label="Bar 1", align="center")
top_ax.bar([0.5, 1.5, 2.5], [0.3, 0.2, 0.2], color="red", width=0.4, label="Bar 2", align="center")
top_ax.legend()
middle_ax.errorbar([0, 1, 2], [2, 3, 1], xerr=0.4, fmt="s", label="test 1")
middle_ax.errorbar([0, 1, 2], [3, 2, 4], yerr=0.3, fmt="o", label="test 2")
middle_ax.errorbar([0, 1, 2], [1, 1, 3], xerr=0.4, yerr=0.3, fmt="^", label="test 3")
middle_ax.legend()

bottom_ax.stem([0.3, 1.5, 2.7], [1, 3.6, 2.7], label="stem test")
bottom_ax.legend()
plt.subplots_adjust(hspace=0.7)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

99.113 pylab_examples example code: legend_demo5.py

```python
from __future__ import
    absolute_import, division,
    print_function, unicode_literals)
import six
from matplotlib import
    pyplot as plt
import numpy as np
from matplotlib.legend_handler import
    HandlerLineCollection
import matplotlib.collections as mcol
from matplotlib.lines import
    Line2D

class HandlerDashedLines(HandlerLineCollection):
    """
    Custom Handler for LineCollection instances.
    """
```

```python
def create_artists(self, legend, orig_handle, xdescent, ydescent, width, height, fontsize, trans):
    # figure out how many lines there are
    numlines = len(orig_handle.get_segments())
    xdata, xdata_marker = self.get_xdata(legend, xdescent, ydescent, width, height, fontsize)
    lelines = []
    # divide the vertical space where the lines will go
    # into equal parts based on the number of lines
    ydata = ((height) / (numlines + 1)) * np.ones(xdata.shape, float)
    # for each line, create the line at the proper location
    # and set the dash pattern
    for i in range(numlines):
        leline = Line2D(xdata, ydata * (numlines - i) - ydescent)
        self.update_prop(legline, orig_handle, legend)
        # set color, dash pattern, and linewidth to that
        # of the lines in linecollection
        try:
            color = orig_handle.get_colors()[i]
        except IndexError:
            color = orig_handle.get_colors()[0]
        try:
            dashes = orig_handle.get_dashes()[i]
        except IndexError:
            dashes = orig_handle.get_dashes()[0]
        try:
            lw = orig_handle.get_linewidths()[i]
        except IndexError:
            lw = orig_handle.get_linewidths()[0]
        if dashes[0] is not None:
            leline.set_dashes(dashes[1])
            leline.set_color(color)
            leline.set_transform(trans)
            leline.set_linewidth(lw)
            lelines.append(legline)
    return lelines

x = np.linspace(0, 5, 100)

plt.figure()
colors = plt.rcParams['axes.prop_cycle'].by_key()['color'][:5]
styles = ['solid', 'dashed', 'dashed', 'dashed', 'solid']
lines = []
for i, color, style in zip(range(5), colors, styles):
    plt.plot(x, np.sin(x) - .1 * i, c=color, ls=style)

# make proxy artists
# make list of one line -- doesn't matter what the coordinates are
line = [[(0, 0)]]
# set up the proxy artist
lc = mcol.LineCollection(5 * line, linestyles=styles, colors=colors)
# create the legend
```

99.113. pylab_examples example code: legend_demo5.py
```python
import matplotlib.pyplot as plt
from matplotlib.collections import LineCollection
from matplotlib import colors as mcolors
import numpy as np

# In order to efficiently plot many lines in a single set of axes,
# Matplotlib has the ability to add the lines all at once. Here is a
# simple example showing how it is done.

x = np.arange(100)
# Here are many sets of y to plot vs x
ys = x[:50, np.newaxis] + x[np.newaxis, :]
```
```python
segs = np.zeros((50, 100, 2), float)
segs[:, :, 1] = ys
segs[:, :, 0] = x

# Mask some values to test masked array support:
segs = np.ma.masked_where((segs > 50) & (segs < 60), segs)

# We need to set the plot limits.
ax = plt.axes()
ax.set_xlim(x.min(), x.max())
ax.set_ylim(ys.min(), ys.max())

# colors is sequence of rgba tuples
# linestyle is a string or dash tuple. Legal string values are
#     solid|dashed|dashdot|dotted. The dash tuple is (offset, onoffseq)
#     where onoffseq is an even length tuple of on and off ink in points.
#     If linestyle is omitted, 'solid' is used
# See matplotlib.collections.LineCollection for more information
colors = [mcolors.to_rgba(c)
          for c in plt.rcParams['axes.prop_cycle'].by_key()['color']]  

line_segments = LineCollection(segs, linewidths=(0.5, 1, 1.5, 2),
                                colors=colors, linestyle='solid')
ax.add_collection(line_segments)
ax.set_title('Line collection with masked arrays')
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
import matplotlib.pyplot as plt
import numpy as np
from matplotlib.collections import LineCollection

# In order to efficiently plot many lines in a single set of axes,
# Matplotlib has the ability to add the lines all at once. Here is a
# simple example showing how it is done.

N = 50
x = np.arange(N)
# Here are many sets of y to plot vs x
ys = [x + i for i in x]

# We need to set the plot limits, they will not autoscale
ax = plt.axes()
ax.set_xlim((np.amin(x), np.amax(x)))
ax.set_ylim((np.amin(np.amin(ys)), np.amax(np.amax(ys))))

# colors is sequence of rgba tuples
# linestyle is a string or dash tuple. Legal string values are
# solid|dashed|dashdot|dotted. The dash tuple is (offset, onoffseq)
# where onoffseq is an even length tuple of on and off ink in points.
If linestyle is omitted, 'solid' is used

# See matplotlib.collections.LineCollection for more information

# Make a sequence of x,y pairs
line_segments = LineCollection([list(zip(x, y)) for y in ys],
                              linewidths=(0.5, 1, 1.5, 2),
                              linestyles='solid')

line_segments.set_array(x)
ax.add_collection(line_segments)
fig = plt.gcf()
axcb = fig.colorbar(line_segments)
axcb.set_label('Line Number')
ax.set_title('Line Collection with mapped colors')
plt.sci(line_segments)  # This allows interactive changing of the colormap.
plt.show()
import matplotlib.pyplot as plt
import matplotlib.cbook as cbook
import matplotlib.dates as mdates
from matplotlib.dates import bytespdate2num

datafile = cbook.get_sample_data('msft.csv', asfileobj=False)
print('loading', datafile)

dates, closes = np.loadtxt(datafile, delimiter=',',
                           converters={0: bytespdate2num('%d-%b-%y')},
                           skiprows=1, usecols=(0, 2), unpack=True)

fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot_date(dates, closes, '-')
fig.autofmt_xdate()
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

99.117 pylab_examples example code: loadrec.py
from __future__ import print_function
from matplotlib import mlab
import matplotlib.pyplot as plt
import matplotlib.cbook as cbook

datafile = cbook.get_sample_data('msft.csv', asfileobj=False)
print('loading', datafile)
a = mlab.csv2rec(datafile)
a.sort()
print(a.dtype)

fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(a.date, a.adj_close, '-')
fig.autofmt_xdate()

# if you have xlwt installed, you can output excel
try:
    import mpl_toolkits.exceltools as exceltools
    exceltools.rec2excel(a, 'test.xls')
except ImportError:
    pass
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
```python
import matplotlib.pyplot as plt
import numpy as np

data = ((3, 1000), (10, 3), (100, 30), (500, 800), (50, 1))

plt.xlabel("FOO")
plt.ylabel("FOO")
plt.title("Testing")
plt.yscale('log')

dim = len(data[0])
w = 0.75
dimw = w / dim

x = np.arange(len(data))
for i in range(len(data[0])):
    y = [d[i] for d in data]
    b = plt.bar(x + i * dimw, y, dimw, bottom=0.001)

plt.xticks(x + dimw / 2, map(str, x))

plt.show()
```
import numpy as np
import matplotlib.pyplot as plt

plt.subplots_adjust(hspace=0.4)
t = np.arange(0.01, 20.0, 0.01)

# log y axis
plt.subplot(221)
plt.semilogy(t, np.exp(-t/5.0))
plt.title('semilogy')
plt.grid(True)

# log x axis
plt.subplot(222)
plt.semilogx(t, np.sin(2*np.pi*t))
plt.title('semilogx')
plt.grid(True)

# log x and y axis
plt.loglog(t, np.sin(2*np.pi*t)*np.exp(-t/5.0))
plt.title('loglog base 2 on x')
plt.grid(True)

plt.show()
plt.subplot(223)
plt.loglog(t, 20*np.exp(-t/10.0), basex=2)
plt.grid(True)
plt.title('loglog base 2 on x')

# with errorbars: clip non-positive values
ax = plt.subplot(224)
ax.set_yscale("log", nonposy='clip')
ax.set_xscale("log", nonposx='clip')

x = 10.0**np.linspace(0.0, 2.0, 20)
y = x**2.0
plt.errorbar(x, y, xerr=0.1*x, yerr=5.0 + 0.75*y)
al.set_xlim(ymin=0.1)
al.set_title('Errorbars go negative')

plt.show()
import matplotlib.pyplot as plt
import numpy as np

dt = 0.01
t = np.arange(dt, 20.0, dt)

plt.semilogx(t, np.exp(-t/5.0))
plt.grid(True)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

99.121 pylab_examples example code: logo.py

# This file generates an old version of the matplotlib logo
from __future__ import print_function
# Above import not necessary for Python 3 onwards. Recommend taking this
# out in examples in the future, since we should all move to Python 3.
import matplotlib.pyplot as plt
import numpy as np
import matplotlib.cbook as cbook

# convert data to mV
datafile = cbook.get_sample_data('membrane.dat', asfileobj=False)
print('loading', datafile)

x = 1000 * 0.1 * np.fromstring(open(datafile, 'rb').read(), np.float32)
# 0.0005 is the sample interval
t = 0.0005 * np.arange(len(x))
plt.figure(1, figsize=(7, 1), dpi=100)
ax = plt.subplot(111, facecolor='y')
plt.plot(t, x)
plt.text(0.5, 0.5, 'matplotlib', color='r',
        fontsize=40, fontname=['Courier', 'DejaVu Sans Mono'],
        horizontalalignment='center',
        verticalalignment='center',
        transform=ax.transAxes,
    )
plt.axis([1, 1.72, -60, 10])
plt.gca().set_xticklabels([])
plt.gca().set_yticklabels([])

99.121. pylab_examples example code: logo.py 2589
"""
Demonstrate how to use major and minor tickers.

The two relevant userland classes are Locators and Formatters. Locators determine where the ticks are and formatters control the formatting of ticks.

Minor ticks are off by default (NullLocator and NullFormatter). You can turn minor ticks on w/o labels by setting the minor locator. You can also turn labeling on for the minor ticker by setting the minor formatter

Make a plot with major ticks that are multiples of 20 and minor ticks that are multiples of 5. Label major ticks with %d formatting but don't label minor ticks
"""
The MultipleLocator ticker class is used to place ticks on multiples of some base. The FormatStrFormatter uses a string format string (e.g., \%d or \%1.2f or \%1.1f cm') to format the tick.

The pyplot interface grid command changes the grid settings of the major ticks of the y and y axis together. If you want to control the grid of the minor ticks for a given axis, use for example:

```python
ax.xaxis.grid(True, which='minor')
```

Note, you should not use the same locator between different Axis because the locator stores references to the Axis data and view limits.

```python
import matplotlib.pyplot as plt
import numpy as np
from matplotlib.ticker import MultipleLocator, FormatStrFormatter

majorLocator = MultipleLocator(20)
majorFormatter = FormatStrFormatter('%%d')
minorLocator = MultipleLocator(5)

t = np.arange(0.0, 100.0, 0.1)
s = np.sin(0.1*np.pi*t)*np.exp(-t*0.01)

fig, ax = plt.subplots()
plt.plot(t, s)
ax.xaxis.set_major_locator(majorLocator)
ax.xaxis.set_major_formatter(majorFormatter)

# for the minor ticks, use no labels; default NullFormatter
ax.xaxis.set_minor_locator(minorLocator)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Automatic tick selection for major and minor ticks.

Use interactive pan and zoom to see how the tick intervals change. There will be either 4 or 5 minor tick intervals per major interval, depending on the major interval.

```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.ticker import AutoMinorLocator

# One can supply an argument to AutoMinorLocator to
# specify a fixed number of minor intervals per major interval, e.g.:
# minorLocator = AutoMinorLocator(2)
# would lead to a single minor tick between major ticks.

minorLocator = AutoMinorLocator()

t = np.arange(0.0, 100.0, 0.01)
s = np.sin(2*np.pi*t)*np.exp(-t*0.01)
```
```python
fig, ax = plt.subplots()
plt.plot(t, s)

ax.xaxis.set_minor_locator(minorLocator)

plt.tick_params(which='both', width=2)
plt.tick_params(which='major', length=7)
plt.tick_params(which='minor', length=4, color='r')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

**99.124  pylab_examples example code: manual_axis.py**

```

""
The techniques here are no longer required with the new support for spines in matplotlib -- see http://matplotlib.org/examples/pylab_examples/spine_placement_demo.html.

This example should be considered deprecated and is left just for demo

""
```
purposes for folks wanting to make a pseudo-axis

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.lines as lines

def make_xaxis(ax, yloc, offset=0.05, **props):
    xmin, xmax = ax.get_xlim()
    locs = [loc for loc in ax.xaxis.get_majorticklocs()
            if loc >= xmin and loc <= xmax]
    tickline, = ax.plot(locs, [yloc]*len(locs), linestyle='',
                        marker=lines.TICKDOWN, **props)
    axline, = ax.plot([xmin, xmax], [yloc, yloc], **props)
    tickline.set_clip_on(False)
    axline.set_clip_on(False)
    for loc in locs:
        ax.text(loc, yloc - offset, '%1.1f' % loc,
                horizontalalignment='center',
                verticalalignment='top')

def make_yaxis(ax, xloc=0, offset=0.05, **props):
    ymin, ymax = ax.get_ylim()
    locs = [loc for loc in ax.yaxis.get_majorticklocs()
            if loc >= ymin and loc <= ymax]
    tickline, = ax.plot([xloc]*len(locs), locs, linestyle='',
                        marker=lines.TICKLEFT, **props)
    axline, = ax.plot([xloc, xloc], [ymin, ymax], **props)
    tickline.set_clip_on(False)
    axline.set_clip_on(False)
    for loc in locs:
        ax.text(xloc - offset, loc,
                verticalalignment='center',
                horizontalalignment='right')

props = dict(color='black', linewidth=2, markeredgewidth=2)
x = np.arange(200.)
y = np.sin(2*np.pi*x/200.) + np.random.rand(200) - 0.5
fig = plt.figure(facecolor='white')
ax = fig.add_subplot(111, frame_on=False)
ax.axison = False
ax.plot(x, y, 'd', markersize=8, markerfacecolor='blue')
ax.set_xlim(0, 200)
ax.set_ylim(-1.5, 1.5)
make_xaxis(ax, 0, offset=0.1, **props)
make_yaxis(ax, 0, offset=5, **props)
plt.show()
import matplotlib.pyplot as plt
import matplotlib.path as mpath
import numpy as np

star = mpath.Path.unit_regular_star(6)
circle = mpath.Path.unit_circle()

verts = np.concatenate([circle.vertices, star.vertices[::-1, ...]])
codes = np.concatenate([circle.codes, star.codes])
cut_star = mpath.Path(verts, codes)

plt.plot(np.arange(10)**2, 'r--', marker=cut_star, markersize=15)
plt.show()
99.126  `pylab_examples` example code: `markevery_demo.py`

```
markevery=None

markevery=8

markevery=(30, 8)

markevery=[16, 30]

markevery=[0, -1]

markevery=slice(100, 200, 3)

markevery=0.1

markevery=0.3

markevery=1.5

markevery=(0.0, 0.1)

markevery=(0.45, 0.1)
```

2596 Chapter 99. `pylab_examples` Examples
```
markevery=None

markevery=[16, 24, 30]

markevery=8

markevery=(30, 8)

markevery=[0, -1]

markevery=slice(100, 200, 3)

markevery=0.1

markevery=0.3

markevery=1.5

markevery=(0.0, 0.1)

markevery=(0.45, 0.1)
```
This example demonstrates the various options for showing a marker at a subset of data points using the `markevery` property of a Line2D object.

Integer arguments are fairly intuitive. e.g. `markevery=5` will plot every 5th marker starting from the first data point.

Float arguments allow markers to be spaced at approximately equal distances along the line. The theoretical distance along the line between markers is determined by multiplying the display-coordinate distance of the axes bounding-box diagonal by the value of `markevery`. The data points closest to the theoretical distances will be shown.

A slice or list/array can also be used with `markevery` to specify the markers to show.

```python
from __future__ import division
import numpy as np
import matplotlib.pyplot as plt
```
import matplotlib.gridspec as gridspec

# define a list of markevery cases to plot
cases = [None,
    8,
    (30, 8),
    [16, 24, 30], [0, -1],
    slice(100, 200, 3),
    0.1, 0.3, 1.5,
    (0.0, 0.1), (0.45, 0.1)]

# define the figure size and grid layout properties
figsize = (10, 8)
cols = 3
gs = gridspec.GridSpec(len(cases) // cols + 1, cols)

# define the data for cartesian plots
delta = 0.11
x = np.linspace(0, 10 - 2 * delta, 200) + delta
y = np.sin(x) + 1.0 + delta

# plot each markevery case for linear x and y scales
fig1 = plt.figure(num=1, figsize=figsize)
ax = []
for i, case in enumerate(cases):
    row = (i // cols)
    col = i % cols
    ax.append(fig1.add_subplot(gs[row, col]))
    ax[-1].set_title('markevery=%s' % str(case))
    ax[-1].plot(x, y, 'o', ls='-', ms=4, markevery=case)
#fig1.tight_layout()

# plot each markevery case for log x and y scales
fig2 = plt.figure(num=2, figsize=figsize)
axlog = []
for i, case in enumerate(cases):
    row = (i // cols)
    col = i % cols
    axlog.append(fig2.add_subplot(gs[row, col]))
    axlog[-1].set_title('markevery=%s' % str(case))
    axlog[-1].set_xscale('log')
    axlog[-1].set_yscale('log')
    axlog[-1].plot(x, y, 'o', ls='-', ms=4, markevery=case)
fig2.tight_layout()

# plot each markevery case for linear x and y scales but zoomed in
# note the behaviour when zoomed in. When a start marker offset is specified
# it is always interpreted with respect to the first data point which might be
# different to the first visible data point.
fig3 = plt.figure(num=3, figsize=figsize)
axzoom = []
for i, case in enumerate(cases):
    row = (i // cols)
    col = i % cols
    axzoom.append(fig3.add_subplot(gs[row, col]))
    axzoom[-1].set_title('markevery=%s' % str(case))
    axzoom[-1].plot(x, y, 'o', ls='-', ms=4, markevery=case)
#fig3.tight_layout()
col = i % cols
axzoom.append(fig3.add_subplot(gs[row, col]))
axzoom[-1].set_title('markevery=%s' % str(case))
axzoom[-1].plot(x, y, 'o', ls='-', ms=4, markevery=case)
axzoom[-1].set_xlim((6, 6.7))
axzoom[-1].set_ylim((1.1, 1.7))
fig3.tight_layout()

# define data for polar plots
r = np.linspace(0, 3.0, 200)
theta = 2 * np.pi * r

# plot each markevery case for polar plots
fig4 = plt.figure(num=4, figsize=figsize)
axpolar = []
for i, case in enumerate(cases):
    row = (i // cols)
col = i % cols
    axpolar.append(fig4.add_subplot(gs[row, col], projection='polar'))
    axpolar[-1].set_title('markevery=%s' % str(case))
    axpolar[-1].plot(theta, r, 'o', ls='-', ms=4, markevery=case)
fig4.tight_layout()
plt.show()
Plot lines with points masked out.

This would typically be used with gappy data, to break the line at the data gaps.

```python
import matplotlib.pyplot as plt
import numpy as np

x = np.arange(0, 2*np.pi, 0.02)
y = np.sin(x)
y1 = np.sin(2*x)
y2 = np.sin(3*x)

ym1 = np.ma.masked_where(y1 > 0.5, y1)
ym2 = np.ma.masked_where(y2 < -0.5, y2)

lines = plt.plot(x, y, x, ym1, x, ym2, 'o')
plt.setp(lines[0], linewidth=4)
plt.setp(lines[1], linewidth=2)
plt.setp(lines[2], markersize=10)
```
Use matplotlib's internal LaTeX parser and layout engine. For true
latex rendering, see the text.usetex option

import numpy as np
from matplotlib.pyplot import figure, show

fig = figure()
fig.subplots_adjust(bottom=0.2)

ax = fig.add_subplot(111)
ax.plot([1, 2, 3], 'r')
ax.plot([1, 2, 3], 'r')
x = np.arange(0.0, 3.0, 0.1)
ax.grid(True)
ax.set_xlabel(r'$\Delta_i^j$', fontsize=20)
ax.set_ylabel(r'$\Delta_{i+1}^j$', fontsize=20)
tex = r'$\mathcal{R} \prod_{i=\alpha_{i+1}}^\infty a_i \sin(2 \pi f x_i)$'
ax.text(1, 1.6, tex, fontsize=20, va='bottom')
ax.legend([r'$$\sqrt{x^2}$$'])
ax.set_title(r'$\Delta_i^j \hspace{0.4} \mathrm{versus} \hspace{0.4} \Delta_{i+1}^j$', fontsize=20)
show()
Matplotlib's math rendering engine

\[
W^{3\beta}_{\delta_1, \rho_1, \sigma_2} = U^{3\beta}_{\delta_1, \rho_1} + \frac{1}{8\pi^2} \int_{\alpha_2}^{\alpha_2'} d\alpha_2' \left[ \frac{U^{2\beta}_{\delta_1, \rho_1} - \alpha_2' U^{1\beta}_{\rho_1, \sigma_2}}{U^{0\beta}_{\rho_1, \sigma_2}} \right]
\]

Subscripts and superscripts:
\[\alpha_i > \beta_i, \quad \alpha_i^j + 1 = \sin(2\pi f_j t_i) e^{-5t_i/\tau}, \ldots\]

Fractions, binomials and stacked numbers:
\[
\frac{3}{4}, \quad \frac{\frac{3}{4}}{\frac{5}{4}}, \quad \frac{3}{4}, \quad \left(\frac{5 - \frac{1}{x}}{4}\right), \ldots
\]

Radicals:
\[
\sqrt{2}, \quad \sqrt[x]{y}, \ldots
\]

Fonts:
Roman, Italics, Typewriter or \textit{caligraphic}

Accents:
\[\acute{a}, \grave{a}, \breve{a}, \grave{a}, \hat{a}, \tilde{a}, \bar{a}, \ddot{a}, \dddot{a}, \dddot{x}, \dddot{y}, \ldots\]

Greek, Hebrew:
\[\alpha, \beta, \chi, \delta, \lambda, \mu, \Delta, \Gamma, \Omega, \Phi, \Pi, \Upsilon, \nabla, \Xi, \zeta, \eta, \iota, \ldots\]

Delimiters, functions and Symbols:
\[\cup, \int, \phi, \prod, \Sigma, \log, \sin, \approx, \oplus, *, \infty, \partial, \mathbb{R}\]

-----
Selected features of Matplotlib's math rendering engine.
-----

from \_future\_ import print\_function
import matplotlib.pyplot as plt
import os
import sys
import re
import gc

# Selection of features following "Writing mathematical expressions" tutorial
mathtext_titles = {
    0: "Header demo",
    1: "Subscripts and superscripts",
    2: "Fractions, binomials and stacked numbers",
    3: "Radicals",
    4: "Fonts",
    5: "Accents",
    6: "Greek, Hebrew",
    7: "Delimiters, functions and Symbols"
}

n_lines = len(mathtext_titles)

# Randomly picked examples
mathtext_demos = {
    0: r"$W^{3\beta}_{\delta_1 \rho_1 \sigma_2} = \" 
        r"U^\alpha_{\beta} - \frac{1}{8 \pi 2} \int \alpha_2 \ldots$",
    1: r"$\alpha_{i+1}^j = \sin(2\pi f_j t_i) e^{-5 t_i/\tau}, \ldots$",
    2: r"$\binom{3}{4}, \stackrel{3}{4}, \left(\frac{5 - \frac{1}{x}}{4}\right), \ldots$",
    3: r"$\sqrt{2}, \sqrt[3]{x}, \ldots$",
    4: r"$\mathrm{Roman}, \mathit{Italic}, \mathbb{Typewriter}$",
    5: r"$\acute a, \bar a, \breve a, \ddot a, \grave a, \hat a, \tilde a, \vec a, \widehat{xyz}, \widetilde{xyz}, \ldots$",
    6: r"$\alpha, \beta, \chi, \delta, \lambda, \mu, \Delta, \Gamma, \Omega, \Phi, \Pi, \Upsilon, \nabla, \aleph, \beth, \daleth, \gimel, \ldots$",
    7: r"$\coprod, \int, \oint, \prod, \sum, \log, \sin, \approx, \oplus, \star, \varpropto, \infty, \partial, \Re, \leftrightsquigarrow, \ldots$"
}

def doall():
# Creating figure and axis.
plt.figure(figsize=(6,7))
plt.axes([0.01, 0.01, 0.98, 0.90], facecolor="white", frameon=True)
plt.gca().set_xlim(0., 1.)
plt.gca().set_ylim(0., 1.)
plt.gca().set_title("Matplotlib's math rendering engine",
color=mpl_grey_rvb, fontsize=14, weight='bold')
plt.gca().set_xticklabels('', visible=False)
plt.gca().set_yticklabels('', visible=False)

# Gap between lines in axes coords
line_axesfrac = (1. / (n_lines))

# Plotting header demonstration formula
full_demo = mathtext_demos[0]
plt.annotate(full_demo,
xy=(0.5, 1. - 0.59*line_axesfrac),
xycoords='data', color=mpl_orange_rvb, ha='center',
fontsize=20)

# Plotting features demonstration formulae
for i_line in range(1, n_lines):
    baseline = 1. - (i_line)*line_axesfrac
    baseline_next = baseline - line_axesfrac*1.
    title = mathtext_titles[i_line] + "::
    fill_color = ['white', mpl_blue_rvb][i_line % 2]
    plt.fill_between([0., 1.], [baseline, baseline],
                     [baseline_next, baseline_next],
                     color=fill_color, alpha=0.5)
    plt.annotate(title,
                 xy=(0.07, baseline - 0.3*line_axesfrac),
                 xycoords='data', color=mpl_grey_rvb, weight='bold')
    demo = mathtext_demos[i_line]
    plt.annotate(demo,
                 xy=(0.05, baseline - 0.75*line_axesfrac),
                 xycoords='data', color=mpl_grey_rvb,
                 fontsize=16)

for i in range(n_lines):
    s = mathtext_demos[i]
    print(i, s)
plt.show()

if '--latex' in sys.argv:
    # Run: python mathtext_examples.py --latex
    # Need amsmath and amssymb packages.
    fd = open("mathtext_examples.ltx", "w")
    fd.write("\documentclass\{article\}\n")
    fd.write("\usepackage\{amsmath, amssymb\}\n")
    fd.write("\begin\{document\}\n")
    fd.write("\begin\{enumerate\}\n")
for i in range(n_lines):
    s = mathext_demos[i]
    s = re.sub(r'\(?<!\)\$, "$$", s)
    fd.write("\item %s
" % s)

    fd.write("\end{enumerate}
")
    fd.write("\end{document}
")
    fd.close()

    os.system("pdflatex mathtext_examples.ltx")
else:
    doall()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

99.130 pylab_examples example code: matshow.py

"""Simple matshow() example."""
import matplotlib.pyplot as plt
import numpy as np

def samplemat(dims):
    """Make a matrix with all zeros and increasing elements on the diagonal"""
    aa = np.zeros(dims)
    for i in range(min(dims)):
        aa[i, i] = i
    return aa

# Display matrix
plt.matshow(samplemat((15, 35)))
Displays an MRI image.

```python
import matplotlib.pyplot as plt
import matplotlib.cbook as cbook
import matplotlib.cm as cm
import numpy as np

fig, ax = plt.subplots(num="MRI_demo")

# Data are 256x256 16 bit integers
dfile = cbook.get_sample_data('s1045.ima.gz')
im = np.fromstring(dfile.read(), np.uint16).astype(float)
im.shape = (256, 256)
dfile.close()
```
```python
ax.imshow(im, cmap=cm.gray)
ax.axis('off')
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

99.132 `pylab_examples` example code: mri_with_eeg.py

```python
from __future__ import division, print_function

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cbook as cbook
import matplotlib.cm as cm

from matplotlib.collections import LineCollection
from matplotlib.ticker import MultipleLocator

"""Displays a set of subplots with an MRI image, its intensity histogram and some EEG traces."
""

from __future__ import division, print_function

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cbook as cbook
import matplotlib.cm as cm

from matplotlib.collections import LineCollection
from matplotlib.ticker import MultipleLocator

"""Displays a set of subplots with an MRI image, its intensity histogram and some EEG traces."
""

from __future__ import division, print_function

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cbook as cbook
import matplotlib.cm as cm

from matplotlib.collections import LineCollection
from matplotlib.ticker import MultipleLocator
```

"""Displays a set of subplots with an MRI image, its intensity histogram and some EEG traces."
"""
**fig = plt.figure(”MRI_with_EEG”)**

*# Load the MRI data (256x256 16 bit integers)*

dfile = cbook.get_sample_data(’s1045.ima.gz’)
im = np.fromstring(dfile.read(), np.uint16).astype(float)
im.shape = (256, 256)
dfile.close()

*# Plot the MRI image*

ax0 = fig.add_subplot(2, 2, 1)
ax0.imshow(im, cmap=cm.gray)
ax0.axis(’off’)

*# Plot the histogram of MRI intensity*

ax1 = fig.add_subplot(2, 2, 2)
im = np.ravel(im)
im = im[np.nonzero(im)]  # Ignore the background
im = im / (2**16 - 1)  # Normalize
ax1.hist(im, bins=100)
ax1.xaxis.set_major_locator(MultipleLocator(0.4))
ax1.minorticks_on()
ax1.set_yticks([])
ax1.set_xlabel(’Intensity (a.u.’)
ax1.set_ylabel(’MRI density’)

*# Load the EEG data*

numSamples, numRows = 800, 4
eegfile = cbook.get_sample_data(’eeg.dat’, asfileobj=False)
print(’Loading EEG %s’ % eegfile)
data = np.fromfile(eegfile, dtype=float)
data.shape = (numSamples, numRows)
t = 10.0 * np.arange(numSamples) / numSamples

*# Plot the EEG*

ticklocs = []

ax2 = fig.add_subplot(2, 1, 2)
ax2.set_xlim(0, 10)
ax2.set_xticks(np.arange(10))
dmin = data.min()
dmax = data.max()
dr = (dmax - dmin) * 0.7  # Crowd them a bit.
y0 = dmin
y1 = (numRows - 1) * dr + dmax
ax2.set_ylim(y0, y1)

segs = []

for i in range(numRows):
    segs.append(np.hstack((t[:, np.newaxis], data[:, i, np.newaxis])))
ticklocs.append(i * dr)

offsets = np.zeros((numRows, 2), dtype=float)
offsets[:, 1] = ticklocs
lines = LineCollection(segs, offsets=offsets, transOffset=None)
ax2.add_collection(lines)

# Set the yticks to use axes coordinates on the y axis
ax2.set_yticks(ticklocs)
ax2.set_yticklabels(['PG3', 'PG5', 'PG7', 'PG9'])

ax2.set_xlabel('Time (s)')

plt.tight_layout()
plt.show()
from matplotlib import pyplot as plt
import matplotlib.colors as mcolors
import matplotlib.font_manager as font_manager
import numpy as np

Nr = 3
Nc = 2

fig = plt.figure()
cmap = mcolors.cool

figtitle = 'Multiple images'
t = fig.text(0.5, 0.95, figtitle,
            horizontalalignment='center',
            fontproperties=font_manager.FontProperties(size=16))

cax = fig.add_axes([0.2, 0.08, 0.6, 0.04])

w = 0.4
h = 0.22
ax = []
images = []
vmin = 1e40
vmax = -1e40
for i in range(Nr):
    for j in range(Nc):
        pos = [0.075 + j*1.1*w, 0.18 + i*1.2*h, w, h]
        a = fig.add_axes(pos)
        if i > 0:
            a.set_xticklabels([])
        # Make some fake data with a range that varies
        # somewhat from one plot to the next.
        data = ((1 + i + j)/10.0)*np.random.rand(10, 20)*1e-6
        dd = data.ravel()
        # Manually find the min and max of all colors for
        # use in setting the color scale.
        vmin = min(vmin, np.amin(dd))
        vmax = max(vmax, np.amax(dd))
        images.append(a.imshow(data, cmap=cmap))
        ax.append(a)

        # Set the first image as the master, with all the others
        # observing it for changes in cmap or norm.

class ImageFollower(object):
    'update image in response to changes in clim or cmap on another image'

    def __init__(self, follower):
        pass
self.follower = follower

def __call__(self, leader):
    self.follower.set_cmap(leader.get_cmap())
    self.follower.set_clim(leader.get_clim())

norm = colors.Normalize(vmin=vmin, vmax=vmax)
for i, im in enumerate(images):
    im.set_norm(norm)
    if i > 0:
        images[0].callbacksSM.connect('changed', ImageFollower(im))

    # The colorbar is also based on this master image.
    fig.colorbar(images[0], cax, orientation='horizontal')

    # We need the following only if we want to run this interactively and
    # modify the colormap:
    axes(ax[0])  # Return the current axes to the first one,
    sci(images[0])  # because the current image must be in current axes.

show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
99.134  *pylab_examples* example code: *multicolored_line.py*

![Image of a graph with two sine waves of different colors and amplitudes, demonstrating the use of *pylab_examples* example code for creating multicolored lines in plots.]
Color parts of a line based on its properties, e.g., slope.

```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.collections import LineCollection
from matplotlib.colors import ListedColormap, BoundaryNorm

x = np.linspace(0, 3 * np.pi, 500)
y = np.sin(x)
z = np.cos(0.5 * (x[:-1] + x[1:]))  # first derivative

# Create a colormap for red, green and blue and a norm to color
# f' < -0.5 red, f' > 0.5 blue, and the rest green
cmap = ListedColormap(['r', 'g', 'b'])
norm = BoundaryNorm([-1, -0.5, 0.5, 1], cmap.N)

# Create a set of line segments so that we can color them individually
# This creates the points as a N x 1 x 2 array so that we can stack points
# together easily to get the segments. The segments array for line collection
# needs to be numlines x points per line x 2 (x and y)
points = np.array([x, y]).T.reshape(-1, 1, 2)
segments = np.concatenate([points[:-1], points[1:]], axis=1)

# Create the line collection object, setting the colormapping parameters.
```
```python
# Have to set the actual values used for colormapping separately.
lc = LineCollection(segments, cmap=cmap, norm=norm)
lc.set_array(z)
lc.set_linewidth(3)

fig1 = plt.figure()
plt.gca().add_collection(lc)
plt.xlim(x.min(), x.max())
plt.ylim(-1.1, 1.1)

# Now do a second plot coloring the curve using a continuous colormap

t = np.linspace(0, 10, 200)
x = np.cos(np.pi * t)
y = np.sin(t)
points = np.array([x, y]).T.reshape(-1, 1, 2)
segments = np.concatenate([points[:-1], points[1:]], axis=1)

lc = LineCollection(segments, cmap=plt.get_cmap('copper'),
                    norm=plt.Normalize(0, 10))
lc.set_array(t)
lc.set_linewidth(3)

fig2 = plt.figure()
plt.gca().add_collection(lc)
plt.xlim(-1, 1)
plt.ylim(-1, 1)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
import matplotlib.pyplot as plt
import numpy as np

plt.figure(figsize=(7, 4))
ax = plt.subplot(121)
ax.set_aspect(1)
plt.plot(np.arange(10))
plt.xlabel('this is a xlabel
(with newlines!)
0
2
4
6
8
this is vertical
test this is
yet another test
this is
yet another test
0.0 2.5 5.0 7.5

plt.ylabel('this is vertical
test', multialignment='center')
plt.text(2, 7, 'this is
yet another test',
rotation=45,
horizontalalignment='center',
verticalalignment='top',
multialignment='center')

plt.grid(True)

plt.subplot(122)
plt.text(0.29, 0.4, "Mat\nTTp
123", size=18,
va="baseline", ha="right", multialignment="left",
bbox=dict(fc="none"))

plt.text(0.34, 0.4, "Mag\nTTT
123", size=18,
va="baseline", ha="left", multialignment="left",
bbox=dict(fc="none"))
plt.text(0.95, 0.4, "Mag\nTTT$^{A^A}\n123"). size=18,
       va="baseline", ha="right", multialignment="left",
       bbox=dict(fc="none"))

plt.xticks([0.2, 0.4, 0.6, 0.8, 1.],
           ['Jan\n2009', 'Feb\n2009', 'Mar\n2009', 'Apr\n2009', 'May\n2009'])

plt.axhline(0.4)
plt.title("test line spacing for multilne text")

plt.subplots_adjust(bottom=0.25, top=0.75)
plt.draw()
plt.show()
plt.title('Page Three')
pdf.savefig(fig)  # or you can pass a Figure object to pdf.savefig
plt.close()

# We can also set the file's metadata via the PdfPages object:
d = pdf.infodict()
d['Title'] = 'Multipage PDF Example'
d['Author'] = u'Jouni K. Seppänen'
d['Subject'] = 'How to create a multipage pdf file and set its metadata'
d['Keywords'] = 'PdfPages multipage keywords author title subject'
d['CreationDate'] = datetime.datetime(2009, 11, 13)
d['ModDate'] = datetime.datetime.today()
# Working with multiple figure windows and subplots

```python
import matplotlib.pyplot as plt
import numpy as np

t = np.arange(0.0, 2.0, 0.01)
s1 = np.sin(2*np.pi*t)
s2 = np.sin(4*np.pi*t)

plt.figure(1)
plt.subplot(211)
plt.plot(t, s1)
plt.subplot(212)
plt.plot(t, 2*s1)
plt.figure(2)
plt.plot(t, s2)

# now switch back to figure 1 and make some changes
plt.figure(1)
plt.subplot(211)
plt.plot(t, s2, 's')
ax = plt.gca()
ax.set_xticklabels([])

plt.show()
```
import matplotlib.pyplot as plt

def make_patch_spines_invisible(ax):
    ax.set_frame_on(True)
    ax.patch.set_visible(False)
    for sp in ax.spines.values():
        sp.set_visible(False)

fig, host = plt.subplots()
fig.subplots_adjust(right=0.75)
par1 = host.twinx()
par2 = host.twinx()

# Offset the right spine of par2. The ticks and label have already been
# placed on the right by twinx above.
par2.spines["right"].set_position(('axes', 1.2))
# Having been created by twinx, par2 has its frame off, so the line of its
# detached spine is invisible. First, activate the frame but make the patch # and spines invisible.
make_patch_spines_invisible(par2)
# Second, show the right spine.
par2.spines["right"].set_visible(True)

p1, = host.plot([0, 1, 2], [0, 1, 2], "b-", label="Density")
p2, = par1.plot([0, 1, 2], [0, 3, 2], "r-", label="Temperature")
p3, = par2.plot([0, 1, 2], [50, 30, 15], "g-", label="Velocity")

host.set_xlim(0, 2)
host.set_ylim(0, 2)
par1.set_ylim(0, 4)
par2.set_ylim(1, 65)

host.set_xlabel("Distance")
host.set_ylabel("Density")
par1.set_ylabel("Temperature")
par2.set_ylabel("Velocity")

host.yaxis.label.set_color(p1.get_color())
par1.yaxis.label.set_color(p2.get_color())
par2.yaxis.label.set_color(p3.get_color())

tkw = dict(size=4, width=1.5)
host.tick_params(axis='y', colors=p1.get_color(), **tkw)
par1.tick_params(axis='y', colors=p2.get_color(), **tkw)
par2.tick_params(axis='y', colors=p3.get_color(), **tkw)
host.tick_params(axis='x', **tkw)

lines = [p1, p2, p3]
host.legend(lines, [l.get_label() for l in lines])

plt.show()
```python
import numpy as np
import matplotlib.pyplot as plt

t = np.arange(0.0, 1.0 + 0.01, 0.01)
s = np.cos(2 * 2 * np.pi * t)
t[41:60] = np.nan

plt.subplot(2, 1, 1)
plt.plot(t, s, '-o', lw=2)
plt.xlabel('time (s)')
plt.ylabel('voltage (mV)')
plt.title('A sine wave with a gap of NaNs between 0.4 and 0.6')
plt.grid(True)

plt.subplot(2, 1, 2)
t[0] = np.nan
t[-1] = np.nan
plt.plot(t, s, '-o', lw=2)
```

"""
Example: simple line plots with NaNs inserted.
"""

```
plt.title('Also with NaN in first and last point')
plt.xlabel('time (s)')
plt.ylabel('more nans')
plt.grid(True)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

99.140 `pylab_examples` example code: `newscalarformatter_demo.py`

The old formatter

![Graph showing the old formatter with NaN values in the first and last points.

The plots illustrate the correct display of numbers with NaN values, highlighting the improvements in handling these values with the new scalar formatter.
The new formatter, default settings
The new formatter, no numerical offset

```
1.000000 1.000005 ... 50000
 0
0.00010
0.00008
0.00006
0.00004
0.00002
0.00000
```

99.140. pylab_examples example code: newscalarformatter_demo.py 2627
import matplotlib.pyplot as plt
import numpy as np
from matplotlib.ticker import OldScalarFormatter, ScalarFormatter

# Example 1
x = np.arange(0, 1, .01)
fig, [[ax1, ax2], [ax3, ax4]] = plt.subplots(2, 2, figsize=(6, 6))
fig.text(0.5, 0.975, 'The old formatter',
          horizontalalignment='center', verticalalignment='top')
ax1.plot(x * 1e5 + 1e10, x * 1e-10 + 1e-5)
ax1.xaxis.set_major_formatter(OldScalarFormatter())
ax1.yaxis.set_major_formatter(OldScalarFormatter())

ax2.plot(x * 1e5, x * 1e-4)
ax2.xaxis.set_major_formatter(OldScalarFormatter())
ax2.yaxis.set_major_formatter(OldScalarFormatter())
Example 2

```python
import matplotlib.pyplot as plt
import numpy as np

fig, [[ax1, ax2], [ax3, ax4]] = plt.subplots(2, 2, figsize=(6, 6))
fig.text(0.5, 0.975, 'The new formatter, default settings',
        horizontalalignment='center',
        verticalalignment='top')

ax1.plot(x * 1e5 + 1e10, x * 1e-10 + 1e-5)
ax1.xaxis.set_major_formatter(ScalarFormatter())
ax1.yaxis.set_major_formatter(ScalarFormatter())

ax2.plot(x * 1e5, x * 1e-4)
ax2.xaxis.set_major_formatter(ScalarFormatter())
ax2.yaxis.set_major_formatter(ScalarFormatter())

ax3.plot(-x * 1e5 - 1e10, -x * 1e-5 - 1e-10)
ax3.xaxis.set_major_formatter(ScalarFormatter())
ax3.yaxis.set_major_formatter(ScalarFormatter())

ax4.plot(-x * 1e5, -x * 1e-4)
ax4.xaxis.set_major_formatter(ScalarFormatter())
ax4.yaxis.set_major_formatter(ScalarFormatter())

fig.subplots_adjust(wspace=0.7, hspace=0.6)
```

Example 3

```python
import matplotlib.pyplot as plt
import numpy as np

fig, [[ax1, ax2], [ax3, ax4]] = plt.subplots(2, 2, figsize=(6, 6))
fig.text(0.5, 0.975, 'The new formatter, no numerical offset',
        horizontalalignment='center',
        verticalalignment='top')

ax1.plot(x * 1e5 + 1e10, x * 1e-10 + 1e-5)
ax1.xaxis.set_major_formatter(ScalarFormatter(useOffset=False))
ax1.yaxis.set_major_formatter(ScalarFormatter(useOffset=False))

ax2.plot(x * 1e5, x * 1e-4)
ax2.xaxis.set_major_formatter(ScalarFormatter(useOffset=False))
ax2.yaxis.set_major_formatter(ScalarFormatter(useOffset=False))

ax3.plot(-x * 1e5 - 1e10, -x * 1e-5 - 1e-10)
ax3.xaxis.set_major_formatter(ScalarFormatter())
ax3.yaxis.set_major_formatter(ScalarFormatter())

ax4.plot(-x * 1e5, -x * 1e-4)
ax4.xaxis.set_major_formatter(ScalarFormatter())
ax4.yaxis.set_major_formatter(ScalarFormatter())

fig.subplots_adjust(wspace=0.7, hspace=0.6)
```
ax3.yaxis.set_major_formatter(ScalarFormatter(useOffset=False))
ax4.plot(-x * 1e5, -x * 1e-4)
ax4.xaxis.set_major_formatter(ScalarFormatter(useOffset=False))
ax4.yaxis.set_major_formatter(ScalarFormatter(useOffset=False))

fig.subplots_adjust(wspace=0.7, hspace=0.6)

# Example 4
x = np.arange(0, 1, .01)
fig, [[ax1, ax2], [ax3, ax4]] = plt.subplots(2, 2, figsize=(6, 6))
fig.text(0.5, 0.975, '
The new formatter, with mathtext',
        horizontalalignment='center',
        verticalalignment='top')

ax1.plot(x * 1e5 + 1e10, x * 1e-10 + 1e-5)
ax1.xaxis.set_major_formatter(ScalarFormatter(useMathText=True))
ax1.yaxis.set_major_formatter(ScalarFormatter(useMathText=True))

ax2.plot(x * 1e5, x * 1e-4)
ax2.xaxis.set_major_formatter(ScalarFormatter(useMathText=True))
ax2.yaxis.set_major_formatter(ScalarFormatter(useMathText=True))

ax3.plot(-x * 1e5 - 1e10, -x * 1e-5 - 1e-10)
ax3.xaxis.set_major_formatter(ScalarFormatter(useMathText=True))
ax3.yaxis.set_major_formatter(ScalarFormatter(useMathText=True))

ax4.plot(-x * 1e5, -x * 1e-4)
ax4.xaxis.set_major_formatter(ScalarFormatter(useMathText=True))
ax4.yaxis.set_major_formatter(ScalarFormatter(useMathText=True))

fig.subplots_adjust(wspace=0.7, hspace=0.6)

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
```python
import matplotlib.pyplot as plt
import matplotlib.patheffects as PathEffects
import numpy as np

if 1:
    plt.figure(1, figsize=(8, 3))
    ax1 = plt.subplot(131)
    ax1.imshow([[1, 2], [2, 3]])
    txt = ax1.annotate("test", (1., 1.), (0., 0),
        arrowprops=dict(arrowstyle="->",
                         connectionstyle="angle3", lw=2),
        size=20, ha="center", path_effects=[PathEffects."
            withStroke(linewidth=3,
                        foreground="w")])
    txt.arrow_patch.set_path_effects([PathEffects.Stroke(linewidth=5, foreground="w"),
                                        PathEffects.Normal()])
    ax1.grid(True, linestyle="-")
    pe = [PathEffects.withStroke(linewidth=3,
                                 foreground="w")]
    for l in ax1.get_xgridlines() + ax1.get_ygridlines():
        l.set_path_effects(pe)
    ax2 = plt.subplot(132)
    arr = np.arange(25).reshape((5, 5))
    ax2.imshow(arr)
    cntr = ax2.contour(arr, colors="k")
    plt.setp(cntr.collections, path_effects=[PathEffects.withStroke(linewidth=3, foreground="w")])
    clbls = ax2.clabel(cntr, fmt="%.2f", use_clabeltext=True)
```

---

**99.141**  
**pylab_examples example code: patheffect_demo.py**
```python
plt.setp(clbls, path_effects=[
    PathEffects.withStroke(linewidth=3, foreground="w")])

# shadow as a path effect
ax3 = plt.subplot(133)
p1, = ax3.plot([0, 1], [0, 1])
leg = ax3.legend([p1], ["Line 1"], fancybox=True, loc=2)
leg.legendPatch.set_path_effects([PathEffects.withSimplePatchShadow()])

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

99.142 pylab_examples example code: pcolor_demo.py

```python
import matplotlib.pyplot as plt
import numpy as np
```

""
Demonstrates similarities between pcolor, pcolormesh, imshow and pcolorfast
for drawing quadrilateral grids.
"""
# make these smaller to increase the resolution
dx, dy = 0.15, 0.05

# generate 2 2d grids for the x & y bounds
y, x = np.mgrid[slice(-3, 3 + dy, dy),
                slice(-3, 3 + dx, dx)]

z = (1 - x / 2. + x ** 5 + y ** 3) * np.exp(-x ** 2 - y ** 2)
# x and y are bounds, so z should be the value *inside* those bounds.
# Therefore, remove the last value from the z array.
z = z[:1, :1]
z_min, z_max = -np.abs(z).max(), np.abs(z).max()

plt.subplot(2, 2, 1)
plt.pcolor(x, y, z, cmap='RdBu', vmin=z_min, vmax=z_max)
plt.title('pcolor')
# set the limits of the plot to the limits of the data
plt.axis([x.min(), x.max(), y.min(), y.max()])
plt.colorbar()

plt.subplot(2, 2, 2)
plt.pcolormesh(x, y, z, cmap='RdBu', vmin=z_min, vmax=z_max)
plt.title('pcolormesh')
# set the limits of the plot to the limits of the data
plt.axis([x.min(), x.max(), y.min(), y.max()])
plt.colorbar()

plt.subplot(2, 2, 3)
plt.imshow(z, cmap='RdBu', vmin=z_min, vmax=z_max,
           extent=[x.min(), x.max(), y.min(), y.max()],
           interpolation='nearest', origin='lower')
plt.title('image (nearest)')
plt.colorbar()

ax = plt.subplot(2, 2, 4)
ax.pcolorfast(x, y, z, cmap='RdBu', vmin=z_min, vmax=z_max)
plt.title('pcolorfast')
plt.colorbar()

plt.subplots_adjust(wspace=0.5, hspace=0.5)
plt.show()
import matplotlib.pyplot as plt
from matplotlib.colors import LogNorm
import numpy as np
from matplotlib.mlab import bivariate_normal

N = 100
X, Y = np.mgrid[-3:3:complex(0, N), -2:2:complex(0, N)]

# A low hump with a spike coming out of the top right.
# Needs to have z/colour axis on a log scale so we see both hump and spike.
# linear scale only shows the spike.
Z1 = bivariate_normal(X, Y, 0.1, 0.2, 1.0, 1.0) + 0.1 * bivariate_normal(X, Y, 1.0, 1.0, 0.0, 0.0)

plt.subplot(2, 1, 1)
plt.pcolor(X, Y, Z1, norm=LogNorm(vmin=Z1.min(), vmax=Z1.max()), cmap='PuBu_r')
plt.colorbar()

plt.subplot(2, 1, 2)
plt.pcolor(X, Y, Z1, cmap='PuBu_r')
plt.colorbar()
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

99.144  pylab_examples example code: pcolor_small.py

import matplotlib.pyplot as plt
from numpy.random import rand

Z = rand(6, 10)

plt.subplot(2, 1, 1)
c = plt.pcolor(Z)
plt.title('default: no edges')

plt.subplot(2, 1, 2)
c = plt.pcolor(Z, edgecolors='k', linewidths=4)
plt.title('thick edges')

plt.show()
Make a pie charts of varying size - see http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.pie for the docstring.

This example shows a basic pie charts with labels optional features, like autolabeling the percentage, offsetting a slice with "explode" and adding a shadow, in different sizes.

```python
import matplotlib.pyplot as plt
from matplotlib.gridspec import GridSpec

# Some data

labels = ['Frogs', 'Hogs', 'Dogs', 'Logs']
fracs = [15, 30, 45, 10]

explode = (0, 0.05, 0, 0)

...
# Make square figures and axes

```python
the_grid = GridSpec(2, 2)

plt.subplot(the_grid[0, 0], aspect=1)
plt.pie(fracs, labels=labels, autopct='%.1f%%', shadow=True)

plt.subplot(the_grid[0, 1], aspect=1)
plt.pie(fracs, explode=explode, labels=labels, autopct='%.0f%%', shadow=True)

plt.subplot(the_grid[1, 0], aspect=1)
patches, texts, autotexts = plt.pie(fracs, labels=labels,
    autopct='%.0f%%',
    shadow=True, radius=0.5)

# Make the labels on the small plot easier to read.
for t in texts:
    t.set_size('smaller')
for t in autotexts:
    t.set_size('x-small')
autotexts[0].set_color('y')

plt.subplot(the_grid[1, 1], aspect=1)

# Turn off shadow for tiny plot with exploded slice.
patches, texts, autotexts = plt.pie(fracs, explode=explode,
labels=labels, autopct='%.0f%%',
shadow=False, radius=0.5)

for t in texts:
    t.set_size('smaller')
for t in autotexts:
    t.set_size('x-small')
autotexts[0].set_color('y')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
99.146 pylab_examples example code: plotfile_demo.py
$f(x) = x^2$

$\begin{array}{ll}
f(x) = x^2 \\
0 & 25 \\
0 & 50 \\
0 & 75 \\
0 & 100 \\
\end{array}$

$\begin{array}{ll}
f(x) = x^3 \\
0 & 250 \\
0 & 500 \\
0 & 750 \\
0 & 1000 \\
\end{array}$
$f(x) = x^2, x^3$
import matplotlib.pyplot as plt
import numpy as np

import matplotlib.cbook as cbook
fname = cbook.get_sample_data('msft.csv', asfileobj=False)
fname2 = cbook.get_sample_data('data_x_x2_x3.csv', asfileobj=False)

# test 1; use ints
plt.plotfile(fname, (0, 5, 6))

# test 2; use names
plt.plotfile(fname, ('date', 'volume', 'adj_close'))

# test 3; use semilogy for volume
plt.plotfile(fname, ('date', 'volume', 'adj_close'),
             plotfuncs={'volume': 'semilogy'})

# test 4; use semilogy for volume
plt.plotfile(fname, (0, 5, 6), plotfuncs={5: 'semilogy'})

# test 5; single subplot
plt.plotfile(fname, ('date', 'open', 'high', 'low', 'close'), subplots=False)

# test 6; labeling, if no names in csv-file
plt.plotfile(fname2, cols=(0, 1, 2), delimiter=' ')
names=['$x$', '$f(x)=x^2$', '$f(x)=x^3$'])

# test 7; more than one file per figure--illustrated here with a single file
plt.plotfile(fname2, cols=(0, 1), delimiter=' ')
plt.plotfile(fname2, cols=(0, 2), newfig=False, delimiter=' ')  # use current figure
plt.xlabel(r'$x$')
plt.ylabel(r'$f(x) = x^2, x^3$')

# test 8; use bar for volume
plt.plotfile(fname, (0, 5, 6), plotfuncs={5: 'bar'})

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

99.147 pylab_examples example code: polar_demo.py
import matplotlib.pyplot as plt

r = np.arange(0, 2, 0.01)
theta = 2 * np.pi * r

ax = plt.subplot(111, projection='polar')
ax.plot(theta, r)
ax.set_rmax(2)
ax.set_rticks([0.5, 1, 1.5, 2])  # less radial ticks
ax.set_rlabel_position(-22.5)  # get radial labels away from plotted line
ax.grid(True)

ax.set_title('A line plot on a polar axis', va='bottom')
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
import numpy as np
from matplotlib.pyplot import figure, show, rc

# radar green, solid grid lines
rc('grid', color='#316931', linewidth=1, linestyle='-')
rc('xtick', labelsize=15)
rc('ytick', labelsize=15)

# force square figure and square axes looks better for polar, IMO
fig = figure(figsize=(8, 8))
ax = fig.add_axes([0.1, 0.1, 0.8, 0.8],
                   projection='polar', facecolor='#d5de9c')

r = np.arange(0, 3.0, 0.01)
theta = 2*np.pi*r
ax.plot(theta, r, color='ee8d18', lw=3, label='a line')
ax.plot(0.5*theta, r, color='blue', ls='--', lw=3, label='another line')
ax.legend()
show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

99.149  

pylab_examples example code: print_stdout.py

[source code]

# -*- noplot -*-
# print png to standard out
# usage: python print_stdout.py > somefile.png

import sys
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt

plt.plot([1, 2, 3])

if sys.version_info[0] >= 3:
    plt.savefig(sys.stdout.buffer)
else:
    plt.savefig(sys.stdout)

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
import matplotlib.pyplot as plt
import numpy as np

dt = 0.01
t = np.arange(0, 10, dt)
nse = np.random.randn(len(t))
r = np.exp(-t/0.05)

cnse = np.convolve(nse, r)*dt
ecnse = cnse[:len(t)]
s = 0.1*np.sin(2*np.pi*t) + cnse

plt.subplot(211)
plt.plot(t, s)
plt.subplot(212)
plt.psd(s, 512, 1/dt)
plt.show()
```python
t = [0:dt:10];
nse = randn(size(t));
r = exp(-t/0.05);
cnse = conv(nse, r)*dt;
cnse = cnse(1:length(t));
s = 0.1*sin(2*pi*t) + cnse;

subplot(211)
plot(t,s)

subplot(212)
psd(s, 512, 1/dt)
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

99.151  **pylab_examples example code: psd_demo2.py**

```python
# This example shows the effects of some of the different PSD parameters
import numpy as np
import matplotlib.pyplot as plt

# This example shows the effects of some of the different PSD parameters
import numpy as np
import matplotlib.pyplot as plt
```
dt = np.pi / 100.
fs = 1. / dt
t = np.arange(0, 8, dt)
y = 10. * np.sin(2 * np.pi * 4 * t) + 5. * np.sin(2 * np.pi * 4.25 * t)
y = y + np.random.randn(*t.shape)

# Plot the raw time series
fig = plt.figure()
fig.subplots_adjust(hspace=0.45, wspace=0.3)
ax = fig.add_subplot(2, 1, 1)
ax.plot(t, y)

# Plot the PSD with different amounts of zero padding. This uses the entire time series at once
ax2 = fig.add_subplot(2, 3, 4)
ax2.psd(y, NFFT=len(t), pad_to=len(t), Fs=fs)
ax2.psd(y, NFFT=len(t), pad_to=len(t)**2, Fs=fs)
ax2.psd(y, NFFT=len(t), pad_to=len(t)**4, Fs=fs)
plt.title('zero padding')

# Plot the PSD with different block sizes, Zero pad to the length of the original data sequence.
ax3 = fig.add_subplot(2, 3, 5, sharex=ax2, sharey=ax2)
ax3.psd(y, NFFT=len(t), pad_to=len(t), Fs=fs)
ax3.psd(y, NFFT=len(t)//2, pad_to=len(t), Fs=fs)
ax3.psd(y, NFFT=len(t)//4, pad_to=len(t), Fs=fs)
ax3.set_ylabel('')
plt.title('block size')

# Plot the PSD with different amounts of overlap between blocks
ax4 = fig.add_subplot(2, 3, 6, sharex=ax2, sharey=ax2)
ax4.psd(y, NFFT=len(t)//2, pad_to=len(t), noverlap=0, Fs=fs)
ax4.psd(y, NFFT=len(t)//2, pad_to=len(t), noverlap=int(0.05*len(t)/2.), Fs=fs)
ax4.psd(y, NFFT=len(t)//2, pad_to=len(t), noverlap=int(0.2*len(t)/2.), Fs=fs)
ax4.set_ylabel('')
plt.title('overlap')

plt.show()
This is a ported version of a MATLAB example from the signal processing toolbox that showed some difference at one time between Matplotlib's and MATLAB's scaling of the PSD.

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.mlab as mlab

prng = np.random.RandomState(123456)  # to ensure reproducibility

fs = 1000

A = np.array((2, 8)).reshape(-1, 1)
B = np.array((150, 140)).reshape(-1, 1)

x = (A * np.sin(2 * np.pi * f * t)).sum(axis=0) + 5 * prng.randn(*t.shape)

fig, (ax0, ax1) = plt.subplots(ncols=2)

fig.subplots_adjust(hspace=0.45, wspace=0.3)

yticks = np.arange(-50, 30, 10)

fig.subplots_adjust(hspace=0.45, wspace=0.3)
yticks = np.arange(-50, 30, 10)
```
Matplotlib, Release 2.0.2

```python
yrange = (yticks[0], yticks[-1])
xticks = np.arange(0, 550, 100)

ax0.psd(xn, NFFT=301, Fs=fs, window=mlab.window_none, pad_to=1024,
        scale_by_freq=True)
ax0.set_title('Periodogram')
ax0.set_yticks(yticks)
ax0.set_xticks(xticks)
ax0.grid(True)
ax0.set_ylim(yrange)

ax1.psd(xn, NFFT=150, Fs=fs, window=mlab.window_none, pad_to=512, noverlap=75,
        scale_by_freq=True)
ax1.set_title('Welch')
ax1.set_xticks(xticks)
ax1.set_yticks(yticks)
ax1.set_ylabel('')  # overwrite the y-label added by `psd`
ax1.grid(True)
ax1.set_ylim(yrange)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
This is a ported version of a MATLAB example from the signal processing toolbox that showed some difference at one time between Matplotlib's and MATLAB's scaling of the PSD.

This differs from psd_demo3.py in that this uses a complex signal, so we can see that complex PSD's work properly.

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.mlab as mlab

prng = np.random.RandomState(123456)  # to ensure reproducibility

fs = 1000
f = np.linspace(0, 0.3, 301)
A = np.array([2, 8]).reshape(-1, 1)
f = np.array([150, 140]).reshape(-1, 1)
xn = (A * np.exp(2j * np.pi * f * t)).sum(axis=0) + 5 * prng.randn(*t.shape)

fig, (ax0, ax1) = plt.subplots(ncols=2)
```
```python
fig.subplots_adjust(hspace=0.45, wspace=0.3)
yticks = np.arange(-50, 30, 10)
yrange = (yticks[0], yticks[-1])
xticks = np.arange(-500, 550, 200)

ax0.psd(xn, NFFT=301, Fs=fs, window=mlab.window_none, pad_to=1024,
        scale_by_freq=True)
ax0.set_title('Periodogram')
ax0.set_xticks(xticks)
ax0.set_yticks(yticks)
ax0.grid(True)
ax0.set_ylim(yrange)

ax1.psd(xn, NFFT=150, Fs=fs, window=mlab.window_none, pad_to=512, noverlap=75,
        scale_by_freq=True)
ax1.set_title('Welch')
ax1.set_xticks(xticks)
ax1.set_yticks(yticks)
ax1.set_ylabel('')  # overwrite the y-label added by 'psd'
ax1.grid(True)
ax1.set_ylim(yrange)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Some people prefer to write more pythonic, object-oriented code rather than use the pyplot interface to matplotlib. This example shows you how.

Unless you are an application developer, I recommend using part of the pyplot interface, particularly the figure, close, subplot, axes, and show commands. These hide a lot of complexity from you that you don't need to see in normal figure creation, like instantiating DPI instances, managing the bounding boxes of the figure elements, creating and realizing GUI windows and embedding figures in them.

If you are an application developer and want to embed matplotlib in your application, follow the lead of examples/embedding_in_wx.py, examples/embedding_in_gtk.py or examples/embedding_in_tk.py. In this case you will want to control the creation of all your figures, embedding them in application windows, etc.

If you are a web application developer, you may want to use the example in webapp_demo.py, which shows how to use the backend agg figure canvas directly, with none of the globals (current figure,
current axes) that are present in the pyplot interface. Note that there is no reason why the pyplot interface won't work for web application developers, however.

If you see an example in the examples dir written in pyplot interface, and you want to emulate that using the true python method calls, there is an easy mapping. Many of those examples use 'set' to control figure properties. Here's how to map those commands onto instance methods

The syntax of set is

\[
\text{plt.setp(object or sequence, somestring, attribute)}
\]

if called with an object, set calls

\[
\text{object.set_somestring(attribute)}
\]

if called with a sequence, set does

\[
\text{for object in sequence:}
\]
\[
\text{\hspace{1em}object.set_somestring(attribute)}
\]

So for your example, if a is your axes object, you can do

\[
\text{a.set_xticklabels([])}
\]
\[
\text{a.set_yticklabels([])}
\]
\[
\text{a.set_xticks([])}
\]
\[
\text{a.set_yticks([])}
\]

```python
from matplotlib.pyplot import figure, show
from numpy import arange, sin, pi

t = arange(0.0, 1.0, 0.01)
fig = figure(1)

ax1 = fig.add_subplot(211)
ax1.plot(t, sin(2*pi*t))
a1.grid(True)
a1.set_ylim((-2, 2))
a1.set_ylabel('1 Hz')
a1.set_title('A sine wave or two')

for label in ax1.get_xticklabels():
    label.set_color('r')

ax2 = fig.add_subplot(212)
ax2.plot(t, sin(2*2*pi*t))
a2.grid(True)
```

"""
ax2.set_ylim((-2, 2))
l = ax2.set_xlabel('Hi mom')
l.set_color('g')
l.set_fontsize('large')
show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

99.155  *pylab_examples* example code: *quadmesh_demo.py*

````
""
``pcolormesh uses a QuadMesh, a faster generalization of pcolor, but
with some restrictions.

This demo illustrates a bug in quadmesh with masked data.
""
`````nt

```python
import numpy as np
from matplotlib.pyplot import figure, show, savefig
from matplotlib import cm, colors
from numpy import ma
```

99.155.  *pylab_examples* example code: *quadmesh_demo.py*
n = 12
x = np.linspace(-1.5, 1.5, n)
y = np.linspace(-1.5, 1.5, n*2)
X, Y = np.meshgrid(x, y)
Qx = np.cos(Y) - np.cos(X)
Qz = np.sin(Y) + np.sin(X)
Qx = (Qx + 1.1)
Z = np.sqrt(X**2 + Y**2)/5
Z = (Z - Z.min()) / (Z.max() - Z.min())

# The color array can include masked values:
Zm = ma.masked_where(np.fabs(Qz) < 0.5*np.amax(Qz), Z)

fig = figure()
ax = fig.add_subplot(121)
ax.pcolormesh(Qx, Qz, Z, shading='gouraud')
ax.set_title('Without masked values')

ax = fig.add_subplot(122)
# You can control the color of the masked region:
# cmap = cm.RdBu
# cmap.set_bad('y', 1.0)
# ax.pcolormesh(Qx, Qz, Zm, cmap=cmap)
# Or use the default, which is transparent:
col = ax.pcolormesh(Qx, Qz, Zm, shading='gouraud')
ax.set_title('With masked values')

show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Arrows scale with plot width, not view
```python
import matplotlib.pyplot as plt
import numpy as np
from numpy import ma

X, Y = np.meshgrid(np.arange(0, 2 * np.pi, .2), np.arange(0, 2 * np.pi, .2))
U = np.cos(X)
V = np.sin(Y)

plt.figure()
plt.title('Arrows scale with plot width, not view')
Q = plt.quiver(X, Y, U, V, units='width')
qk = plt.quiverkey(Q, 0.9, 0.9, 2, r'$2 \frac{m}{s}$', labelpos='E', coordinates='figure')

plt.figure()
plt.title("pivot='mid'; every third arrow; units='inches'")
```

Known problem: the plot autoscaling does not take into account the arrows, so those on the boundaries are often out of the picture. This is *not* an easy problem to solve in a perfectly general way. The workaround is to manually expand the axes.
Q = plt.quiver(X[::3, ::3], Y[::3, ::3], U[::3, ::3], V[::3, ::3],
                 pivot='mid', units='inches')
qk = plt.quiverkey(Q, 0.9, 0.9, 1, r'$1 \frac{m}{s}$', labelpos='E',
                   coordinates='figure')
plt.scatter(X[::3, ::3], Y[::3, ::3], color='r', s=5)

plt.figure()
plt.title("pivot='tip'; scales with x view")
M = np.hypot(U, V)
Q = plt.quiver(X, Y, U, V, M, units='x', pivot='tip', width=0.022,
               scale=1 / 0.15)
qk = plt.quiverkey(Q, 0.9, 0.9, 1, r'$1 \frac{m}{s}$', labelpos='E',
                   coordinates='figure')
plt.scatter(X, Y, color='k', s=5)
plt.show()
```python
import matplotlib.pyplot as plt
import numpy as np

X = np.arange(-10, 10, 1)
Y = np.arange(-10, 10, 1)
U, V = np.meshgrid(X, Y)

fig, ax = plt.subplots()
q = ax.quiver(X, Y, U, V)
ax.quiverkey(q, X=0.3, Y=1.1, U=10,
             label='Quiver key, length = 10', labelpos='E')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

99.158 pylab_examples example code: scatter_custom_symbol.py
```python
import matplotlib.pyplot as plt
from numpy import arange, pi, cos, sin
from numpy.random import rand

# unit area ellipse
rx, ry = 3., 1.
area = rx * ry * pi
theta = arange(0, 2*pi + 0.01, 0.1)
verts = list(zip(rx/area*cos(theta), ry/area*sin(theta)))

x, y, s, c = rand(4, 30)
s *= 1e2

fig, ax = plt.subplots()
ax.scatter(x, y, s, c, marker=None, verts=verts)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

99.159 pylab_examples example code: scatter_demo2.py
Demo of scatter plot with varying marker colors and sizes.

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cbook as cbook

# Load a numpy record array from yahoo csv data with fields date, open, close, volume, adj_close from the mpl-data/example directory.
# The record array stores python datetime.date as an object array in the date column
datafile = cbook.get_sample_data('goog.npy')
try:
    # Python3 cannot load python2 .npy files with datetime(object) arrays
    # unless the encoding is set to bytes. However this option was not added until numpy 1.10 so this example will only work with python 2 or with numpy 1.10 and later
    price_data = np.load(datafile, encoding='bytes').view(np.recarray)
except TypeError:
    price_data = np.load(datafile).view(np.recarray)
price_data = price_data[-250:]
# get the most recent 250 trading days
delta1 = np.diff(price_data.adj_close)/price_data.adj_close[:-1]

# Marker size in units of points^2
volume = (15 * price_data.volume[:-2] / price_data.volume[0])**2
close = 0.003 * price_data.close[:-2] / 0.003 * price_data.open[:-2]

fig, ax = plt.subplots()
ax.scatter(delta1[:-1], delta1[1:], c=close, s=volume, alpha=0.5)
ax.set_xlabel(r'$\Delta_i$', fontsize=15)
ax.set_ylabel(r'$\Delta_{i+1}$', fontsize=15)
ax.set_title('Volume and percent change')
ax.grid(True)
fig.tight_layout()
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.ticker import NullFormatter

# the random data
x = np.random.randn(1000)
y = np.random.randn(1000)

nullfmt = NullFormatter()  # no labels
# definitions for the axes
left, width = 0.1, 0.65
bottom, height = 0.1, 0.65
bottom_h = left_h = left + width + 0.02
rect_scatter = [left, bottom, width, height]
rect_histx = [left, bottom_h, width, 0.2]
rect_histy = [left_h, bottom, 0.2, height]

# start with a rectangular Figure
plt.figure(1, figsize=(8, 8))

axScatter = plt.axes(rect_scatter)
axHistx = plt.axes(rect_histx)
axHisty = plt.axes(rect_histy)

# no labels
axHistx.xaxis.set_major_formatter(nullfmt)
axHisty.yaxis.set_major_formatter(nullfmt)

# the scatter plot:
axScatter.scatter(x, y)

# now determine nice limits by hand:
binwidth = 0.25
xymax = np.max([np.max(np.fabs(x)), np.max(np.fabs(y))])
lim = (int(xymax/binwidth) + 1) * binwidth

axScatter.set_xlim((-lim, lim))
axScatter.set_ylim((-lim, lim))

bins = np.arange(-lim, lim + binwidth, binwidth)
axHistx.hist(x, bins=bins)
axHisty.hist(y, bins=bins, orientation='horizontal')

axHistx.set_xlim(axScatter.get_xlim())
axHisty.set_ylim(axScatter.get_ylim())

plt.show()
```python
import matplotlib.pyplot as plt
import numpy as np

N = 100
r0 = 0.6
x = 0.9*np.random.rand(N)
y = 0.9*np.random.rand(N)
area = np.pi*(10*np.random.rand(N))**2  # 0 to 10 point radii
r = np.sqrt(x*x + y*y)
c = np.sqrt(area)
area1 = np.ma.masked_where(r < r0, area)
area2 = np.ma.masked_where(r >= r0, area)
plt.scatter(x, y, s=area1, marker='^', c=c)
plt.scatter(x, y, s=area2, marker='o', c=c)
plt.plot(r0*np.cos(theta), r0*np.sin(theta))
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
import matplotlib.pyplot as plt
import numpy as np
import time

for N in (20, 100, 1000, 10000, 50000):
    tstart = time.time()
    x = 0.9*np.random.rand(N)
    y = 0.9*np.random.rand(N)
    s = 20*np.random.rand(N)
    plt.scatter(x, y, s)
    print('%d symbols in %1.2f s' % (N, time.time() - tstart))
import numpy as np
import matplotlib.pyplot as plt

x = np.random.rand(10)
y = np.random.rand(10)
z = np.sqrt(x**2 + y**2)

plt.subplot(321)
plt.scatter(x, y, s=80, c=z, marker=">")

plt.subplot(322)
plt.scatter(x, y, s=80, c=z, marker=(5, 0))

verts = list(zip([-1., 1., 1., -1.], [-1., -1., 1., -1.]))
plt.subplot(323)
plt.scatter(x, y, s=80, c=z, marker=(verts, 0))

# equivalent:
# plt.scatter(x, y, s=80, c=z, marker=None, verts=verts)
```python
plt.subplot(324)
plt.scatter(x, y, s=80, c=z, marker=(5, 1))

plt.subplot(325)
plt.scatter(x, y, s=80, c=z, marker='+')

plt.subplot(326)
plt.scatter(x, y, s=80, c=z, marker=(5, 2))
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

**99.164 pylab_examples example code: scatter_symbol.py**

```python
from matplotlib import pyplot as plt
import numpy as np
import matplotlib

x = np.arange(0.0, 50.0, 2.0)
y = x ** 1.3 + np.random.rand(*x.shape) * 30.0
```
s = np.random.rand(*x.shape) * 800 + 500
plt.scatter(x, y, s, c="g", alpha=0.5, marker=r'$\clubsuit$',
            label="Luck")
plt.xlabel("Leprechauns")
plt.ylabel("Gold")
plt.legend(loc=2)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

99.165 pylab_examples example code: set_and_get.py

"""
The pyplot interface allows you to use setp and getp to set and get object properties, as well as to do introspection on the object set:

To set the linestyle of a line to be dashed, you can do

>>> line, = plt.plot([1,2,3])
"""
```python
>>> plt.setp(line, linestyle='--')

If you want to know the valid types of arguments, you can provide the name of the property you want to set without a value

```4
```python
>>> plt.setp(line, linestyle)
linestyle: ['-' | '--' | ':']
```

If you want to see all the properties that can be set, and their possible values, you can do

```python
>>> plt.setp(line)
```

set operates on a single instance or a list of instances. If you are in query mode introspecting the possible values, only the first instance in the sequence is used. When actually setting values, all the instances will be set. e.g., suppose you have a list of two lines, the following will make both lines thicker and red

```python
>>> x = np.arange(0,1.0,0.01)
>>> y1 = np.sin(2*np.pi*x)
>>> y2 = np.sin(4*np.pi*x)
>>> lines = plt.plot(x, y1, x, y2)
```4
>>> plt.setp(lines, linewidth=2, color='r')
```

get:

get returns the value of a given attribute. You can use get to query the value of a single attribute

```python
>>> plt.getp(line, 'linewidth')
0.5
```
or all the attribute/value pairs

```python
>>> plt.getp(line)
```
```
aa = True
alpha = 1.0
antialiased = True
c = b
clip_on = True
color = b
... long listing skipped ...
```

Aliases:

To reduce keystrokes in interactive mode, a number of properties have short aliases, e.g., 'lw' for 'linewidth' and 'mec' for 'markeredgecolor'. When calling set or get in introspection mode, these properties will be listed as 'fullname or aliasname', as in
from __future__ import print_function

import matplotlib.pyplot as plt
import numpy as np

x = np.arange(0, 1.0, 0.01)
y1 = np.sin(2*np.pi*x)
y2 = np.sin(4*np.pi*x)
lines = plt.plot(x, y1, x, y2)
l1, l2 = lines
plt.setp(lines, linestyle='--')  # set both to dashed
plt.setp(l1, linewidth=2, color='r')  # line1 is thick and red
plt.setp(l2, linewidth=1, color='g')  # line2 is thinner and green

print('Line setters')
plt.setp(l1)
print('Line getters')
plt.getp(l1)

print('Rectangle setters')
plt.setp(plt.gca().patch)
print('Rectangle getters')
plt.getp(plt.gca().patch)


t = plt.title('Hi mom')
print('Text setters')
plt.setp(t)
print('Text getters')
plt.getp(t)

plt.show()
HSV Blending Looks Best with Smooth Surfaces

Colormapped Data

Illumination Intensity

Blend Mode: "hsv" (default)

Blend Mode: "overlay"
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import LightSource
from matplotlib.cbook import get_sample_data

# Example showing how to make shaded relief plots
# like Mathematica
# (http://reference.wolfram.com/mathematica/ref/ReliefPlot.html)
# or Generic Mapping Tools

def main():
    # Test data
    x, y = np.mgrid[-5:5:0.05, -5:5:0.05]
    z = 5 * (np.sqrt(x**2 + y**2) + np.sin(x**2 + y**2))

    filename = get_sample_data('jacksboro_fault_dem.npz', asfileobj=False)
    with np.load(filename) as dem:
        elev = dem['elevation']

    fig = compare(z, plt.cm.copper)
    fig.suptitle('HSV Blending Looks Best with Smooth Surfaces', y=0.95)

    fig = compare(elev, plt.cm.gist_earth, ve=0.05)
    fig.suptitle('Overlay Blending Looks Best with Rough Surfaces', y=0.95)
```python
plt.show()

def compare(z, cmap, ve=1):
    # Create subplots and hide ticks
    fig, axes = plt.subplots(ncols=2, nrows=2)
    for ax in axes.flat:
        ax.set(xticks=[], yticks=[])  

    # Illuminate the scene from the northwest
    ls = LightSource(azdeg=315, altdeg=45)

    axes[0, 0].imshow(z, cmap=cmap)
    axes[0, 0].set(xlabel='Colormapped Data')

    axes[0, 1].imshow(ls.hillshade(z, vert_exag=ve), cmap='gray')
    axes[0, 1].set(xlabel='Illumination Intensity')

    rgb = ls.shade(z, cmap=cmap, vert_exag=ve, blend_mode='hsv')
    axes[1, 0].imshow(rgb)
    axes[1, 0].set(xlabel='Blend Mode: "hsv" (default)')

    rgb = ls.shade(z, cmap=cmap, vert_exag=ve, blend_mode='overlay')
    axes[1, 1].imshow(rgb)
    axes[1, 1].set(xlabel='Blend Mode: "overlay"')

    return fig

if __name__ == '__main__':
    main()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
99.167  

pylab_examples example code: shared_axis_across_figures.py

![Graph showing scattered data points ranging from 0 to 100 on the x-axis and 0 to 1.0 on the y-axis. The data points are distributed randomly across the chart.]
connect the data limits on the axes in one figure with the axes in another. This is not the right way to do this for two axes in the same figure -- use the sharex and sharey property in that case

```python
import numpy as np
import matplotlib.pyplot as plt

fig1 = plt.figure()
fig2 = plt.figure()

ax1 = fig1.add_subplot(111)
ax2 = fig2.add_subplot(111, sharex=ax1, sharey=ax1)

ax1.plot(np.random.rand(100), 'o')
ax2.plot(np.random.rand(100), 'o')

# In the latest release, it is no longer necessary to do anything special to share axes across figures:

# ax1.sharex_foreign(ax2)
# ax2.sharex_foreign(ax1)
# ax1.sharey_foreign(ax2)
# ax2.sharey_foreign(ax1)
```
You can share the x or y axis limits for one axis with another by passing an axes instance as a sharex or sharey kwarg.

Changing the axis limits on one axes will be reflected automatically in the other, and vice-versa, so when you navigate with the toolbar the axes will follow each other on their shared axes. Ditto for changes in the axis scaling (e.g., log vs linear). However, it is possible to have differences in tick labeling, e.g., you can selectively turn off the tick labels on one axes.

The example below shows how to customize the tick labels on the various axes. Shared axes share the tick locator, tick formatter, view limits, and transformation (e.g., log, linear). But the ticklabels themselves do not share properties. This is a feature and not a bug, because you may want to make the tick labels smaller on the upper
axes, e.g., in the example below.

If you want to turn off the ticklabels for a given axes (e.g., on subplot(211) or subplot(212), you cannot do the standard trick

```python
setp(ax2, xticklabels=[])```

because this changes the tick Formatter, which is shared among all axes. But you can alter the visibility of the labels, which is a property

```python
setp(ax2.get_xticklabels(), visible=False)
```

```python
import matplotlib.pyplot as plt
import numpy as np

t = np.arange(0.01, 5.0, 0.01)
s1 = np.sin(2*np.pi*t)
s2 = np.exp(-t)
s3 = np.sin(4*np.pi*t)

ax1 = plt.subplot(311)
plt.plot(t, s1)
plt.setp(ax1.get_xticklabels(), fontsize=6)

# share x only
ax2 = plt.subplot(312, sharex=ax1)
plt.plot(t, s2)
plt.setp(ax2.get_xticklabels(), visible=False)

# share x and y
ax3 = plt.subplot(313, sharex=ax1, sharey=ax1)
plt.plot(t, s3)
plt.xlim(0.01, 5.0)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
import matplotlib.pyplot as plt
import numpy as np

t = np.arange(0.0, 2.0, 0.01)
s = 1 + np.sin(2*np.pi*t)
plt.plot(t, s)

plt.xlabel('time (s)')
plt.ylabel('voltage (mV)')
plt.title('About as simple as it gets, folks')
plt.grid(True)
plt.savefig("test.png")
plt.show()
import matplotlib.pyplot as plt
import numpy as np

dt = 0.0005
t = np.arange(0.0, 20.0, dt)
s1 = np.sin(2*np.pi*100*t)
s2 = 2*np.sin(2*np.pi*400*t)

# create a transient "chirp"
mask = np.where(np.logical_and(t > 10, t < 12), 1.0, 0.0)
s2 = s2 * mask

# add some noise into the mix
nse = 0.01*np.random.random(size=len(t))

x = s1 + s2 + nse  # the signal

NFFT = 1024  # the length of the windowing segments
Fs = int(1.0/dt)  # the sampling frequency

# Pxx is the segments x freqs array of instantaneous power, freqs is
# the frequency vector, bins are the centers of the time bins in which
# the power is computed, and im is the matplotlib.image.AxesImage

---

99.170. *pylab_examples* example code: *specgram_demo.py*
# instance

```python
ax1 = plt.subplot(211)
plt.plot(t, x)
plt.subplot(212, sharex=ax1)
Pxx, freqs, bins, im = plt.specgram(x, NFFT=NFFT, Fs=Fs, noverlap=noverlap)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

99.171 pylab_examples example code: spectrum_demo.py

```python
import matplotlib.pyplot as plt
import numpy as np

np.random.seed(0)

dt = 0.01
Fs = 1/dt
t = np.arange(0, 10, dt)
nse = np.random.randn(len(t))
```
```python
r = np.exp(-t/0.05)
cnse = np.convolve(nse, r)*dt
cnse = cnse[:len(t)]
s = 0.1*np.sin(2* np.pi*t) + cnse
plt.subplot(3, 2, 1)
plt.plot(t, s)
plt.subplot(3, 2, 3)
plt.magnitude_spectrum(s, Fs=Fs)
plt.subplot(3, 2, 4)
plt.magnitude_spectrum(s, Fs=Fs, scale='dB')
plt.subplot(3, 2, 5)
plt.angle_spectrum(s, Fs=Fs)
plt.subplot(3, 2, 6)
plt.phase_spectrum(s, Fs=Fs)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
99.172  *pylab_examples* example code: *spine_placement_demo.py*

centered spines

spines at axes (0.6, 0.1)

zeroed spines

spines at data (1, 2)
```python
import numpy as np
import matplotlib.pyplot as plt

g = plt.figure()
x = np.linspace(-np.pi, np.pi, 100)
y = 2*np.sin(x)

ax = g.add_subplot(2, 2, 1)
ax.set_title('centered spines')
ax.plot(x, y)
ax.spines['left'].set_position('center')
ax.spines['right'].set_color('none')
ax.spines['bottom'].set_position('center')
ax.spines['top'].set_color('none')
ax.spines['left'].set_smart_bounds(True)
ax.spines['bottom'].set_smart_bounds(True)
ax.xaxis.set_ticks_position('bottom')
ax.yaxis.set_ticks_position('left')

ax = g.add_subplot(2, 2, 2)
ax.set_title('zeroed spines')
ax.plot(x, y)
ax.spines['left'].set_position('zero')
ax.spines['right'].set_color('none')
ax.spines['bottom'].set_position('zero')
```
ax.spines['top'].set_color('none')
ax.spines['left'].set_smart_bounds(True)
ax.spines['bottom'].set_smart_bounds(True)
ax.xaxis.set_ticks_position('bottom')
ax.yaxis.set_ticks_position('left')

ax = fig.add_subplot(2, 2, 3)
ax.set_title('spines at axes (0.6, 0.1)')
ax.plot(x, y)
ax.spines['left'].set_position(('axes', 0.6))
ax.spines['right'].set_color('none')
ax.spines['bottom'].set_position(('axes', 0.1))
ax.spines['top'].set_color('none')
ax.spines['left'].set_smart_bounds(True)
ax.spines['bottom'].set_smart_bounds(True)
ax.xaxis.set_ticks_position('bottom')
ax.yaxis.set_ticks_position('left')

ax = fig.add_subplot(2, 2, 4)
ax.set_title('spines at data (1, 2)')
ax.plot(x, y)
ax.spines['left'].set_position(('data', 1))
ax.spines['right'].set_color('none')
ax.spines['bottom'].set_position(('data', 2))
ax.spines['top'].set_color('none')
ax.spines['left'].set_smart_bounds(True)
ax.spines['bottom'].set_smart_bounds(True)
ax.xaxis.set_ticks_position('bottom')
ax.yaxis.set_ticks_position('left')

# ----------------------------------------------------

def adjust_spines(ax, spines):
    for loc, spine in ax.spines.items():
        if loc in spines:
            spine.set_position(('outward', 10))  # outward by 10 points
            spine.set_smart_bounds(True)
        else:
            spine.set_color('none')  # don't draw spine

    # turn off ticks where there is no spine
    if 'left' in spines:
        ax.yaxis.set_ticks_position('left')
    else:
        # no yaxis ticks
        ax.yaxis.set_ticks([])
    if 'bottom' in spines:
        ax.xaxis.set_ticks_position('bottom')
    else:
        # no xaxis ticks
        ax.xaxis.set_ticks([])
fig = plt.figure()

x = np.linspace(0, 2*np.pi, 100)
y = 2*np.sin(x)

ax = fig.add_subplot(2, 2, 1)
ax.plot(x, y, clip_on=False)
adjust_spines(ax, ['left'])

ax = fig.add_subplot(2, 2, 2)
ax.plot(x, y, clip_on=False)
adjust_spines(ax, [])

ax = fig.add_subplot(2, 2, 3)
ax.plot(x, y, clip_on=False)
adjust_spines(ax, ['left', 'bottom'])

ax = fig.add_subplot(2, 2, 4)
ax.plot(x, y, clip_on=False)
adjust_spines(ax, ['bottom'])

plt.show()
Plot the sparsity pattern of arrays

```python
from matplotlib.pyplot import figure, show
import numpy

fig = figure()
ax1 = fig.add_subplot(221)
ax2 = fig.add_subplot(222)
ax3 = fig.add_subplot(223)
ax4 = fig.add_subplot(224)

x = numpy.random.randn(20, 20)
x[5] = 0.
x[:, 12] = 0.

ax1.spy(x, markersize=5)
ax2.spy(x, precision=0.1, markersize=5)
ax3.spy(x)
ax4.spy(x, precision=0.1)
```
Keywords: python, matplotlib, pylab, example, codex (see Search examples)

99.174  pylab_examples example code: stackplot_demo.py
import numpy as np
import matplotlib.pyplot as plt

def fnx():
    return np.random.randint(5, 50, 10)
y = np.row_stack((fnx(), fnx(), fnx()))
x = np.arange(10)
y1, y2, y3 = fnx(), fnx(), fnx()

fig, ax = plt.subplots()
ax.stackplot(x, y)
plt.show()

fig, ax = plt.subplots()
ax.stackplot(x, y1, y2, y3)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(0)

def layers(n, m):
    """Return *n* random Gaussian mixtures, each of length *m*.
    """
    def bump(a):
        x = 1 / (.1 + np.random.random())
        y = 2 * np.random.random() - .5
        z = 10 / (.1 + np.random.random())
        for i in range(m):
            w = (i / float(m) - y) * z
            a[i] += x * np.exp(-w * w)
    a = np.zeros((m, n))
    for i in range(n):
        for j in range(5):
            bump(a[:, i])
    return a
```python
d = layers(3, 100)
plt.subplots()
plt.stackplot(range(100), d.T, baseline='wiggle')
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

99.176  

pylab_examples example code: stem_plot.py

```python
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0.1, 2*np.pi, 10)
markerline, stemlines, baseline = plt.stem(x, np.cos(x), '-.'
plt.setp(baseline, 'color', 'r', 'linewidth', 2)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
import numpy as np
from numpy import ma
import matplotlib.pyplot as plt

x = np.arange(1, 7, 0.4)
y0 = np.sin(x)
y = y0.copy() + 2.5

plt.step(x, y, label='pre (default)')
y -= 0.5
plt.step(x, y, where='mid', label='mid')
y -= 0.5
plt.step(x, y, where='post', label='post')
y = ma.masked_where((y0 > -0.15) & (y0 < 0.15), y - 0.5)
plt.step(x, y, label='masked (pre)')

plt.legend()
plt.xlim(0, 7)
plt.ylim(-0.5, 4)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
99.178  pylab_examples example code: stix_fonts_demo.py

```
①②③①②③①②③

SansΩSansΩSansΩ

Monospace

CALLIGRAPHIC

Blackboardπ

Blackboardπ

FrakturFraktur

Script
```
from __future__ import unicode_literals

import os
import sys
import re
import gc
import matplotlib.pyplot as plt
import numpy as np

stests = [
    r'$\mathcircled{123} \mathrm{\mathcircled{123}}$',
    r'\mathbf{\mathcircled{123}}$,
    r'$\mathsf{Sans \Omega} \mathrm{\mathsf{Sans \Omega}}$',
    r'$\mathbf{\mathsf{Sans \Omega}}$',
    r'$\mathtt{Monospace}$',
    r'$\mathcal{CALLIGRAPHIC}$$',
    r'$\mathbb{Blackboard \pi}$',
    r'$\mathrm{\mathbb{Blackboard \pi}}$',
    r'$\mathbf{\mathbb{Blackboard \pi}}$',
    r'$\mathfrak{Fraktur}$ \mathbf{\mathfrak{Fraktur}}$',
    r'$\mathscr{Script}$']

if sys.maxunicode > 0xffff:
    s = r'\Direct Unicode: $\u23ce \mathrm{\ue0f2 \U0001D538}$'

def doall():
    tests = stests

    plt.figure(figsize=(8, (len(tests) * 1) + 2))
    plt.plot([0, 0], 'r')
    plt.grid(False)
    plt.axis([0, 3, -len(tests), 0])
    plt.xticks(np.arange(len(tests)) * -1)
    for i, s in enumerate(tests):
        plt.text(0.1, -i, s, fontsize=32)

    plt.savefig('stix_fonts_example')
    plt.show()

if '--latex' in sys.argv:
    fd = open('stix_fonts_examples.ltx', 'w')
    fd.write(r'\documentclass{article}
    \begin{document}
    \begin{enumerate}
    for i, s in enumerate(stests):
        s = re.sub(r'(?<!\$)$', '$$', s)
        fd.write(r'\item %s
        % s)
    fd.write(r'\end{enumerate}
    \end{document}
    """
}
import numpy as np
import matplotlib.pyplot as plt

x1 = np.linspace(0.0, 5.0)
x2 = np.linspace(0.0, 2.0)

y1 = np.cos(2 * np.pi * x1) * np.exp(-x1)
y2 = np.cos(2 * np.pi * x2)

A tale of 2 subplots

Simple demo with multiple subplots.
```python
plt.subplot(2, 1, 1)
plt.plot(x1, y1, 'ko-')
plt.title('A tale of 2 subplots')
plt.ylabel('Damped oscillation')

plt.subplot(2, 1, 2)
plt.plot(x2, y2, 'r.-')
plt.xlabel('time (s)')
plt.ylabel('Undamped')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

99.180 pylab_examples example code: subplot_toolbar.py
import matplotlib.pyplot as plt
import numpy.random as rnd

fig = plt.figure()
plt.subplot(221)
plt.imshow(rnd.random((100, 100)))
plt.subplot(222)
plt.imshow(rnd.random((100, 100)))
plt.subplot(223)
plt.imshow(rnd.random((100, 100)))
plt.subplot(224)
plt.imshow(rnd.random((100, 100)))

plt.subplot_tool()
plt.show()
import matplotlib.pyplot as plt
import numpy as np

plt.subplot(211)
plt.imshow(np.random.random((100, 100)), cmap=plt.cm.BuPu_r)
plt.subplot(212)
plt.imshow(np.random.random((100, 100)), cmap=plt.cm.BuPu_r)

plt.subplots_adjust(bottom=0.1, right=0.8, top=0.9)
cax = plt.axes([0.85, 0.1, 0.075, 0.8])
plt.colorbar(cax=cax)
plt.show()
Simple plot
Sharing Y axis

![Graph showing sharing of Y axis with two plots side by side.](image)
Sharing both axes

![Graph showing three functions sharing both axes](https://example.com/graph.png)
Sharing x per column, y per row

2710 Chapter 99. pylab_examples Examples
"""Examples illustrating the use of plt.subplots().

This function creates a figure and a grid of subplots with a single call, while providing reasonable control over how the individual plots are created. For very refined tuning of subplot creation, you can still use add_subplot() directly on a new figure.
"""

import matplotlib.pyplot as plt
import numpy as np

# Simple data to display in various forms
x = np.linspace(0, 2 * np.pi, 400)
y = np.sin(x ** 2)
plt.close('all')

# Just a figure and one subplot
f, ax = plt.subplots()
ax.plot(x, y)
ax.set_title('Simple plot')

# Two subplots, the axes array is 1-d
f, axarr = plt.subplots(2, sharex=True)
axarr[0].plot(x, y)
axarr[0].set_title('Sharing X axis')
Matplotlib, Release 2.0.2

```
axarr[1].scatter(x, y)

# Two subplots, unpack the axes array immediately
f, (ax1, ax2) = plt.subplots(1, 2, sharey=True)
ax1.plot(x, y)
ax1.set_title('Sharing Y axis')
ax2.scatter(x, y)

# Three subplots sharing both x/y axes
f, (ax1, ax2, ax3) = plt.subplots(3, sharex=True, sharey=True)
ax1.plot(x, y)
ax1.set_title('Sharing both axes')
ax2.scatter(x, y)
ax3.scatter(x, 2 * y ** 2 - 1, color='r')

# Fine-tune figure; make subplots close to each other and hide x ticks for
# all but bottom plot.
f.subplots_adjust(hspace=0)
plt.setp([a.get_xticklabels() for a in f.axes[:-1]], visible=False)

# row and column sharing
f, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, sharex='col', sharey='row')
ax1.plot(x, y)
ax1.set_title('Sharing x per column, y per row')
ax2.scatter(x, y)
ax3.scatter(x, 2 * y ** 2 - 1, color='r')
ax4.plot(x, 2 * y ** 2 - 1, color='r')

# Four axes, returned as a 2-d array
f, axarr = plt.subplots(2, 2)
axarr[0, 0].plot(x, y)
axarr[0, 0].set_title('Axis [0,0]')
axarr[0, 1].scatter(x, y)
axarr[0, 1].set_title('Axis [0,1]')
axarr[1, 0].plot(x, y ** 2)
axarr[1, 0].set_title('Axis [1,0]')
axarr[1, 1].scatter(x, y ** 2)
axarr[1, 1].set_title('Axis [1,1]')

# Fine-tune figure; hide x ticks for top plots and y ticks for right plots
plt.setp([a.get_xticklabels() for a in axarr[0, :]], visible=False)
plt.setp([a.get_yticklabels() for a in axarr[:, 1]], visible=False)

# Four polar axes
f, axarr = plt.subplots(2, 2, subplot_kw=dict(projection='polar'))
axarr[0, 0].plot(x, y)
axarr[0, 0].set_title('Axis [0,0]')
axarr[0, 1].scatter(x, y)
axarr[0, 1].set_title('Axis [0,1]')
axarr[1, 0].plot(x, y ** 2)
axarr[1, 0].set_title('Axis [1,0]')
axarr[1, 1].scatter(x, y ** 2)
axarr[1, 1].set_title('Axis [1,1]')

# Fine-tune figure; make subplots farther from each other.
f.subplots_adjust(hspace=0.3)
```
import matplotlib.pyplot as plt
import numpy as np

dt = 0.01
x = np.arange(-50.0, 50.0, dt)
y = np.arange(0, 100.0, dt)

plt.subplot(311)
plt.plot(x, y)
plt.xscale('symlog')
plt.ylabel('symlogyx')
plt.grid(True)
plt.gca().xaxis.grid(True, which='minor')  # minor grid on too

plt.subplot(312)
plt.plot(x, y)
plt.xscale('symlog')
plt.ylabel('symlogy')
plt.xlabel('symlog both')
plt.ylabel('symlog both')
plt.grid(True)
plt.gca().xaxis.grid(True, which='minor')  # minor grid on too

plt.show()
```python
plt.plot(y, x)
plt.yscale('symlog')
plt.ylabel('symlogy')

plt.subplot(313)
plt.plot(x, np.sin(x / 3.0))
plt.xscale('symlog')
plt.yscale('symlog', linthreshy=0.015)
plt.grid(True)
plt.ylabel('symlog both')

plt.subplots_adjust(hspace=0.5, left=0.2)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

### 99.184  `pylab_examples` example code: `system_monitor.py`

![System Monitor](image)

```
import time
import matplotlib.pyplot as plt
import numpy as np
```
def get_memory(t):
    "Simulate a function that returns system memory"
    return 100 * (0.5 + 0.5 * np.sin(0.5 * np.pi * t))

def get_cpu(t):
    "Simulate a function that returns cpu usage"
    return 100 * (0.5 + 0.5 * np.sin(0.2 * np.pi * (t - 0.25)))

def get_net(t):
    "Simulate a function that returns network bandwidth"
    return 100 * (0.5 + 0.5 * np.sin(0.7 * np.pi * (t - 0.1)))

def get_stats(t):
    return get_memory(t), get_cpu(t), get_net(t)

fig, ax = plt.subplots()
ind = np.arange(1, 4)

# show the figure, but do not block
plt.show(block=False)

pm, pc, pn = plt.bar(ind, get_stats(0))
pm.set_facecolor('r')
pc.set_facecolor('g')
pn.set_facecolor('b')
ax.set_xticks(ind)
ax.set_xticklabels(['Memory', 'CPU', 'Bandwidth'])
ax.set_ylim([0, 100])
ax.set_ylabel('Percent usage')
ax.set_title('System Monitor')

start = time.time()
for i in range(200):  # run for a little while
    m, c, n = get_stats(i / 10.0)

    # update the animated artists
    pm.set_height(m)
    pc.set_height(c)
    pn.set_height(n)

    # ask the canvas to re-draw itself the next time it
    # has a chance.
    # For most of the GUI backends this adds an event to the queue
    # of the GUI frameworks event loop.
    fig.canvas.draw_idle()
    try:
        # make sure that the GUI framework has a chance to run its event loop
        ...
    except Exception as e:
        print(e)
# and clear any GUI events. This needs to be in a try/except block
# because the default implementation of this method is to raise
# NotImplementedError
fig.canvas.flush_events()
except NotImplementedError:
    pass

stop = time.time()
print("{fps:.1f} frames per second".format(fps=200 / (stop - start)))

Keywords: python, matplotlib, pylab, example, codex (see \textit{Search examples})

99.185 \textit{pylab_examples} example code: \textit{table_demo.py}

Demo of table function to display a table within a plot.

```python
import numpy as np
import matplotlib.pyplot as plt

data = [[66386, 174296, 75131, 577908, 32015],
        [431.5, 1049.4, 799.6, 2149.8, 917.9],
        [292.2, 717.8, 456.4, 1368.5, 865.6],
        [213.8, 636.0, 305.7, 1175.2, 796.0],
        [124.6, 555.4, 153.2, 677.2, 192.5],
        [66.4, 174.3, 75.1, 577.9, 32.0]]
```
columns = ('Freeze', 'Wind', 'Flood', 'Quake', 'Hail')
rows = ['%d year' % x for x in (100, 50, 20, 10, 5)]

values = np.arange(0, 2500, 500)
value_increment = 1000

# Get some pastel shades for the colors
colors = plt.cm.BuPu(np.linspace(0, 0.5, len(rows)))
n_rows = len(data)

index = np.arange(len(columns)) + 0.3
bar_width = 0.4

# Initialize the vertical-offset for the stacked bar chart.
y_offset = np.array([0.0] * len(columns))

# Plot bars and create text labels for the table
cell_text = []
for row in range(n_rows):
    plt.bar(index, data[row], bar_width, bottom=y_offset, color=colors[row])
    y_offset = y_offset + data[row]
    cell_text.append([1.1f % (x/1000.0) for x in y_offset])

# Reverse colors and text labels to display the last value at the top.
colors = colors[::-1]
cell_text.reverse()

# Add a table at the bottom of the axes
the_table = plt.table(cellText=cell_text,
                       rowLabels=rows,
                       rowColours=colors,
                       colLabels=columns,
                       loc='bottom')

# Adjust layout to make room for the table:
plt.subplots_adjust(left=0.2, bottom=0.2)

plt.ylabel("Loss in $%0's. format(value_increment))
plt.yticks(values * value_increment, ['%d' % val for val in values])
plt.xticks([])
plt.title('Loss by Disaster')

plt.show()
You can use TeX to render all of your matplotlib text if the rc parameter text.usetex is set. This works currently on the agg and ps backends, and requires that you have tex and the other dependencies described at http://matplotlib.org/users/usetex.html properly installed on your system. The first time you run a script you will see a lot of output from tex and associated tools. The next time, the run may be silent, as a lot of the information is cached in ~/.tex.cache

import numpy as np
import matplotlib.pyplot as plt

plt.rc('text', usetex=True)
plt.rc('font', family='serif')
plt.figure(1, figsize=(6, 4))
ax = plt.axes([0.1, 0.1, 0.8, 0.7])
t = np.linspace(0.0, 1.0, 100)
s = np.cos(4 * np.pi * t) + 2
plt.plot(t, s)
plt.xlabel(r'$\textbf{time (s)}$')
plt.ylabel(r'$\textit{voltage (mV)}$', fontsize=16)
plt.title(r''\TeX\ is Number $\sum_{n=1}^{\infty}\frac{-e^{i\pi}}{2^n}$!', fontsize=16, color='r')
plt.grid(True)
plt.savefig('tex_demo')
plt.show()
```python
import matplotlib.pyplot as plt
import numpy as np

t = np.arange(0.0, 1.0 + 0.01, 0.01)
s = np.cos(2*2*np.pi*t) + 2
plt.plot(t, s)

plt.xlabel(r'$\textbf{time (s)}$')
plt.ylabel(r'\textit{Velocity (\degree/sec)}', fontsize=16)
plt.title(r'\TeX\ is Number $\sum_{n=1}^\infty \frac{-e^{i\pi}}{2^n}$!')
plt.grid(True)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

**99.188  pylab_examples example code: text_handles.py**

---

"""
Controlling the properties of axis text using handles

See examples/text_themes.py for a more elegant, pythonic way to control fonts. After all, if we were slaves to MATLAB, we wouldn't be using python!
"""

---

99.188.  pylab_examples example code: text_handles.py 2721
import matplotlib.pyplot as plt
import numpy as np

def f(t):
    s1 = np.sin(2*np.pi*t)
    e1 = np.exp(-t)
    return np.multiply(s1, e1)

t1 = np.arange(0.0, 5.0, 0.1)
t2 = np.arange(0.0, 5.0, 0.02)

fig, ax = plt.subplots()
plt.plot(t1, f(t1), 'bo', t2, f(t2), 'k')
plt.text(3.0, 0.6, 'f(t) = exp(-t) sin(2 pi t)')
ttext = plt.title('Fun with text!')
ytext = plt.ylabel('Damped oscillation')
xtext = plt.xlabel('time (s)')

plt.setp(ttext, size='large', color='r', style='italic')
plt.setp(xtext, size='medium', name=['Courier', 'DejaVu Sans Mono'],
         weight='bold', color='g')
plt.setp(ytext, size='medium', name=['Helvetica', 'DejaVu Sans'],
         weight='light', color='b')
plt.show()
The way matplotlib does text layout is counter-intuitive to some, so this example is designed to make it a little clearer. The text is aligned by its bounding box (the rectangular box that surrounds the ink rectangle). The order of operations is basically rotation then alignment, rather than alignment then rotation. Basically, the text is centered at your $x,y$ location, rotated around this point, and then aligned according to the bounding box of the rotated text.

So if you specify left, bottom alignment, the bottom left of the bounding box of the rotated text will be at the $x,y$ coordinate of the text.

But a picture is worth a thousand words!

```python
import matplotlib.pyplot as plt
import numpy as np

def addtext(props):
    plt.text(0.5, 0.5, 'text 0', props, rotation=0)
```

```python
def addtext(props):
    plt.text(0.5, 0.5, 'text 0', props, rotation=0)
```
```python
plt.text(1.5, 0.5, 'text 45', props, rotation=45)
plt.text(2.5, 0.5, 'text 135', props, rotation=135)
plt.text(3.5, 0.5, 'text 225', props, rotation=225)
plt.text(4.5, 0.5, 'text -45', props, rotation=-45)
plt.yticks([0, .5, 1])
plt.grid(True)

# the text bounding box
bbox = {'fc': '0.8', 'pad': 0}

plt.subplot(211)
addtext({'ha': 'center', 'va': 'center', 'bbox': bbox})
plt.xlim(0, 5)
plt.xticks(np.arange(0, 5.1, 0.5), []
plt.ylabel('center / center')

plt.subplot(212)
addtext({'ha': 'left', 'va': 'bottom', 'bbox': bbox})
plt.xlim(0, 5)
plt.xticks(np.arange(0, 5.1, 0.5))
plt.ylabel('left / bottom')
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Text objects in matplotlib are normally rotated with respect to the screen coordinate system (i.e., 45 degrees rotation plots text along a line that is in between horizontal and vertical no matter how the axes are changed). However, at times one wants to rotate text with respect to something on the plot. In this case, the correct angle won’t be the angle of that object in the plot coordinate system, but the angle that that object APPEARS in the screen coordinate system. This angle is found by transforming the angle from the plot to the screen coordinate system, as shown in the example below.

```python
import matplotlib.pyplot as plt
import numpy as np

# Plot diagonal line (45 degrees)
h = plt.plot(np.arange(0, 10), np.arange(0, 10))

# set limits so that it no longer looks on screen to be 45 degrees
plt.xlim([-10, 20])

# Locations to plot text
```
```python
l1 = np.array((1, 1))
l2 = np.array((5, 5))

# Rotate angle
angle = 45
trans_angle = plt.gca().transData.transform_angles(np.array((45,)),
                                                  l2.reshape((1, 2)))[0]

# Plot text
th1 = plt.text(l1[0], l1[1],
                'text not rotated correctly',
               fontsize=16,
               rotation=angle, rotation_mode='anchor')
th2 = plt.text(l2[0], l2[1],
                'text rotated correctly',
               fontsize=16,
               rotation=trans_angle, rotation_mode='anchor')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

**99.191 pylab_examples example code: titles_demo.py**

Matplotlib can display plot titles centered, flush with the left side of
import matplotlib.pyplot as plt
plt.plot(range(10))
plt.title('Center Title')
plt.title('Left Title', loc='left')
plt.title('Right Title', loc='right')
plt.show()
If you want two images with different shapes to be plotted with the same extent, they must have the same "extent" property

As usual, we'll define some random images for demo. Real data is much more exciting!

Note, on the wx backend on some platforms (e.g., linux), you have to first click on the figure before the keypress events are activated. If you know how to fix this, please email us!

```python
import matplotlib.pyplot as plt
import numpy as np

# two images x1 is initially visible, x2 is not
x1 = np.random.random((100, 100))
x2 = np.random.random((150, 175))

# arbitrary extent - both images must have same extent if you want them to be resampled into the same axes space
extent = (0, 1, 0, 1)
im1 = plt.imshow(x1, extent=extent)
im2 = plt.imshow(x2, extent=extent)
im2.set_visible(False)

def toggle_images(event):
    'toggle the visible state of the two images'
    if event.key != 't':
        return
    b1 = im1.get_visible()
b2 = im2.get_visible()
im1.set_visible(not b1)
im2.set_visible(not b2)
plt.draw()

plt.connect('key_press_event', toggle_images)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
99.193  

pylab_examples example code: transoffset.py

---

```python
import numpy as np
import matplotlib.pyplot as plt

theta = np.arange(0, np.pi, np.pi/8)
r = np.arange(0, 10, 1)

fig, ax = plt.subplots(projection='polar')
ax.scatter(theta=r, r=theta, marker='o')
ax.set_thetagrids(theta, r)

plt.show()
```
This illustrates the use of transforms.offset_copy to make a transform that positions a drawing element such as a text string at a specified offset in screen coordinates (dots or inches) relative to a location given in any coordinates.

Every Artist--the mpl class from which classes such as Text and Line are derived--has a transform that can be set when the Artist is created, such as by the corresponding pyplot command. By default this is usually the Axes.transData transform, going from data units to screen dots. We can use the offset_copy function to make a modified copy of this transform, where the modification consists of an offset.

```python
import matplotlib.pyplot as plt
import matplotlib.transforms as mtrans
import numpy as np

from matplotlib.transforms import offset_copy

xs = np.arange(7)
ys = xs**2

fig = plt.figure(figsize=(5, 10))
ax = plt.subplot(2, 1, 1)

# If we want the same offset for each text instance, 
# we only need to make one transform. To get the 
# transform argument to offset_copy, we need to make the axes 
# first; the subplot command above is one way to do this.
trans_offset = mtrans.offset_copy(ax.transData, fig=fig, 
                                  x=0.05, y=0.10, units='inches')

for x, y in zip(xs, ys):
    plt.plot((x,), (y,), 'ro')
    plt.text(x, y, '%d,%d' % (int(x), int(y)), transform=trans_offset)

# offset_copy works for polar plots also.
ax = plt.subplot(2, 1, 2, projection='polar')
trans_offset = mtrans.offset_copy(ax.transData, fig=fig, y=6, units='dots')

for x, y in zip(xs, ys):
    plt.polar((x,), (y,), 'ro')
    plt.text(x, y, '%d,%d' % (int(x), int(y)), transform=trans_offset, 
             horizontalalignment='center', 
             verticalalignment='bottom')
```
Contour plot of Delaunay triangulation

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

99.194 pylab_examples example code: tricontour_demo.py
## Contour plots of unstructured triangular grids.

```python
import matplotlib.pyplot as plt
import matplotlib.tri as tri
import numpy as np
import math

# Creating a Triangulation without specifying the triangles results in the
# Delaunay triangulation of the points.

# First create the x and y coordinates of the points.

n_angles = 48
n_radii = 8
min_radius = 0.25
radii = np.linspace(min_radius, 0.95, n_radii)
angles = np.linspace(0, 2*math.pi, n_angles, endpoint=False)
angles = np.repeat(angles[..., np.newaxis], n_radii, axis=1)
angles[:, 1::2] += math.pi/n_angles

x = (radii*np.cos(angles)).flatten()
y = (radii*np.sin(angles)).flatten()
z = (np.cos(radii)*np.cos(angles*3.0)).flatten()

# Create the Triangulation; no triangles so Delaunay triangulation created.
```

---

99.194. pylab_examples example code: tricontour_demo.py 2733
```python
import matplotlib.pyplot as plt
import numpy as np
import matplotlib.tri as tri

# Create a Delaunay triangulation
triang = tri.Triangulation(x, y)

# Mask off unwanted triangles.
xis = x[triang.triangles].mean(axis=1)
yis = y[triang.triangles].mean(axis=1)
mask = np.where(xis*xis + yis*yis < min_radius*min_radius, 1, 0)
triang.set_mask(mask)

# pcolor plot.
plt.figure()
plt.gca().set_aspect('equal')
plt.tricontourf(triang, z)
plt.colorbar()
plt.tricontour(triang, z, colors='k')
plt.title('Contour plot of Delaunay triangulation')

# You can specify your own triangulation rather than perform a Delaunay triangulation of the points, where each triangle is given by the indices of the three points that make up the triangle, ordered in either a clockwise or anticlockwise manner.

xy = np.array([[[-0.101, 0.872], [-0.080, 0.883], [-0.069, 0.888], [-0.054, 0.890],
[-0.045, 0.897], [-0.057, 0.895], [-0.073, 0.900], [-0.087, 0.898],
[-0.090, 0.904], [-0.069, 0.907], [-0.069, 0.921], [-0.089, 0.919],
[-0.073, 0.928], [-0.052, 0.930], [-0.048, 0.942], [-0.062, 0.949],
[-0.054, 0.958], [-0.069, 0.954], [-0.087, 0.952], [-0.087, 0.959],
[-0.080, 0.966], [-0.085, 0.973], [-0.087, 0.965], [-0.097, 0.965],
[-0.097, 0.975], [-0.092, 0.984], [-0.101, 0.980], [-0.108, 0.980],
[-0.104, 0.987], [-0.102, 0.993], [-0.115, 1.001], [-0.099, 0.996],
[-0.101, 1.007], [-0.090, 1.010], [-0.087, 1.021], [-0.069, 1.021],
[-0.052, 1.022], [-0.053, 1.017], [-0.069, 1.010], [-0.064, 1.005],
[-0.048, 1.005], [-0.031, 1.005], [-0.031, 0.996], [-0.040, 0.987],
[-0.045, 0.980], [-0.052, 0.975], [-0.040, 0.973], [-0.026, 0.968],
[-0.020, 0.954], [-0.006, 0.947], [ 0.003, 0.935], [ 0.006, 0.926],
[ 0.005, 0.921], [ 0.022, 0.923], [ 0.033, 0.912], [ 0.029, 0.905],
[ 0.017, 0.900], [ 0.012, 0.895], [ 0.027, 0.893], [ 0.019, 0.886],
[ 0.001, 0.883], [-0.012, 0.884], [-0.029, 0.883], [-0.038, 0.879],
[-0.057, 0.881], [-0.062, 0.876], [-0.078, 0.876], [-0.087, 0.872],
[-0.030, 0.907], [-0.007, 0.905], [-0.057, 0.916], [-0.025, 0.933],
[-0.077, 0.990], [-0.059, 0.993]])
x = np.degrees(xy[:, 0])
y = np.degrees(xy[:, 1])
x0 = -5
y0 = 52
z = np.exp(-0.01*(x - x0)*(x - x0) + (y - y0)*(y - y0))

triangles = np.array([[67, 66, 1], [65, 2, 66], [1, 66, 2], [64, 2, 65], [63, 3, 64],
[60, 59, 57], [2, 64, 3], [3, 63, 4], [0, 67, 1], [62, 4, 63],
[57, 59, 56], [59, 58, 56], [61, 60, 69], [57, 69, 60], [4, 62, 68],
[6, 5, 9], [61, 68, 62], [69, 68, 61], [9, 5, 70], [6, 8, 7],
[4, 70, 5], [8, 6, 9], [56, 69, 57], [69, 56, 52], [70, 10, 9]], dtype=int32)
```
# Rather than create a Triangulation object, can simply pass x, y and triangles
# arrays to tripcolor directly. It would be better to use a Triangulation
# object if the same triangulation was to be used more than once to save
# duplicated calculations.
plt.figure()
plt.gca().set_aspect('equal')
plt.tricontourf(x, y, triangles, z)
plt.colorbar()
plt.title('Contour plot of user-specified triangulation')
plt.xlabel('Longitude (degrees)')
plt.ylabel('Latitude (degrees)')
plt.show()
Filtering a Delaunay mesh
(application to high-resolution tricontouring)

Demonstrates high-resolution tricontouring of a random set of points; a matplotlib.tri.TriAnalyzer is used to improve the plot quality.

The initial data points and triangular grid for this demo are:
- a set of random points is instantiated, inside \([-1, 1]\) x \([-1, 1]\) square
- A Delaunay triangulation of these points is then computed, of which a random subset of triangles is masked out by the user (based on *init_mask_frac* parameter). This simulates invalidated data.

The proposed generic procedure to obtain a high resolution contouring of such a data set is the following:
1) Compute an extended mask with a matplotlib.tri.TriAnalyzer, which will exclude badly shaped (flat) triangles from the border of the triangulation. Apply the mask to the triangulation (using set_mask).
2) Refine and interpolate the data using a matplotlib.tri.UniformTriRefiner.
3) Plot the refined data with tricontour.

```python
from matplotlib.tri import Triangulation, TriAnalyzer, UniformTriRefiner
import matplotlib.pyplot as plt
```
# Analytical test function

def experiment_res(x, y):
    """An analytic function representing experiment results""
    x = 2.*x
    r1 = np.sqrt((0.5 - x)**2 + (0.5 - y)**2)
    thetal1 = np.arctan2(0.5 - x, 0.5 - y)
    r2 = np.sqrt((-x - 0.2)**2 + (-y - 0.2)**2)
    thetal2 = np.arctan2(-x - 0.2, -y - 0.2)
    z = (4*(np.exp((r1/10)**2) - 1)*30.*np.cos(3*thetal1) +
         (np.exp((r2/10)**2) - 1)*30.*np.cos(5*thetal2) +
         2*(x**2 + y**2))
    return (np.max(z) - z)/(np.max(z) - np.min(z))

# Generating the initial data test points and triangulation for the demo

# User parameters for data test points
n_test = 200  # Number of test data points, tested from 3 to 5000 for subdiv=3
subdiv = 3  # Number of recursive subdivisions of the initial mesh for smooth plots. Values >3 might result in a very high number of triangles # for the refine mesh: new triangles numbering = (4**subdiv)*ntri
init_mask_frac = 0.0  # Float > 0. adjusting the proportion of (invalid) initial triangles which will be masked # out. Enter 0 for no mask.
min_circle_ratio = .01  # Minimum circle ratio - border triangles with circle ratio below this will be masked if they touch a # border. Suggested value 0.01 ; Use -1 to keep all triangles.

# Random points
random_gen = np.random.mtrand.RandomState(seed=127260)
x_test = random_gen.uniform(-1., 1., size=n_test)
y_test = random_gen.uniform(-1., 1., size=n_test)
z_test = experiment_res(x_test, y_test)

# meshing with Delaunay triangulation
tri = Triangulation(x_test, y_test)
ntri = tri.triangles.shape[0]

# Some invalid data are masked out
mask_init = np.zeros(ntri, dtype=np.bool)
masked_tri = random_gen.randint(0, ntri, int(ntri*init_mask_frac))
mask_init[masked_tri] = True
tri.set_mask(mask_init)
# Improving the triangulation before high-res plots: removing flat triangles
# masking badly shaped triangles at the border of the triangular mesh.
mask = TriAnalyzer(tri).get_flat_tri_mask(min_circle_ratio)
tri.set_mask(mask)

# refining the data
refiner = UniformTriRefiner(tri)
tri_refi, z_test_refi = refiner.refine_field(z_test, subdiv=subdiv)

# analytical 'results' for comparison
z_expected = experiment_res(tri_refi.x, tri_refi.y)

# for the demo: loading the 'flat' triangles for plot
flat_tri = Triangulation(x_test, y_test)
flat_tri.set_mask(~mask)

# Now the plots
# User options for plots
# Graphical options for tricontouring
levels = np.arange(0., 1., 0.025)
cmap = cm.get_cmap(name='Blues', lut=None)

plt.figure()
plt.gca().set_aspect('equal')
plt.title("Filtering a Delaunay mesh\n" +
"(application to high-resolution tricontouring)"")

# 1) plot of the refined (computed) data contours:
plt.tricontour(tri_refi, z_test_refi, levels=levels, cmap=cmap,
               linewidths=[2.0, 0.5, 1.0, 0.5])
# 2) plot of the expected (analytical) data contours (dashed):
if plot_expected:
    plt.tricontour(tri_refi, z_expected, levels=levels, cmap=cmap,
                   linestyles='--')
# 3) plot of the fine mesh on which interpolation was done:
if plot_refi_tri:
    plt.triplot(tri_refi, color='0.97')
# 4) plot of the initial 'coarse' mesh:
if plot_tri:
    plt.triplot(tri, color='0.7')
# 4) plot of the unvalidated triangles from naive Delaunay Triangulation:
if plot_masked_tri:
    plt.triplot(flat_tri, color='red')
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

99.196  pylab_examples example code: tricontour_smooth_user.py

```python
"""
Demonstrates high-resolution tricontouring on user-defined triangular grids
with matplotlib.tri.UniformTriRefiner
"""
import matplotlib.tri as tri
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import numpy as np
import math

#----------------------------------
```
# Analytical test function

```python
# A function of 2 variables

def function_z(x, y):
    r1 = np.sqrt((0.5 - x)**2 + (0.5 - y)**2)
    theta1 = np.arctan2(0.5 - x, 0.5 - y)
    r2 = np.sqrt((-x - 0.2)**2 + (-y - 0.2)**2)
    theta2 = np.arctan2(-x - 0.2, -y - 0.2)
    z = -2*(np.exp((r1/10)**2) - 1)**30. * np.cos(7.*theta1) +
       (np.exp((r2/10)**2) - 1)**30. * np.cos(11.*theta2) +
       0.7*(x**2 + y**2)
    return (np.max(z) - z)/(np.max(z) - np.min(z))
```

# Creating a Triangulation

```python
# First create the x and y coordinates of the points.

n_angles = 20
n_radii = 10
min_radius = 0.15
radii = np.linspace(min_radius, 0.95, n_radii)

angles = np.linspace(0, 2*math.pi, n_angles, endpoint=False)
angles = np.repeat(angles[..., np.newaxis], n_radii, axis=1)
angles[:, 1::2] += math.pi/n_angles

x = (radii*np.cos(angles)).flatten()

y = (radii*np.sin(angles)).flatten()

z = function_z(x, y)
```

# Now create the Triangulation.  
# (Creating a Triangulation without specifying the triangles results in the  
# Delaunay triangulation of the points.)

```python
triang = tri.Triangulation(x, y)
```

# Mask off unwanted triangles.

```python
xmid = x[triang.triangles].mean(axis=1)

ymid = y[triang.triangles].mean(axis=1)

mask = np.where(xmid*xmid + ymid*ymid < min_radius*min_radius, 1, 0)

triang.set_mask(mask)
```

# Refine data

```python
refiner = tri.UniformTriRefiner(triang)

tri_refi, z_test_refi = refiner.refine_field(z, subdiv=3)
```

# Plot the triangulation and the high-res iso-contours

```python
plt.figure()

plt.triplot(triang, lw=0.5, color='white')
```
levels = np.arange(0., 1., 0.025)
cmap = cm.get_cmap(name='terrain', lut=None)
plt.tricontourf(tri_refi, z_test_refi, levels=levels, cmap=cmap)
plt.tricontour(tri_refi, z_test_refi, levels=levels,
               colors=['0.25', '0.5', '0.5', '0.5', '0.5'],
               linewidths=[1.0, 0.5, 0.5, 0.5, 0.5])
plt.title("High-resolution tricontouring")
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

99.197  pylab_examples example code: tricontour_vs_griddata.py

Comparison of griddata and tricontour for an unstructured triangular grid.

from __future__ import print_function
import matplotlib.pyplot as plt
import matplotlib.tri as tri

griddata and contour (200 points, 20000 grid points)

tricontour (200 points)
import numpy as np
import numpy.random as rnd
import matplotlib.mlab as mlab
import time

rnd.seed(0)
npts = 200
ngridx = 100
ngridy = 200
x = rnd.uniform(-2, 2, npts)
y = rnd.uniform(-2, 2, npts)
z = x*np.exp(-x**2 - y**2)

# griddata and contour.
start = time.clock()
plt.subplot(211)
xi = np.linspace(-2.1, 2.1, ngridx)
iy = np.linspace(-2.1, 2.1, ngridy)
zi = mlab.griddata(x, y, z, xi, iy, interp='linear')
plt.contour(xi, iy, zi, 15, linewidths=0.5, colors='k')
plt.contourf(xi, iy, zi, 15,
            norm=plt.Normalize(vmax=abs(zi).max(), vmin=-abs(zi).max()))
plt.colorbar()
plt.plot(x, y,
        'ko', ms=3)
plt.xlim(-2, 2)
plt.ylim(-2, 2)
plt.title('griddata and contour (%d points, %d grid points)' %
          (npts, ngridx*ngridy))
print('griddata and contour seconds: %f' % (time.clock() - start))

# tricontour.
start = time.clock()
plt.subplot(212)
triang = tri.Triangulation(x, y)
plt.tricontour(x, y, z, 15, linewidths=0.5, colors='k')
plt.tricontourf(x, y, z, 15,
                norm=plt.Normalize(vmax=abs(zi).max(), vmin=-abs(zi).max()))
plt.colorbar()
plt.plot(x, y,
        'ko', ms=3)
plt.xlim(-2, 2)
plt.ylim(-2, 2)
plt.title('tricontour (%d points)' % npts)
print('tricontour seconds: %f' % (time.clock() - start))

plt.subplots_adjust(hspace=0.5)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Gradient plot: an electrical dipole

Demonstrates computation of gradient with matplotlib.tri.CubicTriInterpolator.

```python
from matplotlib.tri import Triangulation, UniformTriRefiner, CubicTriInterpolator
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import numpy as np
import math

# Electrical potential of a dipole

def dipole_potential(x, y):
    """The electric dipole potential V""
    r_sq = x**2 + y**2
    theta = np.arctan2(y, x)
    z = np.cos(theta)/r_sq
    return (np.max(z) - z) / (np.max(z) - np.min(z))
```

```python
from matplotlib.tri import Triangulation, UniformTriRefiner, CubicTriInterpolator
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import numpy as np
import math

# Electrical potential of a dipole

def dipole_potential(x, y):
    """The electric dipole potential V""
    r_sq = x**2 + y**2
    theta = np.arctan2(y, x)
    z = np.cos(theta)/r_sq
    return (np.max(z) - z) / (np.max(z) - np.min(z))
```
```
# Creating a Triangulation
# First create the x and y coordinates of the points.
n_angles = 30
n_radii = 10
min_radius = 0.2
radii = np.linspace(min_radius, 0.95, n_radii)
angles = np.linspace(0, 2*math.pi, n_angles, endpoint=False)
angles[:, 1::2] += math.pi/n_angles

x = (radii*np.cos(angles)).flatten()
y = (radii*np.sin(angles)).flatten()
V = dipole_potential(x, y)

# Create the Triangulation; no triangles specified so Delaunay triangulation
# created.
triang = Triangulation(x, y)

# Mask off unwanted triangles.
xmid = x[triang.triangles].mean(axis=1)
ymid = y[triang.triangles].mean(axis=1)
mask = np.where(xmid*xmid + ymid*ymid < min_radius*min_radius, 1, 0)
triang.set_mask(mask)

# Refine data - interpolates the electrical potential V
refiner = UniformTriRefiner(triang)
tri_refi, z_test_refi = refiner.refine_field(V, subdiv=3)

# Computes the electrical field (Ex, Ey) as gradient of electrical potential

tci = CubicTriInterpolator(triang, -V)
# Gradient requested here at the mesh nodes but could be anywhere else:
(Ex, Ey) = tci.gradient(triang.x, triang.y)
E_norm = np.sqrt(Ex**2 + Ey**2)

# Plot the triangulation, the potential iso-contours and the vector field
fig, ax = plt.subplots()
ax.set_aspect('equal')
# Enforce the margins, and enlarge them to give room for the vectors.
ax.use_sticky_edges = False
ax.margins(0.07)

ax.triplot(triang, color='0.8')

levels = np.arange(0., 1., 0.01)
```
```python
cmap = cm.get_cmap(name='hot', lut=None)
ax.tricontour(tri_refi, z_test_refi, levels=levels, cmap=cmap,
             linewidths=[2.0, 1.0, 1.0, 1.0])
# Plots direction of the electrical vector field
ax.quiver(triang.x, triang.y, Ex/E_norm, Ey/E_norm,
          units='xy', scale=10., zorder=3, color='blue',
          width=0.007, headwidth=3., headlength=4.)
ax.set_title('Gradient plot: an electrical dipole')
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

99.199  *pylab_examples* example code: *triinterp_demo.py*
# Create triangulation.
x = np.asarray([0, 1, 2, 3, 0.5, 1.5, 2.5, 1, 2, 1.5])
y = np.asarray([0, 0, 0, 1.0, 1.0, 1.0, 2, 2, 3.0])
triangles = [[0, 1, 4], [1, 2, 5], [2, 3, 6], [1, 5, 4], [2, 6, 5], [4, 5, 7],
             [5, 6, 8], [5, 8, 7], [7, 8, 9]]
triang = mtri.Triangulation(x, y, triangles)

# Interpolate to regularly-spaced quad grid.
z = np.cos(1.5*x)*np.cos(1.5*y)
xi, yi = np.meshgrid(np.linspace(0, 3, 20), np.linspace(0, 3, 20))

interp_lin = mtri.LinearTriInterpolator(triang, z)
zi_lin = interp_lin(xi, yi)

interp_cubic_geom = mtri.CubicTriInterpolator(triang, z, kind='geom')
zi_cubic_geom = interp_cubic_geom(xi, yi)

interp_cubic_min_E = mtri.CubicTriInterpolator(triang, z, kind='min_E')
zi_cubic_min_E = interp_cubic_min_E(xi, yi)

# Plot the triangulation.
plt.subplot(221)
plt.tricontourf(triang, z)
plt.triplot(triang, 'ko-')
plt.title('Triangular grid')

# Plot linear interpolation to quad grid.
plt.subplot(222)
plt.contourf(xi, yi, zi_lin)
plt.plot(xi, yi, 'k-', lw=0.5, alpha=0.5)
plt.plot(xi.T, yi.T, 'k-', lw=0.5, alpha=0.5)
plt.title('Linear interpolation')

# Plot cubic interpolation to quad grid, kind=geom
plt.subplot(223)
plt.contourf(xi, yi, zi_cubic_geom)
plt.plot(xi, yi, 'k-', lw=0.5, alpha=0.5)
plt.plot(xi.T, yi.T, 'k-', lw=0.5, alpha=0.5)
plt.title('Cubic interpolation, \texttt{kind=geom}')

# Plot cubic interpolation to quad grid, kind=min_E
plt.subplot(224)
plt.contourf(xi, yi, zi_cubic_min_E)
plt.plot(xi, yi, 'k-', lw=0.5, alpha=0.5)
plt.plot(xi.T, yi.T, 'k-', lw=0.5, alpha=0.5)
plt.title('Cubic interpolation, \texttt{kind=min_E}"

plt.tight_layout()
plt.show()
99.200  pylab_examples example code: tripcolor_demo.py

tripcolor of Delaunay triangulation, flat shading
tripcolor of Delaunay triangulation, gouraud shading
Pseudocolor plots of unstructured triangular grids.

```python
import matplotlib.pyplot as plt
import matplotlib.tri as tri
import numpy as np
import math

# Creating a Triangulation without specifying the triangles results in the
# Delaunay triangulation of the points.

# First create the x and y coordinates of the points.
n_angles = 36
n_radii = 8
min_radius = 0.25
radii = np.linspace(min_radius, 0.95, n_radii)
angles = np.linspace(0, 2*math.pi, n_angles, endpoint=False)
angles = np.repeat(angles[..., np.newaxis], n_radii, axis=1)
angles[:, 1::2] += math.pi/n_angles
x = (radii*np.cos(angles)).flatten()
y = (radii*np.sin(angles)).flatten()
z = (np.cos(radii)*np.cos(angles*3.0)).flatten()

# Create the Triangulation; no triangles so Delaunay triangulation created.
```
```python
matrix = tri.Triangulation(x, y)

# Mask off unwanted triangles.
xmid = x[triang.triangles].mean(axis=1)
ymid = y[triang.triangles].mean(axis=1)
mask = np.where(xmid*xmid + ymid*ymid < min_radius*min_radius, 1, 0)
triang.set_mask(mask)

# tripcolor plot.
plt.figure()
plt.gca().set_aspect('equal')
plt.tripcolor(triang, z, shading='flat')
plt.colorbar()
plt.title('tripcolor of Delaunay triangulation, flat shading')

# Illustrate Gouraud shading.
plt.figure()
plt.gca().set_aspect('equal')
plt.tripcolor(triang, z, shading='gouraud')
plt.colorbar()
plt.title('tripcolor of Delaunay triangulation, gouraud shading')

# You can specify your own triangulation rather than perform a Delaunay triangulation of the points, where each triangle is given by the indices of the three points that make up the triangle, ordered in either a clockwise or anticlockwise manner.

xy = np.asarray([
    [-0.101, 0.872], [-0.080, 0.883], [-0.069, 0.888], [-0.054, 0.890],
    [-0.045, 0.897], [-0.057, 0.895], [-0.073, 0.900], [-0.087, 0.898],
    [-0.090, 0.904], [-0.069, 0.907], [-0.069, 0.921], [-0.080, 0.919],
    [-0.073, 0.928], [-0.052, 0.930], [-0.048, 0.942], [-0.062, 0.949],
    [-0.054, 0.958], [-0.069, 0.954], [-0.087, 0.952], [-0.087, 0.959],
    [-0.058, 0.966], [-0.085, 0.973], [-0.087, 0.965], [-0.097, 0.965],
    [-0.097, 0.975], [-0.092, 0.984], [-0.101, 0.980], [-0.108, 0.980],
    [-0.104, 0.987], [-0.102, 0.993], [-0.115, 1.001], [-0.099, 0.996],
    [-0.101, 1.007], [-0.090, 1.010], [-0.087, 1.021], [-0.069, 1.021],
    [-0.052, 1.022], [-0.052, 1.017], [-0.069, 1.010], [-0.064, 1.005],
    [-0.048, 1.005], [-0.031, 1.005], [-0.031, 0.996], [-0.040, 0.987],
    [-0.045, 0.980], [-0.052, 0.975], [-0.040, 0.973], [-0.026, 0.968],
    [-0.020, 0.954], [-0.006, 0.947], [ 0.003, 0.935], [ 0.006, 0.926],
    [ 0.005, 0.921], [ 0.022, 0.923], [ 0.033, 0.912], [ 0.029, 0.905],
    [ 0.017, 0.900], [ 0.012, 0.895], [ 0.027, 0.893], [ 0.019, 0.886],
    [ 0.001, 0.883], [-0.012, 0.884], [-0.029, 0.883], [-0.038, 0.879],
    [-0.057, 0.881], [-0.062, 0.876], [-0.078, 0.876], [-0.087, 0.872],
    [-0.030, 0.907], [-0.007, 0.905], [-0.057, 0.916], [-0.025, 0.933],
    [-0.077, 0.990], [-0.059, 0.993]])

x = xy[:, 0]*180/3.14159
y = xy[:, 1]*180/3.14159

triangles = np.asarray([[
    [67, 66, 1], [65, 2, 66], [1, 66, 2], [64, 2, 65], [63, 3, 64],
    [62, 3, 64], [61, 3, 64], [60, 3, 64], [59, 3, 64], [58, 3, 64],
    [57, 3, 64], [56, 3, 64], [55, 3, 64], [54, 3, 64], [53, 3, 64],
    [52, 3, 64], [51, 3, 64], [50, 3, 64], [49, 3, 64], [48, 3, 64],
    [47, 3, 64], [46, 3, 64], [45, 3, 64], [44, 3, 64], [43, 3, 64],
    [42, 3, 64], [41, 3, 64], [40, 3, 64], [39, 3, 64], [38, 3, 64],
    [37, 3, 64], [36, 3, 64], [35, 3, 64], [34, 3, 64], [33, 3, 64],
    [32, 3, 64], [31, 3, 64], [30, 3, 64], [29, 3, 64], [28, 3, 64],
    [27, 3, 64], [26, 3, 64], [25, 3, 64], [24, 3, 64], [23, 3, 64],
    [22, 3, 64], [21, 3, 64], [20, 3, 64], [19, 3, 64], [18, 3, 64],
    [17, 3, 64], [16, 3, 64], [15, 3, 64], [14, 3, 64], [13, 3, 64],
    [12, 3, 64], [11, 3, 64], [10, 3, 64], [9, 3, 64], [8, 3, 64],
    [7, 3, 64], [6, 3, 64], [5, 3, 64], [4, 3, 64], [3, 3, 64],
    [2, 3, 64], [1, 3, 64], [0, 3, 64]])
```
xmid = x[triangles].mean(axis=1)
ymid = y[triangles].mean(axis=1)
x0 = -5
y0 = 52
zfaces = np.exp(-0.01*((xmid - x0)*(xmid - x0) + (ymid - y0)*(ymid - y0)))

# Rather than create a Triangulation object, can simply pass x, y and triangles
# arrays to tripcolor directly. It would be better to use a Triangulation
# object if the same triangulation was to be used more than once to save
# duplicated calculations.
# Can specify one color value per face rather than one per point by using the
# facecolors kwarg.
plt.figure()
plt.gca().set_aspect('equal')
plt.tripcolor(x, y, triangles, facecolors=zfaces, edgecolors='k')
plt.colorbar()
plt.title('tripcolor of user-specified triangulation')
plt.xlabel('Longitude (degrees)')
plt.ylabel('Latitude (degrees)')
plt.show()
99.201  

pylab_examples example code: triplot_demo.py

triplot of Delaunay triangulation
Creating and plotting unstructured triangular grids.

```python
import matplotlib.pyplot as plt
import matplotlib.tri as tri
import numpy as np
import math

# Creating a Triangulation without specifying the triangles results in the
# Delaunay triangulation of the points.

# First create the x and y coordinates of the points.
```

```python
n_angles = 36
n_radii = 8
min_radius = 0.25
radii = np.linspace(min_radius, 0.95, n_radii)
```

```python
angles = np.linspace(0, 2*math.pi, n_angles, endpoint=False)
angles = np.repeat(angles[..., np.newaxis], n_radii, axis=1)
angles[:, 1::2] += math.pi/n_angles
```

```python
x = (radii*np.cos(angles)).flatten()
y = (radii*np.sin(angles)).flatten()
```

```python
# Create the Triangulation; no triangles so Delaunay triangulation created.
triang = tri.Triangulation(x, y)
```
# Mask off unwanted triangles.
xmid = x[triang.triangles].mean(axis=1)
ymid = y[triang.triangles].mean(axis=1)
mask = np.where(xmid*xmid + ymid*ymid < min_radius*min_radius, 1, 0)
triang.set_mask(mask)

# Plot the triangulation.
plt.figure()
plt.gca().set_aspect('equal')
plt.triplot(triang, 'bo-', lw=1)
plt.title('triplot of Delaunay triangulation')

# You can specify your own triangulation rather than perform a Delaunay
# triangulation of the points, where each triangle is given by the indices of
# the three points that make up the triangle, ordered in either a clockwise or
# anticlockwise manner.

xy = np.asarray([[-0.101, 0.872], [-0.080, 0.883], [-0.069, 0.888], [-0.054, 0.890],
                 [-0.045, 0.897], [-0.057, 0.895], [-0.073, 0.900], [-0.087, 0.898],
                 [-0.090, 0.904], [-0.069, 0.907], [-0.069, 0.921], [-0.080, 0.919],
                 [-0.073, 0.928], [-0.052, 0.930], [-0.048, 0.942], [-0.062, 0.949],
                 [-0.054, 0.958], [-0.069, 0.954], [-0.087, 0.952], [-0.087, 0.959],
                 [-0.080, 0.966], [-0.085, 0.973], [-0.087, 0.965], [-0.097, 0.975],
                 [-0.097, 0.975], [-0.092, 0.984], [-0.101, 0.980], [-0.108, 0.980],
                 [-0.104, 0.987], [-0.102, 0.993], [-0.115, 1.001], [-0.099, 0.996],
                 [-0.101, 1.007], [-0.090, 1.010], [-0.087, 1.021], [-0.069, 1.021],
                 [-0.052, 1.022], [-0.052, 1.017], [-0.069, 1.010], [-0.064, 1.005],
                 [-0.048, 1.005], [-0.031, 1.005], [-0.031, 0.996], [-0.040, 0.987],
                 [-0.045, 0.980], [-0.052, 0.975], [-0.049, 0.973], [-0.026, 0.968],
                 [-0.020, 0.954], [-0.006, 0.947], [ 0.003, 0.935], [ 0.006, 0.926],
                 [ 0.005, 0.921], [ 0.022, 0.923], [ 0.033, 0.912], [ 0.029, 0.905],
                 [ 0.017, 0.906], [ 0.012, 0.895], [ 0.027, 0.893], [ 0.019, 0.886],
                 [ 0.001, 0.883], [-0.012, 0.884], [-0.029, 0.883], [-0.038, 0.879],
                 [-0.057, 0.881], [-0.062, 0.876], [-0.078, 0.876], [-0.087, 0.872],
                 [-0.030, 0.907], [-0.007, 0.905], [-0.057, 0.916], [-0.025, 0.933],
                 [-0.077, 0.990], [-0.059, 0.993]])
x = np.degrees(xy[:, 0])
y = np.degrees(xy[:, 1])

triangles = np.asarray([67, 66, 1], [65, 2, 66], [1, 66, 2], [64, 65, 63, 3, 64],
                        [60, 59, 57], [2, 64, 3], [3, 63, 4], [0, 67, 1], [62, 63, 6],
                        [57, 59, 56], [59, 58, 56], [61, 60, 69], [57, 69, 60], [4, 62, 68],
                        [6, 5, 9], [61, 68, 62], [69, 68, 61], [9, 5, 70], [6, 8, 7],
                        [4, 70, 5], [8, 6, 9], [56, 69, 57], [69, 56, 52], [70, 10, 9],
                        [54, 53, 51], [56, 55, 53], [68, 70, 4], [52, 56, 53], [11, 10, 12],
                        [69, 71, 68], [68, 13, 70], [10, 70, 13], [51, 50, 52], [13, 68, 71],
                        [52, 71, 69], [12, 10, 13], [71, 52, 56], [71, 14, 13], [50, 49, 71],
                        [49, 48, 71], [14, 16, 15], [14, 71, 48], [17, 19, 18], [17, 20, 19],
                        [48, 16, 14], [48, 47, 16], [47, 46, 16], [16, 46, 45], [23, 22, 24],]
# Rather than create a Triangulation object, can simply pass x, y and triangles
# arrays to triplot directly. It would be better to use a Triangulation object
# if the same triangulation was to be used more than once to save duplicated
# calculations.
plt.figure()
plt.gca().set_aspect('equal')
plt.triplot(x, y, triangles, 'go-', lw=1.0)
plt.title('triplot of user-specified triangulation')
plt.xlabel('Longitude (degrees)')
plt.ylabel('Latitude (degrees)')
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
import matplotlib
import matplotlib.pyplot as plt
import matplotlib.axes as maxes

from matplotlib import rcParams
rcParams['text.usetex'] = True
rcParams['text.latex.unicode'] = True

class Axes(maxes.Axes):
A hackish way to simultaneously draw texts w/ usetex=True and usetex=False in the same figure. It does not work in the ps backend.

```python
def __init__(self, *kl, **kw):
    self.usetex = kw.pop("usetex", "False")
    self.preview = kw.pop("preview", "False")

    maxes.Axes.__init__(self, *kl, **kw)

def draw(self, renderer):
    usetex = plt.rcParams["text.usetex"]
    preview = plt.rcParams["text.latex.preview"]
    plt.rcParams["text.usetex"] = self.usetex
    plt.rcParams["text.latex.preview"] = self.preview

    maxes.Axes.draw(self, renderer)

    plt.rcParams["text.usetex"] = usetex
    plt.rcParams["text.latex.preview"] = preview

subplot = maxes.subplot_class_factory(Axes)

def test_window_extent(ax, usetex, preview):
    va = "baseline"
    ax.xaxis.set_visible(False)
    ax.yaxis.set_visible(False)

    #t = ax.text(0., 0., r"mlp", va="baseline", size=150)
    text_kw = dict(va=va,
                    size=50,
                    bbox=dict(pad=0., ec="k", fc="none"))

    test_strings = ["lg", r"\frac{1}{2}\pi$",
                    r"p^{3^A}$", r"p_{3_2}$"]

    ax.axvline(0, color="r")
    for i, s in enumerate(test_strings):
        ax.axhline(i, color="r")
        ax.text(0., 3 - i, s, **text_kw)

    ax.set_xlim(-0.1, 1.1)
    ax.set_ylim(-.8, 3.9)
    ax.set_title("usetex=%s
    preview=%s" % (str(usetex), str(preview)))

fig = plt.figure(figsize=(2.*3, 6.5))
```

Matplotlib, Release 2.0.2
for i, usetex, preview in [[0, False, False],
[1, True, False],
[2, True, True]]:
    ax = subplot(fig, 1, 3, i + 1, usetex=usetex, preview=preview)
    fig.add_subplot(ax)
    fig.subplots_adjust(top=0.85)
    test_window_extent(ax, usetex=usetex, preview=preview)

    plt.draw()
    plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

99.203 pylab_examples example code: usetex_demo.py

import matplotlib
matplotlib.rc('text', usetex=True)
import matplotlib.pyplot as plt
import numpy as np
# interface tracking profiles

N = 500
delta = 0.6
X = np.linspace(-1, 1, N)
plt.plot(X, (1 - np.tanh(4.*X/delta))/2, # phase field tanh profiles
    X, (X + 1)/2, # level set distance function
    X, (1.4 + np.tanh(4.*X/delta))/4, # composition profile
    X, X < 0, 'k--', # sharp interface
)

# legend
plt.legend(('phase field', 'level set', 'composition', 'sharp interface'), shadow=True, loc=(0.01, 0.55))
ltext = plt.gca().get_legend().get_texts()
plt.setp(ltext[0], fontsize=20)
plt.setp(ltext[1], fontsize=20)
plt.setp(ltext[2], fontsize=20)
plt.setp(ltext[3], fontsize=20)

# the arrow
height = 0.1
offset = 0.02
plt.plot((-delta / 2., delta / 2), (height, height), 'k', linewidth=2)
plt.plot((-delta / 2, -delta / 2 + offset * 2), (height, height - offset), 'k', linewidth=2)
plt.plot((-delta / 2, -delta / 2 + offset * 2), (height, height + offset), 'k', linewidth=2)
plt.plot((delta / 2, delta / 2 - offset * 2), (height, height - offset), 'k', linewidth=2)
plt.plot((delta / 2, delta / 2 - offset * 2), (height, height + offset), 'k', linewidth=2)
plt.text(-0.06, height - 0.06, r'$\delta$', {'color': 'k', 'fontsize': 24})

# X-axis label
plt.xticks((-1, 0, 1), ('-1', '0', '1'), color='k', size=20)

# Left Y-axis labels
plt.ylabel(r'$\bf{phase field} \ \phi$', {'color': 'b', 'fontsize': 20})
plt.yticks((0, 0.5, 1), ('0', '.5', '1'), color='k', size=20)

# Right Y-axis labels
plt.text(1.05, 0.5, r'$\bf{level set} \ \phi$', {'color': 'g', 'fontsize': 20},
    horizontalalignment='left',
    verticalalignment='center',
    rotation=90,
    clip_on=False)
plt.text(1.01, 0.02, "-1", {'color': 'k', 'fontsize': 20})
plt.text(1.01, 0.98, "1", {'color': 'k', 'fontsize': 20})
plt.text(1.01, 0.48, "0", {'color': 'k', 'fontsize': 20})
# level set equations
plt.text(0.1, 0.85,
    r'$|\nabla \phi| = 1,$ \newline $\frac{\partial \phi}{\partial t' + U|\nabla \phi| = 0$,
    {'color': 'g', 'fontsize': 20})

# phase field equations
plt.text(0.2, 0.15,
    r'$\mathcal{F} = \int f(\phi, c) \, dV,$ \newline $\frac{\delta \mathcal{F}}{\delta \phi} = -M_{\phi}$',
    {'color': 'b', 'fontsize': 20})

# these went wrong in pdf in a previous version
plt.text(-.9, .42, r'$\gamma: \gamma$', {'color': 'r', 'fontsize': 20})
plt.text(-.9, .36, r'$\Omega: \Omega$', {'color': 'b', 'fontsize': 20})

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

99.204 pylab_examples example code:usetex_fonteffects.py

Usetex font effects

Nimbus Roman No9 L (extended)

Nimbus Roman No9 L (condensed)

**Nimbus Roman No9 L (slanted)**

**Nimbus Roman No9 L Italic (real italics for con**

Nimbus Roman No9 L
# This script demonstrates that font effects specified in your pdftex.map
# are now supported in pdf usetex.

import matplotlib
import matplotlib.pyplot as plt
matplotlib.rc('text', usetex=True)

def setfont(font):
    return r'\font\a%s at 14pt\a ' % font

for y, font, text in zip(range(5),
                       ['ptmr8r', 'ptmri8r', 'ptmro8r', 'ptmr8rn', 'ptmrr8re'],
                       ['Nimbus Roman No9 L ' + x for x in
                         ['', 'Italics (real italics for comparison)',
                          '(slanted)', '(condensed)', '(extended)']]):
    plt.text(0, y, setfont(font) + text)

plt.ylim(-1, 5)
plt.xlim(-0.2, 0.6)
plt.setp(plt.gca(), frame_on=False, xticks=(), yticks=())
plt.title('Usetex font effects')
plt.savefig('usetex_fonteffects.pdf')

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

99.205 pylab_examples example code: vline_hline_demo.py

Small demonstration of the hlines and vlines plots.
import matplotlib.pyplot as plt
import numpy as np
import numpy.random as rnd

def f(t):
    s1 = np.sin(2 * np.pi * t)
    e1 = np.exp(-t)
    return np.absolute((s1 * e1)) + .05

t = np.arange(0.0, 5.0, 0.1)
s = f(t)
nse = rnd.normal(0.0, 0.3, t.shape) * s

fig = plt.figure(figsize=(12, 6))
vax = fig.add_subplot(121)
hax = fig.add_subplot(122)

vax.plot(t, s + nse, '^')
vax.vlines(t, [0], s)
vax.set_xlabel('time (s)')
vax.set_title('Vertical lines demo')

hax.plot(s + nse, t, '^')
hax.hlines(t, [0], s, lw=2)
hax.set_xlabel('time (s)')
hax.set_title('Horizontal lines demo')

plt.show()

---

Keywords: python, matplotlib, pylab, example, codex (see `Search examples`)

### 99.206 `pylab_examples` example code: `webapp_demo.py`

[source code]

```python
# -*- noplot -*-

This example shows how to use the agg backend directly to create images, which may be of use to web application developers who want full control over their code without using the pyplot interface to manage figures, figure closing etc.

.. note::

   It is not necessary to avoid using the pyplot interface in order to create figures without a graphical front-end - simply setting the backend to "Agg" would be sufficient.
```
It is also worth noting that, because matplotlib can save figures to file-like
object, matplotlib can also be used inside a cgi-script *without* needing to
write a figure to disk.

```
from matplotlib.backends.backend_agg import FigureCanvasAgg
from matplotlib.figure import Figure
import numpy as np

def make_fig():
    """
    Make a figure and save it to "webagg.png".
    """
    fig = Figure()
    ax = fig.add_subplot(1, 1, 1)
    ax.plot([1, 2, 3], 'ro--', markersize=12, markerfacecolor='g')

    # make a translucent scatter collection
    x = np.random.rand(100)
    y = np.random.rand(100)
    area = np.pi * (10 * np.random.rand(100)) ** 2  # 0 to 10 point radii
    c = ax.scatter(x, y, area)
    c.set_alpha(0.5)

    # add some text decoration
    ax.set_title('My first image')
    ax.set_xlabel('Some numbers')
    ax.set_xticks((.2, .4, .6, .8))
    labels = ax.set_xticklabels(('Bill', 'Fred', 'Ted', 'Ed'))

    # To set object properties, you can either iterate over the
    # objects manually, or define you own set command, as in setapi
    # above.
    for label in labels:
        label.set_rotation(45)
        label.set_fontsize(12)

    FigureCanvasAgg(fig).print_png('webapp.png', dpi=150)

make_fig()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
```python
import matplotlib.pyplot as plt
import numpy as np

np.random.seed(0)

x, y = np.random.randn(2, 100)
fig = plt.figure()
ax1 = fig.add_subplot(211)
ax1.xcorr(x, y, usevlines=True, maxlags=50, normed=True, lw=2)
ax1.grid(True)
ax1.axhline(0, color='black', lw=2)

ax2 = fig.add_subplot(212, sharex=ax1)
ax2.acorr(x, usevlines=True, normed=True, maxlags=50, lw=2)
ax2.grid(True)
ax2.axhline(0, color='black', lw=2)

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
```
99.208   pylab_examples example code: zorder_demo.py

![Diagram showing lines on top of dots and dots on top of lines.]

Lines on top of dots

Dots on top of lines

0.0 0.2 0.4 0.6 0.8 1.0
0.00
0.25
0.50
0.75
1.00

Lines on top of dots

Dots on top of lines

0.0 0.2 0.4 0.6 0.8 1.0
0.00
0.25
0.50
0.75
1.00
The default drawing order for axes is patches, lines, text. This order is determined by the zorder attribute. The following defaults are set:

<table>
<thead>
<tr>
<th>Artist</th>
<th>Z-order</th>
</tr>
</thead>
<tbody>
<tr>
<td>Patch / PatchCollection</td>
<td>1</td>
</tr>
<tr>
<td>Line2D / LineCollection</td>
<td>2</td>
</tr>
<tr>
<td>Text</td>
<td>3</td>
</tr>
</tbody>
</table>

You can change the order for individual artists by setting the zorder. Any individual plot() call can set a value for the zorder of that particular item.

In the first subplot below, the lines are drawn above the patch collection from the scatter, which is the default.

In the subplot below, the order is reversed.

The second figure shows how to control the zorder of individual lines.

```python
import matplotlib.pyplot as plt
import numpy as np

x = np.random.random(20)
y = np.random.random(20)
```
# Lines on top of scatter
plt.figure()
plt.subplot(211)
plt.plot(x, y, 'r', lw=3)
plt.scatter(x, y, s=120)
plt.title('Lines on top of dots')

# Scatter plot on top of lines
plt.subplot(212)
plt.plot(x, y, 'r', zorder=1, lw=3)
plt.scatter(x, y, s=120, zorder=2)
plt.title('Dots on top of lines')

# A new figure, with individually ordered items
x = np.linspace(0, 2*np.pi, 100)
plt.figure()
plt.plot(x, np.sin(x), linewidth=10, color='black', label='zorder=10', zorder=10)  # on top
plt.plot(x, np.cos(1.3*x), linewidth=10, color='red', label='zorder=1', zorder=1)    # on top
plt.plot(x, np.sin(2.1*x), linewidth=10, color='green', label='zorder=3', zorder=3)  # on bottom
plt.axhline(0, linewidth=10, color='blue', label='zorder=2', zorder=2)
plt.title('Custom order of elements')
l = plt.legend()
l.set_zorder(20)  # put the legend on top
plt.show()
100.1 pyplots example code: align_ylabels.py

[source code]

```python
import numpy as np
import matplotlib.pyplot as plt

box = dict(facecolor='yellow', pad=5, alpha=0.2)

fig = plt.figure()
fig.subplots_adjust(left=0.2, wspace=0.6)

# Fixing random state for reproducibility
np.random.seed(19680801)

ax1 = fig.add_subplot(221)
ax1.plot(2000*np.random.rand(10))
ax1.set_title('ylabels not aligned')
ax1.set_ylabel('misaligned 1', bbox=box)
ax1.set_ylim(0, 2000)

ax2 = fig.add_subplot(222)
ax2.set_title('ylabels aligned')
ax2.plot(2000*np.random.rand(10))
ax2.set_ylabel('aligned 1', bbox=box)
ax2.yaxis.set_label_coords(-0.3, 0.5)
ax2.set_ylim(0, 2000)

ax3 = fig.add_subplot(223)
ax3.set_ylabel('misaligned 2', bbox=box)
ax3.plot(np.random.rand(10))

ax4 = fig.add_subplot(224)
ax4.plot(np.random.rand(10))
ax4.set_ylabel('aligned 2', bbox=box)
ax4.yaxis.set_label_coords(-0.3, 0.5)

plt.show()
```
100.2 pyplots example code: annotate_transform.py

[source code]

```python
import numpy as np
import matplotlib.pyplot as plt

x = np.arange(0, 10, 0.005)
y = np.exp(-x/2.) * np.sin(2*np.pi*x)

fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(x, y)
ax.set_xlim(0, 10)
ax.set_ylim(-1, 1)

xdata, ydata = 5, 0
xdisplay, ydisplay = ax.transData.transform_point((xdata, ydata))
bbox = dict(boxstyle="round", fc="0.8")
arrowprops = dict(
    arrowstyle = "->",
    connectionstyle = "angle,angleA=0,angleB=90,rad=10")

offset = 72
ax.annotate('data = (%.1f, %.1f)'%(xdata, ydata),
            (xdata, ydata), xytext=(-2*offset, offset),
            textcoords='offset points',
            bbox=bbox, arrowprops=arrowprops)

disp = ax.annotate('display = (%.1f, %.1f)'%(xdisplay, ydisplay),
                   (xdisplay, ydisplay), xytext=(0.5*offset, -offset),
                   xycoords='figure pixels',
                   textcoords='offset points',
                   bbox=bbox, arrowprops=arrowprops)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

100.3 pyplots example code: annotation_basic.py

[source code]

```python
import numpy as np
import matplotlib.pyplot as plt
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
```python
fig = plt.figure()
ax = fig.add_subplot(111)

t = np.arange(0.0, 5.0, 0.01)
s = np.cos(2*np.pi*t)
line, = ax.plot(t, s, lw=2)

ax.annotate('local max', xy=(2, 1), xytext=(3, 1.5),
             arrowprops=dict(facecolor='black', shrink=0.05),
            )
ax.set_ylim(-2,2)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

### 100.4 `pyplots` example code: `annotation_polar.py`

```python
import numpy as np
import matplotlib.pyplot as plt

fig = plt.figure()
ax = fig.add_subplot(111, polar=True)

r = np.arange(0,1,0.001)
theta = 2*2*np.pi*r
line, = ax.plot(theta, r, color='#ee8d18', lw=3)

ind = 800
thisr, thistheta = r[ind], theta[ind]
ax.plot([thistheta], [thisr], 'o')
ax.annotate('a polar annotation',
            xy=(thistheta, thisr), # theta, radius
            xytext=(0.05, 0.05), # fraction, fraction
            textcoords='figure fraction',
            arrowprops=dict(facecolor='black', shrink=0.05),
            horizontalalignment='left',
            verticalalignment='bottom',
           )

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

### 100.5 `pyplots` example code: `auto_subplots_adjust.py`

```python
[source code]
```
```
import matplotlib.pyplot as plt
import matplotlib.transforms as mtransforms
fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(range(10))
ax.set_yticks((2,5,7))
labels = ax.set_yticklabels(('really, really, really', 'long', 'labels'))

def on_draw(event):
    bboxes = []
    for label in labels:
        bbox = label.get_window_extent()
        # the figure transform goes from relative coords->pixels and we
        # want the inverse of that
        bboxi = bbox.inverse_transformed(fig.transFigure)
        bboxes.append(bboxi)

    # this is the bbox that bounds all the bboxes, again in relative
    # figure coords
    bbox = mtransforms.Bbox.union(bboxes)
    if fig.subplotpars.left < bbox.width:
        # we need to move it over
        fig.subplots_adjust(left=1.1*bbox.width)  # pad a little
        fig.canvas.draw()
    return False

fig.canvas.mpl_connect('draw_event', on_draw)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

100.6 pyplots example code: boxplot_demo.py

```
#!/usr/bin/python

# Example boxplot code

import numpy as np
import matplotlib.pyplot as plt

# Fixing random state for reproducibility
np.random.seed(19680801)

# fake up some data
spread = np.random.rand(50) * 100
center = np.ones(25) * 50
```
flier_high = np.random.rand(10) * 100 + 100
flier_low = np.random.rand(10) * -100
data = np.concatenate((spread, center, flier_high, flier_low), 0)

fig1, ax1 = plt.subplots()
ax1.set_title('Basic Plot')
ax1.boxplot(data)

fig2, ax2 = plt.subplots()
ax2.set_title('Notched boxes')
ax2.boxplot(data, notch=True)

green_diamond = dict(markerfacecolor='g', marker='D')
fig3, ax3 = plt.subplots()
ax3.set_title('Changed Outlier Symbols')
ax3.boxplot(data, flierprops=green_diamond)

fig4, ax4 = plt.subplots()
ax4.set_title('Hide Outlier Points')
ax4.boxplot(data, showfliers=False)

red_square = dict(markerfacecolor='r', marker='s')
fig5, ax5 = plt.subplots()
ax5.set_title('Horizontal Boxes')
ax5.boxplot(data, vert=False, flierprops=red_square)

fig6, ax6 = plt.subplots()
ax6.set_title('Shorter Whisker Length')
ax6.boxplot(data, flierprops=red_square, vert=False, whis=0.75)

# fake up some more data
spread = np.random.rand(50) * 100
center = np.ones(25) * 40
flier_high = np.random.rand(10) * 100 + 100
flier_low = np.random.rand(10) * -100
d2 = np.concatenate((spread, center, flier_high, flier_low), 0)
data.shape = (-1, 1)
d2.shape = (-1, 1)

# Making a 2-D array only works if all the columns are the same length. If they are not, then use a list instead.
# This is actually more efficient because boxplot converts a 2-D array into a list of vectors internally anyway.
data = [data, d2, d2[::-2,0]]
fig7, ax7 = plt.subplots()
ax7.set_title('Multiple Samples with Different sizes')
ax7.boxplot(data)

plt.show()
100.7 `pyplots` example code: compound_path_demo.py

[source code]

```python
import numpy as np

import matplotlib.pyplot as plt
import matplotlib.patches as patches
import matplotlib.path as path

fig = plt.figure()
ax = fig.add_subplot(111)

# Fixing random state for reproducibility
np.random.seed(19680801)

# histogram our data with numpy
data = np.random.randn(1000)
n, bins = np.histogram(data, 100)

# get the corners of the rectangles for the histogram
left = np.array(bins[:-1])
right = np.array(bins[1:])
bottom = np.zeros(len(left))
top = bottom + n
nrects = len(left)

nverts = nrects*(1+3+1)
verts = np.zeros((nverts, 2))
codes = np.ones(nverts, int) * path.Path.LINETO

codes[0::5] = path.Path.MOVETO

codes[4::5] = path.Path.CLOSEPOLY

verts[0::5,0] = left
verts[0::5,1] = bottom
verts[1::5,0] = left
verts[1::5,1] = top
verts[2::5,0] = right
verts[2::5,1] = top
verts[3::5,0] = right
verts[3::5,1] = bottom

barpath = path.Path(verts, codes)
patch = patches.PathPatch(barpath, facecolor='green', edgecolor='yellow', alpha=0.5)
ax.add_patch(patch)

ax.set_xlim(left[0], right[-1])
ax.set_ylim(bottom.min(), top.max())

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
100.8 pyplots example code: dollar_ticks.py

[source code]

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker

# Fixing random state for reproducibility
np.random.seed(19680801)

fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(100*np.random.rand(20))

formatter = ticker.FormatStrFormatter(r'$%1.2f$')
ax.yaxis.set_major_formatter(formatter)

for tick in ax.yaxis.get_major_ticks():
    tick.label1On = False
    tick.label2On = True
    tick.label2.set_color('green')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

100.9 pyplots example code: fig_axes_customize_simple.py

[source code]

```python
import numpy as np
import matplotlib.pyplot as plt

# plt.figure creates a matplotlib.figure.Figure instance
fig = plt.figure()
rect = fig.patch # a rectangle instance
rect.set_facecolor('lightgoldenrodyellow')

ax1 = fig.add_axes([0.1, 0.3, 0.4, 0.4])
rect = ax1.patch
rect.set_facecolor('lightslategray')

for label in ax1.xaxis.get_ticklabels():
    # label is a Text instance
    label.set_color('red')
    label.set_rotation(45)
    label.set_fontsize(16)

for line in ax1.yaxis.get_ticklines():
```

100.8. pyplots example code: dollar_ticks.py 2775
# line is a Line2D instance
line.set_color('green')
line.set_markersize(25)
line.set_markeredgewidth(3)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

### 100.10 pyplots example code: fig_axes_labels_simple.py

[source code]

```python
import numpy as np
import matplotlib.pyplot as plt

fig = plt.figure()
fig.subplots_adjust(top=0.8)
ax1 = fig.add_subplot(211)
ax1.set_ylabel('volts')
ax1.set_title('a sine wave')

t = np.arange(0.0, 1.0, 0.01)
s = np.sin(2*np.pi*t)
line, = ax1.plot(t, s, color='blue', lw=2)

# Fixing random state for reproducibility
np.random.seed(19680801)

ax2 = fig.add_axes([0.15, 0.1, 0.7, 0.3])
n, bins, patches = ax2.hist(np.random.randn(1000), 50,
    facecolor='yellow', edgecolor='yellow')
ax2.set_xlabel('time (s)')
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

### 100.11 pyplots example code: fig_x.py

[source code]

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.lines as lines

fig = plt.figure()
```
```python
l1 = lines.Line2D([0, 1], [0, 1], transform=fig.transFigure, figure=fig)
l2 = lines.Line2D([0, 1], [1, 0], transform=fig.transFigure, figure=fig)
fig.lines.extend([l1, l2])
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

### 100.12 pyplots example code: pyplot_annotate.py

[source code]

```python
import numpy as np
import matplotlib.pyplot as plt

ax = plt.subplot(111)
t = np.arange(0.0, 5.0, 0.01)
s = np.cos(2*np.pi*t)
line, = plt.plot(t, s, lw=2)
plt.annotate('local max', xy=(2, 1), xytext=(3, 1.5),
            arrowprops=dict(facecolor='black', shrink=0.05),
            )
plt.ylim(-2,2)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

### 100.13 pyplots example code: pyplot_formatstr.py

[source code]

```python
import matplotlib.pyplot as plt
plt.plot([1,2,3,4], [1,4,9,16], 'ro')
plt.axis([0, 6, 0, 20])
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

### 100.14 pyplots example code: pyplot_mathtext.py

[source code]
```python
import numpy as np
import matplotlib.pyplot as plt

t = np.arange(0.0, 2.0, 0.01)
s = np.sin(2*np.pi*t)

plt.plot(t, s)
plt.title(r'$\alpha_i > \beta_i$', fontsize=20)
plt.text(1, -0.6, r'$\sum_{i=0}^{\infty} x_i$', fontsize=20)
plt.text(0.6, 0.6, r'$\mathcal{A}\sin(2 \omega t)$', fontsize=20)
plt.xlabel('time (s)')
plt.ylabel('volts (mV)')
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

### 100.15 pyplots example code: pyplot_scales.py

```python
import numpy as np
import matplotlib.pyplot as plt

# Fixing random state for reproducibility
np.random.seed(19680801)

# make up some data in the interval [0, 1[
y = np.random.normal(loc=0.5, scale=0.4, size=1000)
y = y[(y > 0) & (y < 1)]
y.sort()
x = np.arange(len(y))

# plot with various axes scales
plt.figure(1)

# linear
plt.subplot(221)
plt.plot(x, y)
plt.yscale('linear')
plt.title('linear')
plt.grid(True)

# log
plt.subplot(222)
plt.plot(x, y)
plt.yscale('log')
plt.title('log')
plt.grid(True)
```
# symmetric log
plt.subplot(223)
plt.plot(x, y - y.mean())
plt.yscale('symlog', linthreshy=0.01)
plt.title('symlog')
plt.grid(True)

# logit
plt.subplot(224)
plt.plot(x, y)
plt.yscale('logit')
plt.title('logit')
plt.grid(True)

# Format the minor tick labels of the y-axis into empty strings with
# `NullFormatter`, to avoid cumbering the axis with too many labels.
plt.gca().yaxis.set_minor_formatter(NullFormatter())

# Adjust the subplot layout, because the logit one may take more space
# than usual, due to y-tick labels like "1 - 10^{1-3}"
plt.subplots_adjust(top=0.92, bottom=0.08, left=0.10, right=0.95, hspace=0.25, wspace=0.35)

plt.show()
Matplotlib, Release 2.0.2

x = mu + sigma * np.random.randn(10000)

# the histogram of the data
n, bins, patches = plt.hist(x, 50, normed=1, facecolor='g', alpha=0.75)

plt.xlabel('Smarts')
plt.ylabel('Probability')
plt.title('Histogram of IQ')
plt.text(60, .025, r'$\mu=100, \sigma=15$')
plt.axis([40, 160, 0, 0.03])
plt.grid(True)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

100.18 pyplots example code: pyplot_three.py

[source code]

import numpy as np
import matplotlib.pyplot as plt

# evenly sampled time at 200ms intervals
T = np.arange(0., 5., 0.2)

# red dashes, blue squares and green triangles
plt.plot(T, T, 'r--', T, T**2, 'bs', T, T**3, 'g^')
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

100.19 pyplots example code: pyplot_two_subplots.py

[source code]

import numpy as np
import matplotlib.pyplot as plt

def f(t):
    return np.exp(-t) * np.cos(2*np.pi*t)

t1 = np.arange(0.0, 5.0, 0.1)
t2 = np.arange(0.0, 5.0, 0.02)

plt.figure(1)
plt.subplot(211)
plt.plot(t1, f(t1), 'bo', t2, f(t2), 'k')
```python
plt.subplot(212)
plt.plot(t2, np.cos(2*np.pi*t2), 'r--')
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

## 100.20 pyplots example code: tex_demo.py

[source code]

```python
""
Demo of TeX rendering.

You can use TeX to render all of your matplotlib text if the rc
parameter text.usetex is set. This works currently on the agg and ps
backends, and requires that you have tex and the other dependencies
described at http://matplotlib.org/users/usetex.html
properly installed on your system. The first time you run a script
you will see a lot of output from tex and associated tools. The next
time, the run may be silent, as a lot of the information is cached in
~/.tex.cache

""

import numpy as np
import matplotlib.pyplot as plt

# Example data
t = np.arange(0.0, 1.0 + 0.01, 0.01)
s = np.cos(4 * np.pi * t) + 2

plt.rc('text', usetex=True)
plt.rc('font', family='serif')
plt.plot(t, s)
plt.xlabel(r'$\textbf{time}$ (s)')
plt.ylabel(r'$\textit{voltage}$ (mV)', fontsize=16)
plt.title(r'\TeX\ is Number 
$\sum_{n=1}^{\infty} \frac{-e^{i\pi}}{2^n}$!', fontsize=16, color='gray')
# Make room for the ridiculously large title.
plt.subplots_adjust(top=0.8)
plt.savefig('tex_demo')
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
100.21 pylots example code: text_commands.py

[source code]

```python
# -*- coding: utf-8 -*-
import matplotlib.pyplot as plt

fig = plt.figure()
fig.suptitle('bold figure suptitle', fontsize=14, fontweight='bold')

ax = fig.add_subplot(111)
fig.subplots_adjust(top=0.85)
ax.set_title('axes title')
ax.set_xlabel('xlabel')
ax.set_ylabel('ylabel')

ax.text(3, 8, 'boxed italics text in data coords', style='italic',
        bbox={'facecolor': 'red', 'alpha': 0.5, 'pad': 10})

ax.text(2, 6, r'an equation: $E=mc^2$', fontsize=15)

ax.text(3, 2, u'unicode: Institut f\374r Festk\366rperphysik')

ax.text(0.95, 0.01, 'colored text in axes coords',
        verticalalignment='bottom', horizontalalignment='right',
        transform=ax.transAxes, color='green', fontsize=15)

ax.plot([2], [1], 'o')
ax.annotate('annotate', xy=(2, 1), xytext=(3, 4),
            arrowprops=dict(facecolor='black', shrink=0.05))

ax.axis([0, 10, 0, 10])
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

100.22 pylots example code: text_layout.py

[source code]

```python
import matplotlib.pyplot as plt
import matplotlib.patches as patches

# build a rectangle in axes coords
left, width = .25, .5
bottom, height = .25, .5
right = left + width
```
top = bottom + height

fig = plt.figure()
ax = fig.add_axes([0,0,1,1])

# axes coordinates are 0,0 is bottom left and 1,1 is upper right
p = patches.Rectangle(
    (left, bottom), width, height,
    fill=False, transform=ax.transAxes, clip_on=False
)

ax.add_patch(p)

ax.text(left, bottom, 'left top',
    horizontalalignment='left',
    verticalalignment='top',
    transform=ax.transAxes)

ax.text(left, bottom, 'left bottom',
    horizontalalignment='left',
    verticalalignment='bottom',
    transform=ax.transAxes)

ax.text(right, top, 'right bottom',
    horizontalalignment='right',
    verticalalignment='bottom',
    transform=ax.transAxes)

ax.text(right, top, 'right top',
    horizontalalignment='right',
    verticalalignment='top',
    transform=ax.transAxes)

ax.text(right, bottom, 'center top',
    horizontalalignment='center',
    verticalalignment='top',
    transform=ax.transAxes)

ax.text(left, 0.5*(bottom+top), 'right center',
    horizontalalignment='right',
    verticalalignment='center',
    rotation='vertical',
    transform=ax.transAxes)

ax.text(left, 0.5*(bottom+top), 'left center',
    horizontalalignment='left',
    verticalalignment='center',
    rotation='vertical',
    transform=ax.transAxes)

ax.text(0.5*(left+right), 0.5*(bottom+top), 'middle',
    horizontalalignment='center',
    verticalalignment='center',
    rotation='vertical',
    transform=ax.transAxes)
from mpl_toolkits.mplot3d.axes3d import Axes3D
from matplotlib import cm
# from matplotlib.ticker import LinearLocator, FixedLocator, FormatStrFormatter
import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure()

ax = fig.add_subplot(1, 2, 1, projection='3d')
X = np.arange(-5, 5, 0.25)
Y = np.arange(-5, 5, 0.25)
X, Y = np.meshgrid(X, Y)
R = np.sqrt(X**2 + Y**2)
Z = np.sin(R)
surf = ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap=cm.jet,
                      linewidth=0, antialiased=False)
ax.set_zlim3d(-1.01, 1.01)
# ax.w_zaxis.set_major_locator(LinearLocator(10))
# ax.w_zaxis.set_major_formatter(FormatStrFormatter('%.03f'))
fig.colorbar(surf, shrink=0.5, aspect=5)

from mpl_toolkits.mplot3d.axes3d import get_test_data
ax = fig.add_subplot(1, 2, 2, projection='3d')
X, Y, Z = get_test_data(0.05)
ax.plot_wireframe(X, Y, Z, rstride=10, cstride=10)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

100.24 pyplots example code: whats_new_98_4_fancy.py

[source code]

```python
import matplotlib.patches as mpatch
import matplotlib.pyplot as plt

figheight = 8
fig = plt.figure(1, figsize=(9, figheight), dpi=80)
fontsize = 0.4 * fig.dpi

def make_boxstyles(ax):
    styles = mpatch.BoxStyle.get_styles()
    for i, (stylename, styleclass) in enumerate(sorted(styles.items())):
        ax.text(0.5, (float(len(styles)) - 0.5 - i)/len(styles), stylename,
                ha="center",
                size=fontsize,
                transform=ax.transAxes,
                bbox=dict(boxstyle=stylename, fc="w", ec="k"))

def make_arrowstyles(ax):
    styles = mpatch.ArrowStyle.get_styles()
    ax.set_xlim(0, 4)
    ax.set_ylim(0, figheight)
    for i, (stylename, styleclass) in enumerate(sorted(styles.items())):
        y = (float(len(styles)) - 0.25 - i) / figheight
        p = mpatch.Circle((3.2, y), 0.2, fc="w")
        ax.add_patch(p)
        ax.annotate(stylename, (3.2, y),
                    xycoords="figure fraction", textcoords="figure fraction",
                    ha="right", va="center",
                    size=fontsize,
                    arrowprops=dict(arrowstyle=stylename,
                                    shrinkB=p,
                                    shrinkA=5,
                                    shrinkB=5,
                                    fc="w", ec="k",
                                    connectionstyle="arc3,rad=-0.05",
                                    ),
                                    bbox=dict(boxstyle="square", fc="w"))
```

100.24. pyplots example code: whats_new_98_4_fancy.py 2785
ax.xaxis.set_visible(False)
ax.yaxis.set_visible(False)

ax1 = fig.add_subplot(121, frameon=False, xticks=[], yticks=[])
make_boxstyles(ax1)

ax2 = fig.add_subplot(122, frameon=False, xticks=[], yticks=[])
make_arrowstyles(ax2)

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

100.25 pyplots example code: whats_new_98_4_fill_between.py

[source code]

#!/usr/bin/env python
import matplotlib.mlab as mlab
import matplotlib.pyplot as plt
import numpy as np

x = np.arange(0.0, 2, 0.01)
y1 = np.sin(2*np.pi*x)
y2 = 1.2*np.sin(4*np.pi*x)

fig, ax = plt.subplots()
ax.plot(x, y1, x, y2, color='black')
ax.fill_between(x, y1, y2, where=y2>y1, facecolor='green')
ax.fill_between(x, y1, y2, where=y2<=y1, facecolor='red')
ax.set_title('fill between where')

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

100.26 pyplots example code: whats_new_98_4_legend.py

[source code]

import matplotlib.pyplot as plt
import numpy as np
import matplotlib.pyplot as plt

ax = plt.subplot(111)
t1 = np.arange(0.0, 1.0, 0.01)
for n in [1, 2, 3, 4]:
    plt.plot(t1, t1**n, label="n=%d"%(n,))

leg = plt.legend(loc='best', ncol=2, mode="expand", shadow=True, fancybox=True)
leg.get_frame().set_alpha(0.5)

plt.show()
r, g, b = get_rgb()
kwreqs = dict(origin="lower", interpolation="nearest")
ax.imshow_rgb(r, g, b, **kwreqs)

ax.RGB.set_xlim(0., 9.5)
ax.RGB.set_ylim(0.9, 10.6)

plt.draw()
plt.show()
```python
if loc in spines:
    spine.set_position(('outward', 10))  # outward by 10 points
else:
    spine.set_color('none')  # don't draw spine

# turn off ticks where there is no spine
if 'left' in spines:
    ax.yaxis.set_ticks_position('left')
else:
    # no yaxis ticks
    ax.yaxis.set_ticks([])

if 'bottom' in spines:
    ax.xaxis.set_ticks_position('bottom')
else:
    # no xaxis ticks
    ax.xaxis.set_ticks([])

fig = plt.figure()
x = np.linspace(0, 2*np.pi, 100)
y = 2*np.sin(x)

ax = fig.add_subplot(2, 2, 1)
ax.plot(x, y)
adjust_spines(ax, ['left'])

ax = fig.add_subplot(2, 2, 2)
ax.plot(x, y)
adjust_spines(ax, [])

ax = fig.add_subplot(2, 2, 3)
ax.plot(x, y)
adjust_spines(ax, ['left', 'bottom'])

ax = fig.add_subplot(2, 2, 4)
ax.plot(x, y)
adjust_spines(ax, ['bottom'])

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Illustrate the scale transformations applied to axes, e.g. log, symlog, logit.

```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.ticker import NullFormatter

np.random.seed(1)
# make up some data in the interval ]0, 1[
```
```python
y = np.random.normal(loc=0.5, scale=0.4, size=1000)
y = y[(y > 0) & (y < 1)]
y.sort()
x = np.arange(len(y))

# plot with various axes scales
fig, axs = plt.subplots(2, 2, sharex=True)
fig.subplots_adjust(left=0.08, right=0.98, wspace=0.3)

# linear
ax = axs[0, 0]
ax.plot(x, y)
ax.set_yscale('linear')
ax.set_title('linear')
ax.grid(True)

# log
ax = axs[0, 1]
ax.plot(x, y)
ax.set_yscale('log')
ax.set_title('log')
ax.grid(True)

# symmetric log
ax = axs[1, 1]
ax.plot(x, y - y.mean())
ax.set_yscale('symlog', linthreshy=0.02)
ax.set_title('symlog')
ax.grid(True)

# logit
ax = axs[1, 0]
ax.plot(x, y)
ax.set_yscale('logit')
ax.set_title('logit')
ax.grid(True)
ax.yaxis.set_minor_formatter(NullFormatter())

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
102.1 shapes_and_collections example code: artist_reference.py

""
Reference for matplotlib artists

This example displays several of matplotlib's graphics primitives (artists) drawn using matplotlib API. A full list of artists and the documentation is available at http://matplotlib.org/api/artist_api.html.
"""
import matplotlib.pyplot as plt
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.path as mpath
import matplotlib.lines as mlines
import matplotlib.patches as mpatches
from matplotlib.collections import PatchCollection

def label(xy, text):
    y = xy[1] - 0.15  # shift y-value for label so that it's below the artist
    plt.text(xy[0], y, text, ha="center", family='sans-serif', size=14)

fig, ax = plt.subplots()
# create 3x3 grid to plot the artists
grid = np.mgrid[0.2:0.8:3j, 0.2:0.8:3j].reshape(2, -1).T
patches = []

# add a circle
circle = mpatches.Circle(grid[0], 0.1, ec="none")
patches.append(circle)
label(grid[0], "Circle")

# add a rectangle
rect = mpatches.Rectangle(grid[1] - [0.025, 0.05], 0.05, 0.1, ec="none")
patches.append(rect)
label(grid[1], "Rectangle")

# add a wedge
wedge = mpatches.Wedge(grid[2], 0.1, 30, 270, ec="none")
patches.append(wedge)
label(grid[2], "Wedge")

# add a Polygon
polygon = mpatches.RegularPolygon(grid[3], 5, 0.1)
patches.append(polygon)
label(grid[3], "Polygon")

# add an ellipse
ellipse = mpatches.Ellipse(grid[4], 0.2, 0.1)
patches.append(ellipse)
label(grid[4], "Ellipse")

# add an arrow
arrow = mpatches.Arrow(grid[5, 0] - 0.05, grid[5, 1] - 0.05, 0.1, 0.1, width=0.1)
patches.append(arrow)
label(grid[5], "Arrow")

# add a path patch
Path = mpath.Path
path_data = [
    (Path.MOVETO, [0.018, -0.11]),
    (Path.CURVE4, [-0.031, -0.051]),
    (Path.CURVE4, [-0.115, 0.073]),
    (Path.CURVE4, [-0.03, 0.073]),
    (Path.LINETO, [-0.011, 0.039]),
    (Path.CURVE4, [0.043, 0.121]),
    (Path.CURVE4, [0.075, -0.005]),
    (Path.CURVE4, [0.035, -0.027]),
    (Path.CLOSEPOLY, [0.018, -0.11])
]
codes, verts = zip(*path_data)
path = mpath.Path(verts + grid[6], codes)
pathpatch = mpatches.PathPatch(path)
patches.append(pathpatch)
label(grid[6], "PathPatch")

# add a fancy box
fancybox = mpatches.FancyBboxPatch(
    grid[7] - [0.025, 0.05], 0.05, 0.1,
    boxstyle=mpatches.BoxStyle("Round", pad=0.02))
patches.append(fancybox)
label(grid[7], "FancyBboxPatch")

# add a line
x, y = np.array([[[-0.06, 0.0, 0.1], [0.05, -0.05, 0.05]]])
line = mlines.Line2D(x + grid[8, 0], y + grid[8, 1], lw=5., alpha=0.3)
label(grid[8], "Line2D")

colors = np.linspace(0, 1, len(patches))
collection = PatchCollection(patches, cmap=plt.cm.hsv, alpha=0.3)
collection.set_array(np.array(colors))
ax.add_collection(collection)
ax.add_line(line)

plt.subplots_adjust(left=0, right=1, bottom=0, top=1)
plt.axis('equal')
plt.axis('off')
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Demo of a PathPatch object.

```python
import matplotlib.path as mpath
import matplotlib.patches as mpatches
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
Path = mpath.Path
path_data = [
    (Path.MOVETO, (1.58, -2.57)),
    (Path.CURVE4, (0.35, -1.1)),
    (Path.CURVE4, (-1.75, 2.0)),
    (Path.CURVE4, (0.375, 2.0)),
    (Path.LINETO, (0.85, 1.15)),
    (Path.CURVE4, (2.2, 3.2)),
    (Path.CURVE4, (3, 0.05)),
    (Path.CURVE4, (2.0, -0.5)),
    (Path.CLOSEPOLY, (1.58, -2.57)),
]
```

```
codes, verts = zip(*path_data)
path = mpath.Path(verts, codes)
patch = mpatches.PathPatch(path, facecolor='r', alpha=0.5)
ax.add_patch(patch)

# plot control points and connecting lines
x, y = zip(*path.vertices)
line, = ax.plot(x, y, 'go-')

ax.grid()
ax.axis('equal')
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

### 102.3 shapes_and_collections example code: scatter_demo.py

```python
import numpy as np
import matplotlib.pyplot as plt

# Simple demo of a scatter plot.

import numpy as np
import matplotlib.pyplot as plt
```
N = 50
x = np.random.rand(N)
y = np.random.rand(N)
colors = np.random.rand(N)
area = np.pi * (15 * np.random.rand(N))**2  # 0 to 15 point radii

plt.scatter(x, y, s=area, c=colors, alpha=0.5)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
103.1 showcase example code: anatomy.py

Anatomy of a figure

Blue signal
Red signal

Figure
Axes
Line (line plot)
Markers (scatter plot)
Spines
Legend
Grid
Title
Y axis label
X axis label
Minor tick label
Major tick
Minor tick
# This figure shows the name of several matplotlib elements composing a figure

```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.ticker import AutoMinorLocator, MultipleLocator, FuncFormatter

np.random.seed(19680801)
X = np.linspace(0.5, 3.5, 100)
Y1 = 3+np.cos(X)
Y2 = 1+np.cos(1+X/0.75)/2
Y3 = np.random.uniform(Y1, Y2, len(X))

fig = plt.figure(figsize=(8, 8))
ax = fig.add_subplot(1, 1, 1, aspect=1)

def minor_tick(x, pos):
    if not x % 1.0:
        return ""
    return "%.2f" % x

ax.xaxis.set_major_locator(MultipleLocator(1.000))
ax.xaxis.set_minor_locator(AutoMinorLocator(4))
ax.yaxis.set_major_locator(MultipleLocator(1.000))
ax.yaxis.set_minor_locator(AutoMinorLocator(4))
ax.xaxis.set_minor_formatter(FuncFormatter(minor_tick))
ax.set_xlim(0, 4)
ax.set_ylim(0, 4)
ax.tick_params(which='major', width=1.0)
ax.tick_params(which='major', length=10)
ax.tick_params(which='minor', width=1.0, labelsize=10)
ax.tick_params(which='minor', length=5, labelsize=10, labelcolor='0.25')

ax.grid(linestyle="--", linewidth=0.5, color='0.25', zorder=-10)
ax.plot(X, Y1, c=(0.25, 0.25, 1.00), lw=2, label="Blue signal", zorder=10)
ax.plot(X, Y2, c=(1.00, 0.25, 0.25), lw=2, label="Red signal")
ax.plot(X, Y3, linewidth=0,
        marker='o', markerfacecolor='w', markeredgecolor='k')

ax.set_title("Anatomy of a figure", fontsize=20, verticalalignment='bottom')
ax.set_xlabel("X axis label")
ax.set_ylabel("Y axis label")

ax.legend()

def circle(x, y, radius=0.15):
    from matplotlib.patches import Circle
```
from matplotlib.path effects import withStroke

circle = Circle(xy, radius, clip_on=False, zorder=10, linewidth=1,
edgecolor='black', facecolor=(0, 0, 0, 0.0125),
path_effects=[withStroke(linewidth=5, foreground='w')])

ax.add_artist(circle)

def text(x, y, text):
    ax.text(x, y, text, backgroundcolor="white",
    ha='center', va='top', weight='bold', color='blue')

# Minor tick
circle(0.50, -0.10)
text(0.50, -0.32, "Minor tick label")

# Major tick
circle(-0.03, 4.00)
text(0.03, 3.80, "Major tick")

# Minor tick
circle(0.00, 3.50)
text(0.00, 3.30, "Minor tick")

# Major tick label
circle(-0.15, 3.00)
text(-0.15, 2.80, "Major tick label")

# X Label
circle(1.80, -0.27)
text(1.80, -0.45, "X axis label")

# Y Label
circle(-0.27, 1.80)
text(-0.27, 1.6, "Y axis label")

# Title
circle(1.60, 4.13)
text(1.60, 3.93, "Title")

# Blue plot
circle(1.75, 2.80)
text(1.75, 2.60, "Line\n(line plot)"

# Red plot
circle(1.20, 0.60)
text(1.20, 0.40, "Line\n(line plot)"

# Scatter plot
circle(3.20, 1.75)
text(3.20, 1.55, "Markers\n(scatter plot)"

# Grid
circle(3.00, 3.00)
text(3.00, 2.80, "Grid")

# Legend
circle(3.70, 3.80)
text(3.70, 3.60, "Legend")

# Axes
circle(0.5, 0.5)
text(0.5, 0.3, "Axes")

# Figure
circle(-0.3, 0.65)
text(-0.3, 0.45, "Figure")

color = 'blue'
ax.annotate('Spines', xy=(4.0, 0.35), xycoords='data',
            xytext=(3.3, 0.5), textcoords='data',
            weight='bold', color=color,
            arrowprops=dict(arrowstyle='->',
                             connectionstyle="arc3",
                             color=color))

ax.annotate('', xy=(3.15, 0.0), xycoords='data',
            xytext=(3.45, 0.45), textcoords='data',
            weight='bold', color=color,
            arrowprops=dict(arrowstyle='->',
                             connectionstyle="arc3",
                             color=color))

ax.text(4.0, -0.4, "Made with http://matplotlib.org",
         fontsize=10, ha="right", color='0.5')

plt.show()
103.2 showcase example code: bachelors_degrees_by_gender.py

Percentage of Bachelor's degrees conferred to women in the U.S.A. by major (1970-2011)

```python
import matplotlib.pyplot as plt
from matplotlib.mlab import csv2rec
from matplotlib.cbook import get_sample_data
```
fname = get_sample_data('percent_bachelors_degrees_women_usa.csv')
gender_degree_data = csv2rec(fname)

# These are the colors that will be used in the plot
color_sequence = ['#1f77b4', '#aec7e8', '#ff7f0e', '#ffbb78', '#2ca02c',
                 '#98df8a', '#d62728', '#ff9896', '#9467bd', '#c5b0d5',
                 '#8c564b', '#c49c94', '#e377c2', '#f7b6d2', '#7f7f7f',
                 '#bcbd22', '#dbdb8d', '#17becf', '#9edae5']

# You typically want your plot to be ~1.33x wider than tall. This plot
# is a rare exception because of the number of lines being plotted on it.
# Common sizes: (10, 7.5) and (12, 9)
fig, ax = plt.subplots(1, 1, figsize=(12, 14))

# Remove the plot frame lines. They are unnecessary here.
ax.spines['top'].set_visible(False)
ax.spines['bottom'].set_visible(False)
ax.spines['right'].set_visible(False)
ax.spines['left'].set_visible(False)

# Ensure that the axis ticks only show up on the bottom and left of the plot.
# Ticks on the right and top of the plot are generally unnecessary.
ax.get_xaxis().tick_bottom()
ax.get_yaxis().tick_left()

fig.subplots_adjust(left=.06, right=.75, bottom=.02, top=.94)

# Limit the range of the plot to only where the data is.
# Avoid unnecessary whitespace.
ax.set_xlim(1969.5, 2011.1)
ax.set_ylim(-0.25, 90)

# Make sure your axis ticks are large enough to be easily read.
# You don't want your viewers squinting to read your plot.
plt.xticks(range(1970, 2011, 10), fontsize=14)
plt.yticks(range(0, 91, 10), fontsize=14)
ax.xaxis.set_major_formatter(plt.FuncFormatter('{:.0f}'.format))
ax.yaxis.set_major_formatter(plt.FuncFormatter('{:.0f}%'.format))

# Provide tick lines across the plot to help your viewers trace along
# the axis ticks. Make sure that the lines are light and small so they
# don't obscure the primary data lines.
plt.grid(True, 'major', 'y', ls='--', lw=.5, c='k', alpha=.3)

# Remove the tick marks; they are unnecessary with the tick lines we just
# plotted.
plt.tick_params(axis='both', which='both', bottom='off', top='off',
                labelbottom='on', left='off', right='off', labelleft='on')

# Now that the plot is prepared, it's time to actually plot the data!
# Note that I plotted the majors in order of the highest % in the final year.
majors = ['Health Professions', 'Public Administration', 'Education',
          'Psychology', 'Foreign Languages', 'English',
          'Communications\hand Journalism', 'Art and Performance', 'Biology',
          'Communications\and Journalis...
'Agriculture', 'Social Sciences and History', 'Business', 'Math and Statistics', 'Architecture', 'Physical Sciences', 'Computer Science', 'Engineering']

y_offsets = {'Foreign Languages': 0.5, 'English': -0.5, 'Communications\nand Journalism': 0.75, 'Art and Performance': -0.25, 'Agriculture': 1.25, 'Social Sciences and History': 0.25, 'Business': -0.75, 'Math and Statistics': 0.75, 'Architecture': -0.75, 'Computer Science': 0.75, 'Engineering': -0.25}

for rank, column in enumerate(majors):
    # Plot each line separately with its own color.
    column_rec_name = column.replace(' ', '_').lower()
    line = plt.plot(gender_degree_data.year, gender_degree_data[column_rec_name], lw=2.5, color=color_sequence[rank])

    # Add a text label to the right end of every line. Most of the code below is adding specific offsets y position because some labels overlapped.
    y_pos = gender_degree_data[column_rec_name][-1] - 0.5
    if column in y_offsets:
        y_pos += y_offsets[column]

    # Again, make sure that all labels are large enough to be easily read by the viewer.
    plt.text(2011.5, y_pos, column, fontsize=14, color=color_sequence[rank])

# Make the title big enough so it spans the entire plot, but don't make it so big that it requires two lines to show.

# Note that if the title is descriptive enough, it is unnecessary to include axis labels; they are self-evident, in this plot's case.
fig.suptitle('Percentage of Bachelor\'s degrees conferred to women in \nthe U.S.A. by major (1970-2011)\n', fontsize=18, ha='center')

# Finally, save the figure as a PNG.
# You can also save it as a PDF, JPEG, etc.
# Just change the file extension in this call.
plt.savefig('percent-bachelors-degrees-women-usa.png', bbox_inches='tight')
plt.show()
This example shows how to create the Firefox logo with path and patches.

```python
import re
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.path import Path
import matplotlib.patches as patches

# From: http://raphaeljs.com/icons/#firefox
firefox = "M28.4,22.469c0.479-0.964,0.851-1.991,1.095-3.066c0.953-3.661,0.666-6.854,0.666-6.854\n\n→0c-0.327,2.104c0,0-0.469-3.896-1.044-5.353c-0.881-2.231-1.273-2.214-1.274-2.214c0.542,1.379,0.494,2.169,0.483,2.288c-0.01-0.016-0.019-0.032-0.027-0.047c-0.131-0.3324-0.797-1.819-2.225-2.878c-2.502-2.481-5.943-4.014-9.745-4.015c-4.056,0-7.705,1.745-10.238,4.525\n\n→C5.444,6.5,5.183,5.938,5.159,5.317c0,0-0.002,0.002-0.006,0.005c0-0.011-0.003-0.031c0,0-1.61,1.247-1.436,4.612c-0.299,0.574-0.56,1.172-0.777,1.791c-0.375,0.617-0.75,2.064-1.059,2.746c0,0.133-0.422,0.399-0.988c-0.064,0.482-0.103,0.547-0.547,0.893c-0.547,0.893-0.547,0.893-0.393,3.088c0,0,0.032-0.406,0.136-1.021c0.857,\n\n→6.854,6.667,12.165,13.743,12.16510,0c1.86,0,3.636-0.37,5.256-1.036C24.938,27.771,27.\n\n→116,25.196,28.4,22.469zM16.002,3.356c2.446,0,4.73,0.68,6.68,1.86c-2.274-0.528-3.433-0.\n\n→261-3.423-0.248c0.013,0.015,3.384,0.589,3.981,1.411c0,0-1.431,0-2.856,0.41c-0.065,0."

plt.figure()
plt.title("Firefox")
plt.axis('off')
plt.show()
```

103.3 showcase example code: firefox.py
```python
def svg_parse(path):
    commands = {
        'M': (Path.MOVETO,),
        'L': (Path.LINETO,),
        'Q': (Path.CURVE3,)*2,
        'C': (Path.CURVE4,)*3,
        'Z': (Path.CLOSEPOLY,)}
    path_re = re.compile(r'([MLHVCSQTAZ])([^MLHVCSQTAZ]+)', re.IGNORECASE)
    float_re = re.compile(r'(?<=[^;\s]+)(?<![^;\d]+)(?<!\.)\d+(?:\d+)?')
    vertices = []
    codes = []
    last = (0, 0)
    for cmd, values in path_re.findall(path):
        points = [float(v) for v in float_re.findall(values)]
        points = np.array(points).reshape((len(points)//2, 2))
        if cmd.islower():
            points += last
            cmd = cmd.capitalize()
            last = points[-1]
        codes.extend(commands[cmd])
        vertices.extend(points.tolist())
    return codes, vertices

# SVG to matplotlib
codes, verts = svg_parse(firefox)
verts = np.array(verts)
path = Path(verts, codes)

# Make upside down
verts[:, 1] *= -1
xmin, xmax = verts[:, 0].min()-1, verts[:, 0].max()+1
ymin, ymax = verts[:, 1].min()-1, verts[:, 1].max()+1

fig = plt.figure(figsize=(5, 5))
ax = fig.add_axes([0.0, 0.0, 1.0, 1.0], frameon=False, aspect=1)

# White outline (width = 6)
patch = patches.PathPatch(path, facecolor='None', edgecolor='w', lw=6)
ax.add_patch(patch)

# Actual shape with black outline
patch = patches.PathPatch(path, facecolor='orange', edgecolor='k', lw=2)
ax.add_patch(patch)

# Centering
ax.set_xlim(xmin, xmax)
ax.set_ylim(ymin, ymax)

# No ticks
ax.set_xticks([])
ax.set_yticks([])
```

2808 Chapter 103. showcase Examples
# Display
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

## 103.4 showcase example code: integral_demo.py

```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.patches import Polygon
```

Plot demonstrating the integral as the area under a curve.

Although this is a simple example, it demonstrates some important tweaks:

* A simple line plot with custom color and line width.
* A shaded region created using a Polygon patch.
* A text label with mathtext rendering.
* `figtext` calls to label the x- and y-axes.
* Use of axis spines to hide the top and right spines.
* Custom tick placement and labels.

```python
...""
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.patches import Polygon
```

103.4. showcase example code: integral_demo.py 2809
```python
def func(x):
    return (x - 3) * (x - 5) * (x - 7) + 85

a, b = 2, 9  # integral limits
x = np.linspace(0, 10)
y = func(x)

fig, ax = plt.subplots()
plt.plot(x, y, 'r', linewidth=2)
plt.ylim(ymin=0)

# Make the shaded region
ix = np.linspace(a, b)
iy = func(ix)
verts = [(a, 0)] + list(zip(ix, iy)) + [(b, 0)]
poly = Polygon(verts, facecolor='0.9', edgecolor='0.5')
ax.add_patch(poly)

plt.text(0.5 * (a + b), 30, r'$\int_{a}^{b} f(x)\,\mathrm{d}x$',
         horizontalalignment='center', fontsize=20)
plt.figtext(0.9, 0.05, '$x$')
plt.figtext(0.1, 0.9, '$y$')
ax.spines['right'].set_visible(False)
ax.spines['top'].set_visible(False)
ax.xaxis.set_ticks_position('bottom')
ax.set_xticks((a, b))
a.set_xticklabels(('a', 'b'))
ax.set_yticks([])
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Shaded & power normalized rendering

The Mandelbrot set rendering can be improved by using a normalized recount associated with a power normalized colormap (gamma=0.3). Rendering can be further enhanced thanks to shading.

The `maxiter` gives the precision of the computation. `maxiter=200` should take a few seconds on most modern laptops.

```python
import numpy as np

def mandelbrot_set(xmin, xmax, ymin, ymax, xn, yn, maxiter, horizon=2.0):
    X = np.linspace(xmin, xmax, xn, dtype=np.float32)
```
Y = np.linspace(ymin, ymax, yn, dtype=np.float32)
C = X + Y[:, None]*1j
N = np.zeros(C.shape, dtype=int)
Z = np.zeros(C.shape, np.complex64)
for n in range(maxiter):
    I = np.less(abs(Z), horizon)
    N[I] = n
    Z[I] = Z[I]**2 + C[I]
N[N == maxiter-1] = 0
return Z, N

if __name__ == '__main__':
    import time
    import matplotlib
    from matplotlib import colors
    import matplotlib.pyplot as plt
    xmin, xmax, xn = -2.25, +0.75, 3000/2
    ymin, ymax, yn = -1.25, +1.25, 2500/2
    maxiter = 200
    horizon = 2.0 ** 40
    log_horizon = np.log(np.log(horizon))/np.log(2)
    Z, N = mandelbrot_set(xmin, xmax, ymin, ymax, xn, yn, maxiter, horizon)
    # Normalized recount as explained in:
    # https://linas.org/art-gallery/escape/smooth.html
    # This line will generate warnings for null values but it is faster to
    # process them afterwards using the nan_to_num
    with np.errstate(invalid='ignore'):
        M = np.nan_to_num(N + 1 -
                        np.log(np.log(abs(Z)))/np.log(2) +
                        log_horizon)
    dpi = 72
    width = 10
    height = 10*yn/xn
    fig = plt.figure(figsize=(width, height), dpi=dpi)
    ax = fig.add_axes([0.0, 0.0, 1.0, 1.0], frameon=False, aspect=1)
    # Shaded rendering
    light = colors.LightSource(azdeg=315, altdeg=10)
    M = light.shade(M, cmap=plt.cm.hot, vert_exag=1.5,
                    norm=colors.PowerNorm(0.3), blend_mode='hsv')
    plt.imshow(M, extent=[xmin, xmax, ymin, ymax], interpolation="bicubic")
    ax.set_xticks([])
    ax.set_yticks([])
    # Some advertisement for matplotlib
    year = time.strftime('%Y')
    major, minor, micro = matplotlib.__version__.split('.', 2)
Keywords: python, matplotlib, pylab, example, codex (see Search examples)

103.6 showcase example code: xkcd.py

"STOVE OWNERSHIP" FROM XKCD BY RANDALL MONROE
Shows how to create an xkcd-like plot.

```python
import matplotlib.pyplot as plt
import numpy as np

with plt.xkcd():
    fig = plt.figure()
    ax = fig.add_axes([0.1, 0.2, 0.8, 0.7])
    ax.spines['right'].set_color('none')
    ax.spines['top'].set_color('none')
    plt.xticks([])
    plt.yticks([])
    ax.set_ylim([-30, 10])

    data = np.ones(100)
    data[70:] -= np.arange(30)
    plt.annotate("",
```

Matplotlib, Release 2.0.2

plt.plot(data)

plt.xlabel('time')
plt.ylabel('my overall health')

fig.text(0.5, 0.05, """Stove Ownership" from xkcd by Randall Monroe', ha='center')

# Based on "The Data So Far" from XKCD by Randall Monroe
# http://xkcd.com/373/

fig = plt.figure()
ax = fig.add_axes((0.1, 0.2, 0.8, 0.7))
ax.bar([0, 1], [0, 100], 0.25)
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
ax.set_xticks([0, 1])
ax.set_xlim([-0.5, 1.5])
ax.set_ylim([0, 110])
ax.set_xticklabels(['CONFIRMED BY\nEXPERIMENT', 'REFUTED BY\nEXPERIMENT'])
plt.yticks([])
plt.title("CLAIMS OF SUPERNATURAL POWERS")

fig.text(0.5, 0.05, """The Data So Far" from xkcd by Randall Monroe', ha='center')

plt.show()
104.1 specialty_plots example code: advanced_hillshading.py

Using a colorbar with a shaded plot
Avoiding Outliers in Shaded Plots

```
Demonstrates a few common tricks with shaded plots.
```

Shade by one variable, color by another
```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import LightSource, Normalize

def display_colorbar():
    """Display a correct numeric colorbar for a shaded plot."""
    y, x = np.mgrid[-4:2:200j, -4:2:200j]
    z = 10 * np.cos(x**2 + y**2)

    cmap = plt.cm.copper
    ls = LightSource(315, 45)
    rgb = ls.shade(z, cmap)

    fig, ax = plt.subplots()
    ax.imshow(rgb, interpolation='bilinear')

    # Use a proxy artist for the colorbar...
    im = ax.imshow(z, cmap=cmap)
    im.remove()
    fig.colorbar(im)

    ax.set_title('Using a colorbar with a shaded plot', size='x-large')

def avoid_outliers():
    """Use a custom norm to control the displayed z-range of a shaded plot."""
    y, x = np.mgrid[-4:2:200j, -4:2:200j]
    z = 10 * np.cos(x**2 + y**2)

    # Add some outliers...
    z[100, 105] = 2000
    z[120, 110] = -9000

    ls = LightSource(315, 45)
    fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=(8, 4.5))

    rgb = ls.shade(z, plt.cm.copper)
    ax1.imshow(rgb, interpolation='bilinear')
    ax1.set_title('Full range of data')

    rgb = ls.shade(z, plt.cm.copper, vmin=-10, vmax=10)
    ax2.imshow(rgb, interpolation='bilinear')
    ax2.set_title('Manually set range')

    fig.suptitle('Avoiding Outliers in Shaded Plots', size='x-large')

def shade_other_data():
    """Demonstrates displaying different variables through shade and color."""
    y, x = np.mgrid[-4:2:200j, -4:2:200j]
    z1 = np.sin(x**2)  # Data to hillshade
    z2 = np.cos(x**2 + y**2)  # Data to color
```

104.1. specialty_plots example code: advanced_hillshading.py
"""
Demo of a function to create Hinton diagrams.
"""
Hinton diagrams are useful for visualizing the values of a 2D array (e.g. a weight matrix): Positive and negative values are represented by white and black squares, respectively, and the size of each square represents the magnitude of each value.

Initial idea from David Warde-Farley on the SciPy Cookbook

```python
import numpy as np
import matplotlib.pyplot as plt

def hinton(matrix, max_weight=None, ax=None):
    """Draw Hinton diagram for visualizing a weight matrix.""
    ax = ax if ax is not None else plt.gca()

    if not max_weight:
        max_weight = 2 ** np.ceil(np.log(np.abs(matrix).max())) / np.log(2)

    ax.patch.set_facecolor('gray')
    ax.set_aspect('equal', 'box')
    ax.xaxis.set_major_locator(plt.NullLocator())
    ax.yaxis.set_major_locator(plt.NullLocator())

    for (x, y), w in np.ndenumerate(matrix):
        color = 'white' if w > 0 else 'black'
        size = np.sqrt(np.abs(w) / max_weight)
        rect = plt.Rectangle([x - size / 2, y - size / 2], size, size,
                              facecolor=color, edgecolor=color)
        ax.add_patch(rect)

    ax.autoscale_view()
    ax.invert_yaxis()

if __name__ == '__main__':
    hinton(np.random.rand(20, 20) - 0.5)
    plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Demonstrates the visual effect of varying blend mode and vertical exaggeration on "hillshaded" plots.

Note that the "overlay" and "soft" blend modes work well for complex surfaces such as this example, while the default "hsv" blend mode works best for smooth
surfaces such as many mathematical functions.

In most cases, hillshading is used purely for visual purposes, and *dx*/*dy* can be safely ignored. In that case, you can tweak *vert_exag* (vertical exaggeration) by trial and error to give the desired visual effect. However, this example demonstrates how to use the *dx* and *dy* kwargs to ensure that the *vert_exag* parameter is the true vertical exaggeration.

```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.cbook import get_sample_data
from matplotlib.colors import LightSource
dem = np.load(get_sample_data('jacksboro_fault_dem.npz'))
z = dem['elevation']

#-- Optional dx and dy for accurate vertical exaggeration -------------------
# If you need topographically accurate vertical exaggeration, or you don't want
# to guess at what *vert_exag* should be, you'll need to specify the cellsize
# of the grid (i.e. the *dx* and *dy* parameters). Otherwise, any *vert_exag*
# value you specify will be relative to the grid spacing of your input data
# (in other words, *dx* and *dy* default to 1.0, and *vert_exag* is calculated
# relative to those parameters). Similarly, *dx* and *dy* are assumed to be in
# the same units as your input z-values. Therefore, we'll need to convert the
# given dx and dy from decimal degrees to meters.
dx, dy = dem['dx'], dem['dy']
dy = 111200 * dy
dx = 111200 * dx * np.cos(np.radians(dem['ymin']))

# Shade from the northwest, with the sun 45 degrees from horizontal
ls = LightSource(azdeg=315, altdeg=45)
cmap = plt.cm.gist_earth

fig, axes = plt.subplots(nrows=4, ncols=3, figsize=(8, 9))
plt.setp(axes.flat, xticks=[], yticks=[])  # Vary vertical exaggeration and blend mode and plot all combinations
for col, ve in zip(axes.T, [0.1, 1, 10]):
    col[0].imshow(ls.hillshade(z, vert_exag=ve, dx=dx, dy=dy), cmap='gray')
    # Place hillshaded plots with different blend modes in the rest of the rows
    for ax, mode in zip(col[1:], ['hsv', 'overlay', 'soft']):
        rgb = ls.shade(z, cmap=cmap, blend_mode=mode, vert_exag=ve, dx=dx, dy=dy)
        ax.imshow(rgb)

# Label rows and columns
for ax, ve in zip(axes[0], [0.1, 1, 10]):
    ax.set_title('{' + str(ve) + '}', size=18)
for ax, mode in zip(axes[:, 0], ['Hillshade', 'hsv', 'overlay', 'soft']):
    ax.set_ylabel(mode, size=18)
```

104.3. specialty_plots example code: topographic_hillshading.py
# Group labels...
axes[0, 1].annotate('Vertical Exaggeration', (0.5, 1), xytext=(0, 30),
        textcoords='offset points', xycoords='axes fraction',
        ha='center', va='bottom', size=20)
axes[2, 0].annotate('Blend Mode', (0, 0.5), xytext=(-30, 0),
        textcoords='offset points', xycoords='axes fraction',
        ha='right', va='center', size=20, rotation=90)
fig.subplots_adjust(bottom=0.05, right=0.95)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
CHAPTER

FIVE

STATISTICS EXAMPLES

105.1 statistics example code: boxplot_color_demo.py

```python
import matplotlib.pyplot as plt
import numpy as np
```

This plot illustrates how to create two types of box plots (rectangular and notched), and how to fill them with custom colors by accessing the properties of the artists of the box plots. Additionally, the `labels` parameter is used to provide x-tick labels for each sample.

A good general reference on boxplots and their history can be found here: http://vita.had.co.nz/papers/boxplots.pdf
# Random test data
np.random.seed(123)
all_data = [np.random.normal(0, std, 100) for std in range(1, 4)]

fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(9, 4))

# rectangular box plot
bplot1 = axes[0].boxplot(all_data,
                         vert=True,  # vertical box alignment
                         patch_artist=True)  # fill with color

# notch shape box plot
bplot2 = axes[1].boxplot(all_data,
                         notch=True,  # notch shape
                         vert=True,  # vertical box alignment
                         patch_artist=True)  # fill with color

# fill with colors
colors = ['pink', 'lightblue', 'lightgreen']
for bplot in (bplot1, bplot2):
    for patch, color in zip(bplot['boxes'], colors):
        patch.set_facecolor(color)

# adding horizontal grid lines
for ax in axes:
    ax.yaxis.grid(True)
    ax.set_xticks([y+1 for y in range(len(all_data))], )
    ax.set_xlabel('xlabel')
    ax.set_ylabel('ylabel')

# add x-tick labels
plt.setp(axes, xticks=[y+1 for y in range(len(all_data))],
         xticklabels=['x1', 'x2', 'x3', 'x4'])

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
105.2 statistics example code: boxplot_demo.py

Default

showmeans=True

meanline=True

Tufte Style
(showbox=False,
showcaps=False)

notch=True,
bootstrap=10000

showfliers=False
I never said they'd be pretty

Custom boxprops

Custom medianprops and flierprops

whis="range"

Custom mean as point

Custom mean as line

whis=[15, 85] #percentiles

""

========================================
Demo of artist customization in box plots
========================================

This example demonstrates how to use the various kwargs to fully customize box plots. The first figure demonstrates how to remove and add individual components (note that the mean is the only value not shown by default). The second figure demonstrates how the styles of the artists can be customized. It also demonstrates how to set the limit of the whiskers to specific percentiles (lower right axes)

A good general reference on boxplots and their history can be found here: http://vita.had.co.nz/papers/boxplots.pdf
```python
import numpy as np
import matplotlib.pyplot as plt

# fake data
np.random.seed(937)
data = np.random.lognormal(size=(37, 4), mean=1.5, sigma=1.75)
labels = list('ABCD')
fs = 10  # fontsize

# demonstrate how to toggle the display of different elements:
fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(6, 6), sharey=True)
axes[0, 0].boxplot(data, labels=labels)
axes[0, 0].set_title('Default', fontsize=fs)

axes[0, 1].boxplot(data, labels=labels, showmeans=True)
axes[0, 1].set_title('showmeans=True', fontsize=fs)

axes[0, 2].boxplot(data, labels=labels, showmeans=True, meanline=True)
axes[0, 2].set_title('showmeans=True, meanline=True', fontsize=fs)

axes[1, 0].boxplot(data, labels=labels, showbox=False, showcaps=False)
tufte_title = 'Tufte Style (showbox=False, showcaps=False)
axes[1, 0].set_title(tufte_title, fontsize=fs)

axes[1, 1].boxplot(data, labels=labels, notch=True, bootstrap=10000)
axes[1, 1].set_title('notch=True, bootstrap=10000', fontsize=fs)

axes[1, 2].boxplot(data, labels=labels, showfliers=False)
axes[1, 2].set_title('showfliers=False', fontsize=fs)

for ax in axes.flatten():
    ax.set_yscale('log')
    ax.set_yticklabels([])

fig.subplots_adjust(hspace=0.4)
plt.show()

# demonstrate how to customize the display different elements:
boxprops = dict(linestyle='--', linewidth=3, color='darkgoldenrod')
flierprops = dict(marker='o', markerfacecolor='green', markersize=12,
    linestyle='none')
medianprops = dict(linestyle='-.', linewidth=2.5, color='firebrick')
meanpointprops = dict(marker='D', markeredgewidth=12, markerfacecolor='black',
    markeredgecolor='firebrick')
meanlineprops = dict(linestyle='--', linewidth=2.5, color='purple')

fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(6, 6), sharey=True)
axes[0, 0].boxplot(data, boxprops=boxprops)
axes[0, 0].set_title('Custom boxprops', fontsize=fs)
```

105.2. statistics example code: boxplot_demo.py 2829
axes[0, 1].boxplot(data, flierprops=flierprops, medianprops=medianprops)
axes[0, 1].set_title('Custom medianprops\nand flierprops', fontsize=fs)

axes[0, 2].boxplot(data, whis='range')
axes[0, 2].set_title('whis="range"', fontsize=fs)

axes[1, 0].boxplot(data, meanprops=meanpointprops, meanline=False, showmeans=True)
axes[1, 0].set_title('Custom mean\nas point', fontsize=fs)

axes[1, 1].boxplot(data, meanprops=meanlineprops, meanline=True, showmeans=True)
axes[1, 1].set_title('Custom mean\nline', fontsize=fs)

axes[1, 2].boxplot(data, whis=[15, 85])
axes[1, 2].set_title('whis=[15, 85]\n#percentiles', fontsize=fs)

for ax in axes.flatten():
    ax.set_yscale('log')
    ax.set_yticklabels([])

fig.suptitle("I never said they'd be pretty")
fig.subplots_adjust(hspace=0.4)
plt.show()
Note that although violin plots are closely related to Tukey's (1977) box plots, they add useful information such as the distribution of the sample data (density trace).

By default, box plots show data points outside 1.5 * the inter-quartile range as outliers above or below the whiskers whereas violin plots show the whole range of the data.

A good general reference on boxplots and their history can be found here: http://vita.had.co.nz/papers/boxplots.pdf

Violin plots require matplotlib >= 1.4.

For more information on violin plots, the scikit-learn docs have a great section: http://scikit-learn.org/stable/modules/density.html

```python
import matplotlib.pyplot as plt
import numpy as np

fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(9, 4))

# generate some random test data
all_data = [np.random.normal(0, std, 100) for std in range(6, 10)]

# plot violin plot
axes[0].violinplot(all_data,
                   showmeans=False,
                   showmedians=True)
axes[0].set_title('violin plot')

# plot box plot
axes[1].boxplot(all_data)
axes[1].set_title('box plot')

# adding horizontal grid lines
for ax in axes:
    ax.yaxis.grid(True)
    ax.set_xticks([y+1 for y in range(len(all_data))])
    ax.set_xlabel('xlabel')
    ax.set_ylabel('ylabel')

# add x-tick labels
plt.setp(axes, xticks=[y+1 for y in range(len(all_data))],
         xticklabels=['x1', 'x2', 'x3', 'x4'])
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
105.4 statistics example code: bxp_demo.py

Default

showmeans=True

showmeans=True,
meanline=True

Tufte Style
(showbox=False,
showcaps=False)

notch=True

showfliers=False
Demo of the boxplot drawer function

This example demonstrates how to pass pre-computed box plot statistics to the box plot drawer. The first figure demonstrates how to remove and add individual components (note that the mean is the only value not shown by default). The second figure demonstrates how the styles of the artists can be customized.

A good general reference on boxplots and their history can be found here: http://vita.had.co.nz/papers/boxplots.pdf
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cbook as cbook

# fake data
np.random.seed(937)
data = np.random.lognormal(size=(37, 4), mean=1.5, sigma=1.75)
labels = list('ABCD')

# compute the boxplot stats
stats = cbook.boxplot_stats(data, labels=labels, bootstrap=10000)
# After we've computed the stats, we can go through and change anything.
# Just to prove it, I'll set the median of each set to the median of all
# the data, and double the means
for n in range(len(stats)):
    stats[n]['med'] = np.median(data)
    stats[n]['mean'] *= 2

print(stats[0].keys())

fs = 10  # fontsize

# demonstrate how to toggle the display of different elements:
fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(6, 6), sharey=True)
axes[0, 0].bxp(stats)
axes[0, 0].set_title('Default', fontsize=fs)
axes[0, 1].bxp(stats, showmeans=True)
axes[0, 1].set_title('showmeans=True', fontsize=fs)
axes[0, 2].bxp(stats, showmeans=True, meanline=True)
axes[0, 2].set_title('showmeans=True, meanline=True', fontsize=fs)

axes[1, 0].bxp(stats, showbox=False, showcaps=False)
tufte_title = 'Tufte Style
(showbox=False, showcaps=False)'
axes[1, 0].set_title(tufte_title, fontsize=fs)

axes[1, 1].bxp(stats, shownotches=True)
axes[1, 1].set_title('notch=True', fontsize=fs)

axes[1, 2].bxp(stats, showfliers=False)
axes[1, 2].set_title('showfliers=False', fontsize=fs)

for ax in axes.flatten():
    ax.set_yscale('log')
    ax.set_yticklabels([])
fig.subplots_adjust(hspace=0.4)
plt.show()
linestyle='none')
medianprops = dict(linestyle='-', linewidth=2.5, color='firebrick')
meanpointprops = dict(marker='D', markeredgecolor='black',
                      markerfacecolor='firebrick')
meanlineprops = dict(linestyle='--', linewidth=2.5, color='purple')

fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(6, 6), sharey=True)
axes[0, 0].bxp(stats, boxprops=boxprops)
axes[0, 0].set_title('Custom boxprops', fontsize=fs)
axes[0, 1].bxp(stats, flierprops=flierprops, medianprops=medianprops)
axes[0, 1].set_title('Custom medianprops
and flierprops', fontsize=fs)
axes[1, 0].bxp(stats, meanprops=meanpointprops, meanline=False,
               showmeans=True)
axes[1, 0].set_title('Custom mean
as point', fontsize=fs)
axes[1, 1].bxp(stats, meanprops=meanlineprops, meanline=True,
               showmeans=True)
axes[1, 1].set_title('Custom mean
as line', fontsize=fs)
for ax in axes.flatten():
    ax.set_yscale('log')
    ax.set_yticklabels([])
fig.suptitle("I never said they'd be pretty")
fig.subplots_adjust(hspace=0.4)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

105.5 statistics example code: customized_violin_demo.py
Demo of violin plot customization

This example demonstrates how to fully customize violin plots. The first plot shows the default style by providing only the data. The second plot first limits what matplotlib draws with additional kwargs. Then a simplified representation of a box plot is drawn on top. Lastly, the styles of the artists of the violins are modified.

For more information on violin plots, the scikit-learn docs have a great section: http://scikit-learn.org/stable/modules/density.html

```python
import matplotlib.pyplot as plt
import numpy as np

def adjacent_values(vals, q1, q3):
    upper_adjacent_value = q3 + (q3 - q1) * 1.5
    upper_adjacent_value = np.clip(upper_adjacent_value, q3, vals[-1])

    lower_adjacent_value = q1 - (q3 - q1) * 1.5
    lower_adjacent_value = np.clip(lower_adjacent_value, vals[0], q1)
    return lower_adjacent_value, upper_adjacent_value

def set_axis_style(ax, labels):
    ax.get_xaxis().set_tick_params(direction='out')
    ax.xaxis.set_ticks_position('bottom')
    ax.set_xticks(np.arange(1, len(labels) + 1))
    ax.set_xticklabels(labels)
    ax.set_xlim(0.25, len(labels) + 0.75)
    ax.set_xlabel('Sample name')

# create test data
np.random.seed(123)
data = [sorted(np.random.normal(0, std, 100)) for std in range(1, 5)]

fig, (ax1, ax2) = plt.subplots(nrows=1, ncols=2, figsize=(9, 4), sharey=True)

ax1.set_title('Default violin plot')
ax1.set_ylabel('Observed values')
ax1.violinplot(data)

ax1.set_title('Customized violin plot')
parts = ax2.violinplot(  
data, showmeans=False, showmedians=False,  
showextrema=False)
for pc in parts['bodies']:
    pc.set_facecolor('#D43F3A')
    pc.set_edgecolor('black')
    pc.set_alpha(1)

quartile1, medians, quartile3 = np.percentile(data, [25, 50, 75], axis=1)
whiskers = np.array([
    adjacent_values(sorted_array, q1, q3)
    for sorted_array, q1, q3 in zip(data, quartile1, quartile3)])

whiskersMin, whiskersMax = whiskers[:, 0], whiskers[:, 1]

inds = np.arange(1, len(medians) + 1)
ax2.scatter(inds, medians, marker='o', color='white', s=30, zorder=3)
ax2.vlines(inds, quartile1, quartile3, color='k', linestyle='-', lw=5)
ax2.vlines(inds, whiskersMin, whiskersMax, color='k', linestyle='-', lw=1)

# set style for the axes
labels = ['A', 'B', 'C', 'D']
for ax in [ax1, ax2]:
    set_axis_style(ax, labels)

plt.subplots_adjust(bottom=0.15, wspace=0.05)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Demo of the errorbar function

This exhibits the most basic use of the error bar method. In this case, constant values are provided for the error in both the x- and y-directions.

```
import numpy as np
import matplotlib.pyplot as plt

# example data
x = np.arange(0.1, 4, 0.5)
y = np.exp(-x)

fig, ax = plt.subplots()
ax.errorbar(x, y, xerr=0.2, yerr=0.4)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Demo of the different ways of specifying error bars

Errors can be specified as a constant value (as shown in `errorbar_demo.py`). However, this example demonstrates how they vary by specifying arrays of error values.

If the raw `x` and `y` data have length $N$, there are two options:

Array of shape $(N,)$:
- Error varies for each point, but the error values are symmetric (i.e. the lower and upper values are equal).

Array of shape $(2, N)$:
- Error varies for each point, and the lower and upper limits (in that order) are different (asymmetric case)

In addition, this example demonstrates how to use log scale with error bars.
import numpy as np
import matplotlib.pyplot as plt

# example data
x = np.arange(0.1, 4, 0.5)
y = np.exp(-x)

# example error bar values that vary with x-position
error = 0.1 + 0.2 * x

fig, (ax0, ax1) = plt.subplots(nrows=2, sharex=True)
ax0.errorbar(x, y, yerr=error, fmt='-o')
ax0.set_title('variable, symmetric error')

# error bar values w/ different +/- errors that
# also vary with the x-position
lower_error = 0.4 * error
upper_error = error
asymmetric_error = [lower_error, upper_error]

ax1.errorbar(x, y, xerr=asymmetric_error, fmt='o')
ax1.set_title('variable, asymmetric error')
ax1.set_yscale('log')
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Demo of how to include upper and lower limits in error bars

In matplotlib, errors bars can have "limits". Applying limits to the error bars essentially makes the error unidirectional. Because of that, upper and lower limits can be applied in both the y- and x-directions via the `uplims`, `lolims`, `xuplims`, and `xlolims` parameters, respectively. These parameters can be scalar or boolean arrays.

For example, if `xlolims` is `True`, the x-error bars will only extend from the data towards increasing values. If `uplims` is an array filled with `False` except for the 4th and 7th values, all of the y-error bars will be bidirectional, except the 4th and 7th bars, which will extend from the data towards decreasing y-values.

```python
import numpy as np
import matplotlib.pyplot as plt

# example data
x = np.array([0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0])
y = np.exp(-x)
xerr = 0.1
yerr = 0.2
```
# lower & upper limits of the error
lolims = np.array([0, 0, 1, 0, 1, 0, 0, 0, 1, 0], dtype=bool)
uplims = np.array([0, 1, 0, 0, 1, 0, 0, 0, 0, 1], dtype=bool)
ls = 'dotted'

fig, ax = plt.subplots(figsize=(7, 4))

# standard error bars
ax.errorbar(x, y, xerr=xerr, yerr=yerr, linestyle=ls)

# including upper limits
ax.errorbar(x, y + 0.5, xerr=xerr, yerr=yerr, uplims=uplims, linestyle=ls)

# including lower limits
ax.errorbar(x, y + 1.0, xerr=xerr, yerr=yerr, lolims=lolims, linestyle=ls)

# including upper and lower limits
ax.errorbar(x, y + 1.5, xerr=xerr, yerr=yerr, lolims=lolims, uplims=uplims, marker='o', markersize=8, linestyle=ls)

# Plot a series with lower and upper limits in both x & y
# constant x-error with varying y-error
xerr = 0.2
yerr = np.zeros(x.shape) + 0.2
yerr[[3, 6]] = 0.3

# mock up some limits by modifying previous data
xlolims = lolims
xuplims = uplims
lolims = np.zeros(x.shape)
uplims = np.zeros(x.shape)
lolims[[6]] = True  # only limited at this index
uplims[[3]] = True  # only limited at this index

# do the plotting
ax.errorbar(x, y + 2.1, xerr=xerr, yerr=yerr, xlolims=xlolims, xuplims=xuplims, uplims=xuplims, lolims=lolims, marker='o', markersize=8, linestyle='none')

# tidy up the figure
ax.set_xlim((0, 5.5))
ax.set_title('Errorbar upper and lower limits')
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
In this example, we snazz up a pretty standard error bar plot by adding a rectangle patch defined by the limits of the bars in both the x- and y- directions. To do this, we have to write our own custom function called "make_error_boxes". Close inspection of this function will reveal the preferred pattern in writing functions for matplotlib:

1. an "Axes" object is passed directly to the function
2. the function operates on the 'Axes' methods directly, not through the "pyplot" interface
3. plotting kwargs that could be abbreviated are spelled out for better code readability in the future (for example we use "facecolor" instead of "fc")
4. the artists returned by the "Axes" plotting methods are then returned by the function so that, if desired, their styles can be modified later outside of the function (they are not modified in this example).
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.collections import PatchCollection
from matplotlib.patches import Rectangle

# Number of data points
n = 5

# Dummy data
np.random.seed(10)
x = np.arange(0, n, 1)
y = np.random.rand(n) * 5.

# Dummy errors (above and below)
xerr = np.random.rand(2, n) + 0.1
yerr = np.random.rand(2, n) + 0.2

def make_error_boxes(ax, xdata, ydata, xerror, yerror, facecolor='r',
                     edgecolor='None', alpha=0.5):
    # Create list for all the error patches
    errorboxes = []

    # Loop over data points; create box from errors at each point
    for x, y, xe, ye in zip(xdata, ydata, xerror.T, yerror.T):
        rect = Rectangle((x - xe[0], y - ye[0]), xe.sum(), ye.sum())
        errorboxes.append(rect)

    # Create patch collection with specified colour/alpha
    pc = PatchCollection(errorboxes, facecolor=facecolor, alpha=alpha,
                         edgecolor=edgecolor)

    # Add collection to axes
    ax.add_collection(pc)

    # Plot errorbars
    artists = ax.errorbar(xdata, ydata, xerr=xerror, yerr=yerror,
                          fmt='None', ecolor='k')

    return artists

# Create figure and axes
fig, ax = plt.subplots(1)

# Call function to create error boxes
_ = make_error_boxes(ax, x, y, xerr, yerr)

plt.show()
Demo of using histograms to plot a cumulative distribution

This shows how to plot a cumulative, normalized histogram as a step function in order to visualize the empirical cumulative distribution function (CDF) of a sample. We also use the `mlab` module to show the theoretical CDF.

A couple of other options to the `hist` function are demonstrated. Namely, we use the `normed` parameter to normalize the histogram and a couple of different options to the `cumulative` parameter. The `normed` parameter takes a boolean value. When `True`, the bin heights are scaled such that the total area of the histogram is 1. The `cumulative` kwarg is a little more nuanced. Like `normed`, you can pass it `True` or `False`, but you can also pass it `-1` to reverse the distribution.

Since we're showing a normalized and cumulative histogram, these curves are effectively the cumulative distribution functions (CDFs) of the samples. In engineering, empirical CDFs are sometimes called "non-exceedance" curves. In other words, you can look at the $y$-value for a given $x$-value to get the probability of and observation from the sample not exceeding that $x$-value. For example, the value of 225 on the $x$-axis corresponds to about 0.85 on the $y$-axis, so there's an 85% chance that an observation in the sample does not exceed 225. Conversely, setting `cumulative` to `-1` as is done in the last series for this example, creates a "exceedance" curve.
Selecting different bin counts and sizes can significantly affect the shape of a histogram. The Astropy docs have a great section on how to select these parameters:

```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import mlab

np.random.seed(0)
mu = 200
sigma = 25
n_bins = 50
x = np.random.normal(mu, sigma, size=100)

fig, ax = plt.subplots(figsize=(8, 4))

# plot the cumulative histogram
n, bins, patches = ax.hist(x, n_bins, normed=1, histtype='step',
                           cumulative=True, label='Empirical')

# Add a line showing the expected distribution.
y = mlab.normpdf(bins, mu, sigma).cumsum()
y /= y[-1]
ax.plot(bins, y, 'k--', linewidth=1.5, label='Theoretical')

# Overlay a reversed cumulative histogram.
ax.hist(x, bins=bins, normed=1, histtype='step', cumulative=-1,
        label='Reversed emp.')

# tidy up the figure
ax.grid(True)
ax.legend(loc='right')
ax.set_title('Cumulative step histograms')
ax.set_xlabel('Annual rainfall (mm)')
ax.set_ylabel('Likelihood of occurrence')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
**Demo of the histogram (hist) function with a few features**

In addition to the basic histogram, this demo shows a few optional features:

- Setting the number of data bins
- The `normed` flag, which normalizes bin heights so that the integral of the histogram is 1. The resulting histogram is an approximation of the probability density function.
- Setting the face color of the bars
- Setting the opacity (alpha value).

Selecting different bin counts and sizes can significantly affect the shape of a histogram. The Astropy docs have a great section on how to select these parameters:


```python
import numpy as np
```
Matplotlib, Release 2.0.2

import matplotlib.mlab as mlab
import matplotlib.pyplot as plt
np.random.seed(0)
# example data
mu = 100 # mean of distribution
sigma = 15 # standard deviation of distribution
x = mu + sigma * np.random.randn(437)
num_bins = 50
fig, ax = plt.subplots()
# the histogram of the data
n, bins, patches = ax.hist(x, num_bins, normed=1)
# add a 'best fit' line
y = mlab.normpdf(bins, mu, sigma)
ax.plot(bins, y, '--')
ax.set_xlabel('Smarts')
ax.set_ylabel('Probability density')
ax.set_title(r'Histogram of IQ: $\mu=100$, $\sigma=15$')
# Tweak spacing to prevent clipping of ylabel
fig.tight_layout()
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

2848

Chapter 105. statistics Examples


Demo of the histogram function’s different `histtype` settings

* Histogram with step curve that has a color fill.
* Histogram with custom and unequal bin widths.

Selecting different bin counts and sizes can significantly affect the shape of a histogram. The Astropy docs have a great section on how to select these parameters: http://docs.astropy.org/en/stable/visualization/histogram.html

```python
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(0)
mu = 200
sigma = 25
x = np.random.normal(mu, sigma, size=100)

fig, (ax0, ax1) = plt.subplots(ncols=2, figsize=(8, 4))

ax0.hist(x, 20, normed=1, histtype='stepfilled', facecolor='g', alpha=0.75)
ax0.set_title('stepfilled')

# Create a histogram by providing the bin edges (unequally spaced).
bins = [100, 150, 180, 195, 205, 220, 250, 300]
ax1.hist(x, bins, histtype='bar', rwidth=0.8)
ax1.set_title('unequal bins')
```

---

**105.12 statistics example code: histogram_demo_histtypes.py**

![Histogram with step curve and unequal bins](image)

---

---

---
ax1.hist(x, bins, normed=1, histtype='bar', rwidth=0.8)
ax1.set_title('unequal bins')
fig.tight_layout()
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

105.13 statistics example code: histogram_demo_multihist.py

Demo of the histogram (hist) function with multiple data sets

Plot histogram with multiple sample sets and demonstrate:

* Use of legend with multiple sample sets
* Stacked bars
* Step curve with no fill
* Data sets of different sample sizes
Selecting different bin counts and sizes can significantly affect the shape of a histogram. The Astropy docs have a great section on how to select these parameters:

```python
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(0)
n_bins = 10
x = np.random.randn(1000, 3)

fig, axes = plt.subplots(nrows=2, ncols=2)
ax0, ax1, ax2, ax3 = axes.flatten()

colors = ['red', 'tan', 'lime']
ax0.hist(x, n_bins, normed=1, histtype='bar', color=colors, label=colors)
ax0.legend(prop={'size': 10})
ax0.set_title('bars with legend')

ax1.hist(x, n_bins, normed=1, histtype='bar', stacked=True)
ax1.set_title('stacked bar')

ax2.hist(x, n_bins, histtype='step', stacked=True, fill=False)
ax2.set_title('stack step (unfilled)')

# Make a multiple-histogram of data-sets with different length.
x_multi = [np.random.randn(n) for n in [10000, 5000, 2000]]
ax3.hist(x_multi, n_bins, histtype='bar')
ax3.set_title('different sample sizes')

fig.tight_layout()
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Demo of how to produce multiple histograms side by side

This example plots horizontal histograms of different samples along a categorical x-axis. Additionally, the histograms are plotted to be symmetrical about their x-position, thus making them very similar to violin plots.

To make this highly specialized plot, we can't use the standard `hist` method. Instead we use `barh` to draw the horizontal bars directly. The vertical positions and lengths of the bars are computed via the `np.histogram` function. The histograms for all the samples are computed using the same range (min and max values) and number of bins, so that the bins for each sample are in the same vertical positions.

Selecting different bin counts and sizes can significantly affect the shape of a histogram. The Astropy docs have a great section on how to select these parameters:

```python
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(0)
number_of_bins = 20

# An example of three data sets to compare
number_of_data_points = 387
labels = ['A', 'B', 'C']
data_sets = [np.random.normal(0, 1, number_of_data_points),
             np.random.normal(6, 1, number_of_data_points),
             np.random.normal(-3, 1, number_of_data_points)]

# Computed quantities to aid plotting
hist_range = (np.min(data_sets), np.max(data_sets))
binned_data_sets = [
    np.histogram(d, range=hist_range, bins=number_of_bins)[0]
    for d in data_sets
]
binned_maximums = np.max(binned_data_sets, axis=1)
x_locations = np.arange(0, sum(binned_maximums), np.max(binned_maximums))

# The bin_edges are the same for all of the histograms
bin_edges = np.linspace(hist_range[0], hist_range[1], number_of_bins + 1)
centers = 0.5 * (bin_edges + np.roll(bin_edges, 1))[:-1]
heights = np.diff(bin_edges)

# Cycle through and plot each histogram
fig, ax = plt.subplots()
for x_loc, binned_data in zip(x_locations, binned_data_sets):
    lefts = x_loc - 0.5 * binned_data
    ax.barh(centers, binned_data, height=heights, left=lefts)

ax.set_xticks(x_locations)
ax.set_xticklabels(labels)

ax.set_ylabel("Data values")
ax.set_xlabel("Data sets")
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Violin Plotting Examples

Custom violinplot 1

Custom violinplot 2

Custom violinplot 3

Custom violinplot 4

Custom violinplot 5

Custom violinplot 6

===

Demo of the basics of violin plots

Violin plots are similar to histograms and box plots in that they show an abstract representation of the probability distribution of the sample. Rather than showing counts of data points that fall into bins or order statistics, violin plots use kernel density estimation (KDE) to compute an empirical distribution of the sample. That computation is controlled by several parameters. This example demonstrates how to modify the number of points at which the KDE is evaluated (`points`) and how to modify the band-width of the KDE (`bw_method`).
For more information on violin plots and KDE, the scikit-learn docs have a great section: http://scikit-learn.org/stable/modules/density.html

```python
import random
import numpy as np
import matplotlib.pyplot as plt

# fake data
fs = 10  # fontsize
pos = [1, 2, 4, 5, 7, 8]
data = [np.random.normal(0, std, size=100) for std in pos]

fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(6, 6))

axes[0, 0].violinplot(data, pos, points=20, widths=0.3,
                      showmeans=True, showextrema=True, showmedians=True)
axes[0, 0].set_title('Custom violinplot 1', fontsize=fs)

axes[0, 1].violinplot(data, pos, points=40, widths=0.5,
                      showmeans=True, showextrema=True, showmedians=True,
                      bw_method='silverman')
axes[0, 1].set_title('Custom violinplot 2', fontsize=fs)

axes[0, 2].violinplot(data, pos, points=60, widths=0.7,
                      showextrema=True, showmedians=True, bw_method=0.5)
axes[0, 2].set_title('Custom violinplot 3', fontsize=fs)

axes[1, 0].violinplot(data, pos, points=80, vert=False, widths=0.7,
                      showmeans=True, showextrema=True, showmedians=True)
axes[1, 0].set_title('Custom violinplot 4', fontsize=fs)

axes[1, 1].violinplot(data, pos, points=100, vert=False, widths=0.9,
                      showmeans=True, showextrema=True, showmedians=True,
                      bw_method='silverman')
axes[1, 1].set_title('Custom violinplot 5', fontsize=fs)

axes[1, 2].violinplot(data, pos, points=200, vert=False, widths=1.1,
                      showmeans=True, showextrema=True, showmedians=True,
                      bw_method=0.5)
axes[1, 2].set_title('Custom violinplot 6', fontsize=fs)

for ax in axes.flatten():
    ax.set_yticklabels([])

fig.suptitle("Violin Plotting Examples")
fig.subplots_adjust(hspace=0.4)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
106.1 style_sheets example code: plot_bmh.py

```

'bmh' style sheet

This example demonstrates the style used in the Bayesian Methods for Hackers
online book.

```
````
from numpy.random import beta
import matplotlib.pyplot as plt

plt.style.use('bmh')

def plot_beta_hist(ax, a, b):
    ax.hist(beta(a, b, size=10000), histtype="stepfilled",
            bins=25, alpha=0.8, normed=True)

fig, ax = plt.subplots()
plot_beta_hist(ax, 10, 10)
plot_beta_hist(ax, 4, 12)
plot_beta_hist(ax, 50, 12)
plot_beta_hist(ax, 6, 55)
ax.set_title("'bmh' style sheet")

plt.show()
````

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
106.2 style_sheets example code: plot_dark_background.py

```python
# Dark background style sheet

This example demonstrates the "dark_background" style, which uses white for elements that are typically black (text, borders, etc). Note that not all plot elements default to colors defined by an rc parameter.

```
for s in shift:
    ax.plot(x, np.sin(x + s), 'o-')
ax.set_xlabel('x-axis')
ax.set_ylabel('y-axis')
ax.set_title("'dark_background' style sheet")
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

106.3 style_sheets example code: plot_fivethirtyeight.py

'fivethirtyeight' style sheet

""
"""
===========================
FiveThirtyEight style sheet
===========================

This shows an example of the "fivethirtyeight" styling, which tries to replicate the styles from FiveThirtyEight.com.
""

from matplotlib import pyplot as plt
```python
import numpy as np

plt.style.use('fivethirtyeight')

x = np.linspace(0, 10)

# Fixing random state for reproducibility
np.random.seed(19680801)

fig, ax = plt.subplots()

ax.plot(x, np.sin(x) + x + np.random.randn(50))
ax.plot(x, np.sin(x) + 0.5 * x + np.random.randn(50))
ax.plot(x, np.sin(x) + 2 * x + np.random.randn(50))
ax.plot(x, np.sin(x) - 0.5 * x + np.random.randn(50))
ax.plot(x, np.sin(x) - 2 * x + np.random.randn(50))
ax.plot(x, np.sin(x) + np.random.randn(50))

ax.set_title("'fivethirtyeight' style sheet")

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
ggplot style sheet

This example demonstrates the "ggplot" style, which adjusts the style to emulate ggplot_ (a popular plotting package for R_).

These settings were shamelessly stolen from [1]_ (with permission).


.. _ggplot: http://ggplot2.org/
.. _R: https://www.r-project.org/

```python
import numpy as np
import matplotlib.pyplot as plt

plt.style.use('ggplot')

fig, axes = plt.subplots(ncols=2, nrows=2)
```

---

106.4 style_sheets example code: plot_ggplot.py
ax1, ax2, ax3, ax4 = axes.ravel()

# scatter plot (Note: `plt.scatter` doesn't use default colors)
x, y = np.random.normal(size=(2, 200))
ax1.plot(x, y, 'o')

# sinusoidal lines with colors from default color cycle
L = 2*np.pi
x = np.linspace(0, L)
ncolors = len(plt.rcParams['axes.prop_cycle'])
shift = np.linspace(0, L, ncolors, endpoint=False)
for s in shift:
    ax2.plot(x, np.sin(x + s), '-')
ax2.margins(0)

# bar graphs
x = np.arange(5)
y1, y2 = np.random.randint(1, 25, size=(2, 5))
width = 0.25
ax3.bar(x, y1, width)
ax3.bar(x + width, y2, width,
       color=list(plt.rcParams['axes.prop_cycle'])[2]['color'])
ax3.set_xticks(x + width)
ax3.set_xticklabels(['a', 'b', 'c', 'd', 'e'])

# circles with colors from default color cycle
for i, color in enumerate(plt.rcParams['axes.prop_cycle']):
    xy = np.random.normal(size=2)
    ax4.add_patch(plt.Circle(xy, radius=0.3, color=color['color']))
ax4.axis('equal')
ax4.margins(0)

plt.show()
Grayscale style sheet

This example demonstrates the "grayscale" style sheet, which changes all colors that are defined as rc parameters to grayscale. Note, however, that not all plot elements default to colors defined by an rc parameter.

```python
import numpy as np
import matplotlib.pyplot as plt

def color_cycle_example(ax):
    L = 6
    x = np.linspace(0, L)
    ncolors = len(plt.rcParams['axes.prop_cycle'])
    shift = np.linspace(0, L, ncolors, endpoint=False)
    for s in shift:
        ax.plot(x, np.sin(x + s), 'o-')
```

```python
def image_and_patch_example(ax):
    ax.imshow(np.random.random(size=(20, 20)), interpolation='none')
    c = plt.Circle((5, 5), radius=5, label='patch')
    ax.add_patch(c)

plt.style.use('grayscale')

fig, (ax1, ax2) = plt.subplots(ncols=2)
fig.suptitle("'grayscale' style sheet")
color_cycle_example(ax1)
image_and_patch_example(ax2)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Chapter 106. style_sheets Examples
106.6. style_sheets example code: style_sheets_reference.py
""
======================
Style sheets reference
======================

This script demonstrates the different available style sheets on a common set of example plots: scatter plot, image, bar graph, patches, line plot and histogram,

""

import numpy as np
import matplotlib.pyplot as plt

def plot_scatter(ax, prng, nb_samples=100):
    """Scatter plot.
    ""
    for mu, sigma, marker in [(-0.5, 0.75, 'o'), (0.75, 1., 's')]:
        x, y = prng.normal(loc=mu, scale=sigma, size=(2, nb_samples))
        ax.plot(x, y, ls='none', marker=marker)
        ax.set_xlabel('X-label')
    return ax

def plot_colored_sinuosidal_lines(ax):
    """Plot sinusoidal lines with colors following the style color cycle.
    ""

106.6. style_sheets example code: style_sheets_reference.py
L = 2 * np.pi
x = np.linspace(0, L)
nb_colors = len(plt.rcParams['axes.prop_cycle'])
shift = np.linspace(0, L, nb_colors, endpoint=False)
for s in shift:
    ax.plot(x, np.sin(x + s), '-')
ax.set_xlim([x[0], x[-1]])
return ax

def plot_bar_graphs(ax, prng, min_value=5, max_value=25, nb_samples=5):
    """Plot two bar graphs side by side, with letters as x-tick labels."
    x = np.arange(nb_samples)
ya, yb = prng.randint(min_value, max_value, size=(2, nb_samples))
width = 0.25
ax.bar(x, ya, width)
tax(ax.set_xticks(x + width), ax.set_xticklabels(['a', 'b', 'c', 'd', 'e']))
return ax

def plot_colored_circles(ax, prng, nb_samples=15):
    """Plot circle patches.
    NB: draws a fixed amount of samples, rather than using the length of
    the color cycle, because different styles may have different numbers
    of colors.
    """
    for sty_dict, j in zip(plt.rcParams['axes.prop_cycle'], range(nb_samples)):
        ax.add_patch(plt.Circle(prng.normal(scale=3, size=2),
                                radius=1.0, color=sty_dict['color']))
    # Force the limits to be the same across the styles (because different
    # styles may have different numbers of available colors).
    ax.set_xlim([-4, 8])
    ax.set_ylim([-5, 6])
    ax.set_aspect('equal', adjustable='box')  # to plot circles as circles
    return ax

def plot_image_and_patch(ax, prng, size=(20, 20)):
    """Plot an image with random values and superimpose a circular patch.
    """
    values = prng.random_sample(size=size)
    ax.imshow(values, interpolation='none')
c = plt.Circle((5, 5), radius=5, label='patch')
    ax.add_patch(c)
    # Remove ticks
    ax.set_xticks([])
    ax.set_yticks([])
def plot_histograms(ax, prng, nb_samples=10000):
    """Plot 4 histograms and a text annotation."
    for a, b in params:
        values = prng.beta(a, b, size=nb_samples)
        ax.hist(values, histtype="stepfilled", bins=30, alpha=0.8, normed=True)
        ax.annotate('Annotation', xy=(0.25, 4.25), xycoords='data',
                     xytext=(0.9, 0.9), textcoords='axes fraction',
                     va="top", ha="right",
                     bbox=dict(boxstyle="round", alpha=0.2),
                     arrowprops=dict(
                         arrowstyle="->",
                         connectionstyle="angle,angleA=-95,angleB=35,rad=10"),
                     )
    return ax

def plot_figure(style_label=""):
    """Setup and plot the demonstration figure with a given style."
    # Use a dedicated RandomState instance to draw the same "random" values
    # across the different figures.
    prng = np.random.RandomState(96917002)

    # Tweak the figure size to be better suited for a row of numerous plots:
    # double the width and halve the height. NB: use relative changes because
    # some styles may have a figure size different from the default one.
    (fig_width, fig_height) = plt.rcParams['figure.figsize']
    fig_size = [fig_width * 2, fig_height / 2]

    fig, axes = plt.subplots(ncols=6, nrows=1, num=style_label,
                             figsize=fig_size, squeeze=True)
    axes[0].set_ylabel(style_label)

    plot_scatter(axes[0], prng)
    plot_image_and_patch(axes[1], prng)
    plot_bar_graphs(axes[2], prng)
    plot_colored_circles(axes[3], prng)
    plot_colored_sinusoidal_lines(axes[4])
    plot_histograms(axes[5], prng)

    fig.tight_layout()
    return fig

if __name__ == '__main__':
    # Setup a list of all available styles, in alphabetical order but
    # the 'default' and 'classic' ones, which will be forced resp. in
# first and second position.
style_list = list(plt.style.available)  # *new* list: avoids side effects.
style_list.remove('classic')  # 'classic' is in the list: first remove it.
style_list.sort()
style_list.insert(0, u'default')
style_list.insert(1, u'classic')

# Plot a demonstration figure for every available style sheet.
for style_label in style_list:
    with plt.style.context(style_label):
        fig = plot_figure(style_label=style_label)

plt.show()
Demo of how to display two scales on the left and right y axis.

This example uses the Fahrenheit and Celsius scales.

```python
import matplotlib.pyplot as plt
import numpy as np
```
```python
def fahrenheit2celsius(temp):
    """
    Returns temperature in Celsius.
    """
    return (5. / 9.) * (temp - 32)

def convert_ax_c_to_celsius(ax_f):
    """
    Update second axis according with first axis.
    """
    y1, y2 = ax_f.get_ylim()
    ax_c.set_ylim(fahrenheit2celsius(y1), fahrenheit2celsius(y2))
    ax_c.figure.canvas.draw()

fig, ax_f = plt.subplots()
ax_c = ax_f.twinx()

# automatically update ylim of ax2 when ylim of ax1 changes.
ax_f.callbacks.connect("ylim_changed", convert_ax_c_to_celsius)
ax_f.plot(np.linspace(-40, 120, 100))
ax_f.set_xlim(0, 100)

ax_f.set_title('Two scales: Fahrenheit and Celsius')
ax_f.set_ylabel('Fahrenheit')
ax_c.set_ylabel('Celsius')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see `Search examples`)

import numpy as np
import matplotlib.pyplot as plt

x1 = np.linspace(0.0, 5.0)
x2 = np.linspace(0.0, 2.0)

y1 = np.cos(2 * np.pi * x1) * np.exp(-x1)
y2 = np.cos(2 * np.pi * x2)

plt.subplot(2, 1, 1)
plt.plot(x1, y1, 'o-')
plt.title('A tale of 2 subplots')
plt.ylabel('Damped oscillation')

plt.subplot(2, 1, 2)
plt.plot(x2, y2, '.')
plt.xlabel('time (s)')
plt.ylabel('Undamped')

---

Simple demo with multiple subplots.

---
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
108.1 tests example code: backend_driver.py

This is used to drive many of the examples across the backends, for regression testing, and comparing backend efficiency.

You can specify the backends to be tested either via the --backends switch, which takes a comma-separated list, or as separate arguments, e.g.

    python backend_driver.py agg ps

would test the agg and ps backends. If no arguments are given, a default list of backends will be tested.

Interspersed with the backend arguments can be switches for the Python interpreter executing the tests. If entering such arguments causes an option parsing error with the driver script, separate them from driver switches with a --.

```
from __future__ import print_function, division
import os
import time
import sys
import glob
from optparse import OptionParser

import matplotlib.rcsetup as rcsetup
from matplotlib.cbook import Bunch, dedent

all_backends = list(rcsetup.all_backends)  # to leave the original list alone

# actual physical directory for each dir
dirs = dict(files=os.path.join('..', 'lines_bars_and_markers'),
            shapes=os.path.join('..', 'shapes_and_collections'),
```
images=os.path.join('..', 'images_contours_and_fields'),
pie=os.path.join('..', 'pie_and_polar_charts'),
text=os.path.join('..', 'text_labels_and_annotations'),
ticks=os.path.join('..', 'ticks_and_spines'),
subplots=os.path.join('..', 'subplots_axes_and_figures'),
specialty=os.path.join('..', 'specialty_plots'),
showcase=os.path.join('..', 'showcase'),
pylab=os.path.join('..', 'pylab_examples'),
api=os.path.join('..', 'api'),
units=os.path.join('..', 'units'),
mplot3d=os.path.join('..', 'mplot3d'),
colors=os.path.join('..', 'color'))

# files in each dir
files = dict()

files['lines'] = [
    'barh_demo.py',
    'fill_demo.py',
    'fill_demo_features.py',
    'line_demo_dash_control.py',
    'line_styles_reference.py',
    'scatter_with_legend.py',
]

files['shapes'] = [
    'path_patch_demo.py',
    'scatter_demo.py',
]

files['colors'] = [
    'color_cycle_default.py',
    'color_cycle_demo.py',
]

files['images'] = [
    'image_demo.py',
    'contourf_log.py',
]

files['statistics'] = [
    'errorbar_demo.py',
    'errorbar_demo_features.py',
    'histogram_demo_cumulative.py',
    'histogram_demo_features.py',
    'histogram_demo_histtypes.py',
    'histogram_demo_multihist.py',
]

files['pie'] = [
    'pie_demo.py',
    'polar_bar_demo.py',
]
['polar_scatter_demo.py',
]

files['text_labels_and_annotations'] = [
    'text_demo_fontdict.py',
    'unicode_demo.py',
]

files['ticks_and_spines'] = [
    'spines_demo_bounds.py',
    'ticklabels_demo_rotation.py',
]

files['subplots_axes_and_figures'] = [
    'subplot_demo.py',
]

files['showcase'] = [
    'integral_demo.py',
]

files['pylab'] = [
    'accented_text.py',
    'alignment_test.py',
    'annotation_demo.py',
    'annotation_demo.py',
    'annotation_demo2.py',
    'annotation_demo2.py',
    'anscombe.py',
    'arctest.py',
    'arrow_demo.py',
    'axes_demo.py',
    'axes_props.py',
    'axhspan_demo.py',
    'axis_equal_demo.py',
    'bar_stacked.py',
    'barb_demo.py',
    'barchart_demo.py',
    'barcode_demo.py',
    'boxplot_demo.py',
    'broken_barh.py',
    'cohere_demo.py',
    'color_by_yvalue.py',
    'color_demo.py',
    'colorbar_tick_labelling_demo.py',
    'contour_demo.py',
    'contour_image.py',
    'contour_label_demo.py',
    'contourf_demo.py',
    'coords_demo.py',
    'coords_report.py',
    'csd_demo.py',
    'cursor_demo.py',
]
'log_test.py',
'major_minor_demo1.py',
'major_minor_demo2.py',
'manual_axis.py',
'masked_demo.py',
'mathtext_demo.py',
'mathtext_examples.py',
'matshow.py',
'mri_demo.py',
'mri_with_eeg.py',
'multi_image.py',
'multiline.py',
'multiple_figs_demo.py',
'nan_test.py',
'newscalarformatter_demo.py',
'pcolor_demo.py',
'pcolor_log.py',
'pcolor_small.py',
'pie_demo2.py',
'plotfile_demo.py',
'polar_demo.py',
'polar_legend.py',
'psd_demo.py',
'psd_demo2.py',
'psd_demo3.py',
'quadmesh_demo.py',
'quiver_demo.py',
'scatter_custom_symbol.py',
'scatter_demo2.py',
'scatter_masked.py',
'scatter_profile.py',
'scatter_star_poly.py',
'#set_and_get.py',
'shared_axis_across_figures.py',
'shared_axis_demo.py',
'simple_plot.py',
'specgram_demo.py',
'spine_placement_demo.py',
'spy_demos.py',
'stem_plot.py',
'step_demo.py',
'stix_fonts_demo.py',
'subplots_adjust.py',
'symlog_demo.py',
'table_demo.py',
'text_handles.py',
'text_rotation.py',
'text_rotation_relative_to_line.py',
'transoffset.py',
'xcorr_demo.py',
'zorder_demo.py',
]
```python
files['api'] = ['agg_oo.py',
               'barchart_demo.py',
               'bbox_intersect.py',
               'collections_demo.py',
               'colorbar_only.py',
               'custom_projection_example.py',
               'custom_scale_example.py',
               'date_demo.py',
               'date_index_formatter.py',
               'donut_demo.py',
               'font_family_rc.py',
               'image_zcoord.py',
               'joinstyle.py',
               'legend_demo.py',
               'line_with_text.py',
               'logo2.py',
               'mathtext_asarray.py',
               'patch_collection.py',
               'quad_bezier.py',
               'scatter_piecharts.py',
               'span_regions.py',
               'two_scales.py',
               'unicode_minus.py',
               'watermark_image.py',
               'watermark_text.py',
               ]

files['units'] = ['annotate_with_units.py',
                  '#artist_tests.py', '# broken, fixme
                  'bar_demo2.py',
                  '#bar_unit_demo.py', '# broken, fixme
                  #ellipse_with_units.py', '# broken, fixme
                  'radian_demo.py',
                  'units_sample.py',
                  '#units_scatter.py', '# broken, fixme
                  ]

files['mplot3d'] = ['2dcollections3d_demo.py',
                    'bars3d_demo.py',
                    'contour3d_demo.py',
                    'contour3d_demo2.py',
                    'contourf3d_demo.py',
                    'lines3d_demo.py',
                    'polys3d_demo.py',
                    'scatter3d_demo.py',
                    'surface3d_demo.py',
                    'surface3d_demo2.py',
                    'text3d_demo.py',
                    ]
```
'wire3d_demo.py',
]

# dict from dir to files we know we don't want to test (e.g., examples
# not using pyplot, examples requiring user input, animation examples,
# examples that may only work in certain environs (usetex examples?),
# examples that generate multiple figures

excluded = {
    'pylab': ['__init__.py', 'toggle_images.py', ],
    'units': ['__init__.py', 'date_support.py', ],
}

def report_missing(dir, flist):
    'report the py files in dir that are not in flist'
    globstr = os.path.join(dir, '*\'.py')
    fnames = glob.glob(globstr)
    pyfiles = set([os.path.split(fullpath)[-1] for fullpath in set(fnames)])
    exclude = set(excluded.get(dir, []))
    flist = set(flist)
    missing = list(pyfiles - flist - exclude)
    missing.sort()
    if missing:
        print('%s files not tested: %s' % (dir, ', '.join(missing)))

def report_all_missing(directories):
    for f in directories:
        report_missing(dirs[f], files[f])

# tests known to fail on a given backend

failbackend = dict(
    svg=('tex_demo.py', ),
    agg=('hyperlinks.py', ),
    pdf=('hyperlinks.py', ),
    ps=('hyperlinks.py', ),
)

try:
    import subprocess

    def run(arglist):
        try:
            ret = subprocess.call(arglist)
        except KeyboardInterrupt:
            sys.exit()
        else:
return ret
except ImportError:
    def run(arglist):
        os.system(' '.join(arglist))

def drive(backend, directories, python=['python'], switches=[]):
    exclude = failbackend.get(backend, [])

    # Clear the destination directory for the examples
    path = backend
    if os.path.exists(path):
        import glob
        for fname in os.listdir(path):
            os.unlink(os.path.join(path, fname))
    else:
        os.mkdir(backend)
    failures = []

    testcases = [os.path.join(dirs[d], fname)
                 for d in directories
                 for fname in files[d]]

    for fullpath in testcases:
        print('	driving %-40s' % (fullpath))
        sys.stdout.flush()
        fpath, fname = os.path.split(fullpath)

        if fname in exclude:
            print('	Skipping %s, known to fail on backend: %s' % backend)
            continue

        basename, ext = os.path.splitext(fname)
        outfile = os.path.join(path, basename)
        tmpfile_name = '_tmp_%s.py' % basename
        tmpfile = open(tmpfile_name, 'w')

        future_imports = 'from __future__ import division, print_function'

        for line in open(fullpath):
            line_lstrip = line.lstrip()
            if line_lstrip.startswith('#'):
                tmpfile.write(line)
            elif 'unicode_literals' in line:
                future_imports = future_imports + ', unicode_literals'

        tmpfile.writelines((
            future_imports + '
',
            'import sys
',
            'sys.path.append("%s")
' % fpath.replace('\', '\\'),
            'import matplotlib
',
            'matplotlib.use("%s")
' % backend,
            'from pylab import savefig
',
            'import numpy
',
            future_imports + '
',
            'import os
',
            'os.system(
',
            ' '.join(arglist))
            ' )

    return ret
'numpy.seterr(invalid=\"ignore\")\n',
))

for line in open(fullpath):
    line_lstrip = line.lstrip()
    if (line_lstrip.startswith('from __future__ import') or
        line_lstrip.startswith('matplotlib.use') or
        line_lstrip.startswith('savefig') or
        line_lstrip.startswith('show')):
        continue
    tmpfile.write(line)

if backend in rcsetup.interactive_bk:
    tmpfile.write('show()')
else:
    tmpfile.write('\nsavefig(r''%s'', dpi=150)' % outfile)

tmpfile.close()

start_time = time.time()
program = [x % {'name': basename} for x in python]
ret = run(program + [tmpfile_name] + switches)
end_time = time.time()
print('%s %s' % ((end_time - start_time), ret))

os.remove(tmpfile_name)

if ret:
    failures.append(fullpath)
return failures

def parse_options():
    doc = (__doc__ and __doc__.split('\\n\\n')) or "  
    op = OptionParser(description=doc[0].strip(),
        usage='%prog [options] [--] [backends and switches]',
        #epilog='\n'.join(doc[1:]) # epilog not supported on my python2.4
    op.disable_interspersed_args()
    op.set_defaults(dirs='pylab,api,units,mplot3d',
        clean=False, coverage=False, valgrind=False)
    op.add_option('-d', '--dirs', '--directories', type='string',
        dest='dirs', help=dedent('''
Run only the tests in these directories; comma-separated list of
one or more of: pylab (or pylab_examples), api, units, mplot3d'''))
    op.add_option('-b', '--backends', type='string', dest='backends',
        help=dedent('''
Run tests only for these backends; comma-separated list of
one or more of: agg, ps, svg, pdf, template, cairo,
Default is everything except cairo.'''))
    op.add_option('--clean', action='store_true', dest='clean',
        help='Remove result directories, run no tests!')
    op.add_option('--c', '--coverage', action='store_true', dest='coverage',
        help='Run in coverage.py')
    op.add_option('--v', '--valgrind', action='store_true', dest='valgrind',
        help='Run in valgrind')
options, args = op.parse_args()
switches = [x for x in args if x.startswith('--')]
backends = [x.lower() for x in args if not x.startswith('--')]
if options.backends:
    backends += [be.lower() for be in options.backends.split(',')]
result = Bunch(
    dirs=options.dirs.split(','),
    backends=backends or ['agg', 'ps', 'svg', 'pdf', 'template'],
    clean=options.clean,
    coverage=options.coverage,
    valgrind=options.valgrind,
    switches=switches)
if 'pylab_examples' in resultdirs:
    resultdirs[resultdirs.index('pylab_examples')] = 'pylab'
# print(result)
return (result)

if __name__ == '__main__':
times = {}
failures = {}
options = parse_options()

if options.clean:
    localdirs = [d for d in glob.glob('*') if os.path.isdir(d)]
    all_backends_set = set(all_backends)
    for d in localdirs:
        if d.lower() not in all_backends_set:
            continue
        print('removing %s %d' % d)
        for fname in glob.glob(os.path.join(d, '*')):
            os.remove(fname)
        os.rmdir(d)
    for fname in glob.glob('_tmp*.py'):
        os.remove(fname)
    print('all clean...')
    raise SystemExit

if options.coverage:
    python = ['coverage.py', '-x']
elif options.valgrind:
    python = ['valgrind', '--tool=memcheck', '--leak-check=yes',
              '--log-file=%%(name)s', sys.executable]
elif sys.platform == 'win32':
    python = [sys.executable]
else:
    python = [sys.executable]

report_all_missing(options.dirs)
for backend in options.backends:
    print('testing %s %s' % (backend, ' '.join(options.switches)))
t0 = time.time()
failures[backend] = \
    drive(backend, options.dirs, python, options.switches)
t1 = time.time()
times[backend] = (t1 - t0)/60.0

# print(times)
for backend, elapsed in times.items():
    print('Backend %s took %1.2f minutes to complete' % (backend, elapsed))
    failed = failures[backend]
    if failed:
        print(' Failures: %s' % failed)
    if 'template' in times:
        print('\template ratio %1.3f, template residual %1.3f' % (elapsed/times['template'], elapsed - times['template']))
109.1 text_labels_and_annotations example code: autowrap_demo.py

Matplotlib can wrap text automatically, but if it's too long, the text will be displayed slightly outside of the boundaries of the axis anyways.
import matplotlib.pyplot as plt

fig = plt.figure()
plt.axis([0, 10, 0, 10])
t = "This is a really long string that I'd rather have wrapped so that it"
   " doesn't go outside of the figure, but if it's long enough it will go"
   " off the top or bottom!"
plt.text(4, 1, t, ha='left', rotation=15, wrap=True)
plt.text(6, 5, t, ha='left', rotation=15, wrap=True)
plt.text(5, 5, t, ha='right', rotation=-15, wrap=True)
plt.text(5, 10, t, fontsize=18, style='oblique', ha='center',
            va='top', wrap=True)
plt.text(3, 4, t, family='serif', style='italic', ha='right', wrap=True)
plt.text(-1, 0, t, ha='left', rotation=-15, wrap=True)

plt.show()
The example shows how to string together several text objects.

HISTORY
-------
On the matplotlib-users list back in February 2012, Gökhan Sever asked the following question:

Is there a way in matplotlib to partially specify the color of a string?

Example:

plt.ylabel("Today is cloudy.")
How can I show "today" as red, "is" as green and "cloudy." as blue?

Thanks.

Paul Ivanov responded with this answer:

```python
import matplotlib.pyplot as plt
from matplotlib import transforms

def rainbow_text(x, y, strings, colors, ax=None, **kw):
    ""
    Take a list of `strings` and `colors` and place them next to each other, with text strings[i] being shown in colors[i].

    This example shows how to do both vertical and horizontal text, and will pass all keyword arguments to plt.text, so you can set the font size, family, etc.

    The text will get added to the `ax` axes, if provided, otherwise the currently active axes will be used.
    ""
    if ax is None:
        ax = plt.gca()
    t = ax.transData
    canvas = ax.figure.canvas

    # horizontal version
    for s, c in zip(strings, colors):
        text = ax.text(x, y, s + " ", color=c, transform=t, **kw)
        text.draw(canvas.get_renderer())
        ex = text.get_window_extent()
        t = transforms.offset_copy(text._transform, x=ex.width, units='dots')

    # vertical version
    for s, c in zip(strings, colors):
        text = ax.text(x, y, s + " ", color=c, transform=t,
                       rotation=90, va='bottom', ha='center', **kw)
        text.draw(canvas.get_renderer())
```
```python
ex = text.get_window_extent()
t = transforms.offset_copy(text._transform, y=ex.height, units='dots')

rainbow_text(0, 0, "all unicorns poop rainbows ! ! !".split(),
    ['red', 'cyan', 'brown', 'green', 'blue', 'purple', 'black'],
    size=16)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

109.3 text_labels_and_annotations example code: text_demo_fontdict.py

Damped exponential decay

\[ \cos(2\pi t) \exp(-t) \]

Controlling style of text and labels using a dictionary

This example shows how to share parameters across many text objects and labels by creating a dictionary of options passed across several functions.
import numpy as np
import matplotlib.pyplot as plt

font = {'family': 'serif',
       'color': 'darkred',
       'weight': 'normal',
       'size': 16,
       }

x = np.linspace(0.0, 5.0, 100)
y = np.cos(2*np.pi*x) * np.exp(-x)

plt.plot(x, y, 'k')
plt.title('Damped exponential decay', fontdict=font)
plt.text(2, 0.65, r'$\cos(2 \pi t) \exp(-t)$', fontdict=font)
plt.xlabel('time (s)', fontdict=font)
plt.ylabel('voltage (mV)', fontdict=font)

# Tweak spacing to prevent clipping of ylabel
plt.subplots_adjust(left=0.15)
plt.show()
109.4 text_labels_and_annotations example code: unicode_demo.py

```python
# -*- coding: utf-8 -*-

from __future__ import unicode_literals

import matplotlib.pyplot as plt

plt.title('Développés et fabriqués')
plt.xlabel('réactivité nous permettent d’être sélectionnés et adoptés')
plt.ylabel('André was here!
Institut für Festkörperphysik')
plt.text(0.2, 0.8, 'Institut für Festkörperphysik', rotation=45)
plt.text(0.4, 0.2, 'AVA (check kerning)')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
110.1 ticks_and_spines example code: spines_demo.py

```python
import numpy as np
```

This demo compares:
- normal axes, with spines on all four sides;
- an axes with spines only on the left and bottom;
- an axes using custom bounds to limit the extent of the spine.
import matplotlib.pyplot as plt

x = np.linspace(0, 2 * np.pi, 100)
y = 2 * np.sin(x)

fig, (ax0, ax1, ax2) = plt.subplots(nrows=3)

ax0.plot(x, y)
ax0.set_title('normal spines')

ax1.plot(x, y)
ax1.set_title('bottom-left spines')

# Hide the right and top spines
ax1.spines['right'].set_visible(False)
ax1.spines['top'].set_visible(False)
# Only show ticks on the left and bottom spines
ax1.yaxis.set_ticks_position('left')
ax1.xaxis.set_ticks_position('bottom')

ax2.plot(x, y)

# Only draw spine between the y-ticks
ax2.spines['left'].set_bounds(-1, 1)
# Hide the right and top spines
ax2.spines['right'].set_visible(False)
ax2.spines['top'].set_visible(False)
# Only show ticks on the left and bottom spines
ax2.yaxis.set_ticks_position('left')
ax2.xaxis.set_ticks_position('bottom')

# Tweak spacing between subplots to prevent labels from overlapping
plt.subplots_adjust(hspace=0.5)
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
Demo of spines using custom bounds to limit the extent of the spine.

```python
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 2*np.pi, 50)
y = np.sin(x)
y2 = y + 0.1 * np.random.normal(size=x.shape)

fig, ax = plt.subplots()
ax.plot(x, y, 'k--')
ax.plot(x, y2, 'ro')

# set ticks and tick labels
ax.set_xlim((0, 2*np.pi))
ax.set_xticks([0, np.pi, 2*np.pi])
ax.set_xticklabels(['0', '$\pi$', '$2\pi$'])
ax.set_ylim((-1.5, 1.5))
ax.set_yticks([-1, 0, 1])
```

110.2. ticks_and_spines example code: spines_demo_bounds.py

2897
# Only draw spine between the y-ticks
ax.spines['left'].set_bounds(-1, 1)
# Hide the right and top spines
ax.spines['right'].set_visible(False)
ax.spines['top'].set_visible(False)
# Only show ticks on the left and bottom spines
ax.yaxis.set_ticks_position('left')
ax.xaxis.set_ticks_position('bottom')

plt.show()
```python
import numpy as np
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
image = np.random.uniform(size=(10, 10))
ax.imshow(image, cmap=plt.cm.gray, interpolation='nearest')
ax.set_title('dropped spines')

# Move left and bottom spines outward by 10 points
ax.spines['left'].set_position(('outward', 10))
ax.spines['bottom'].set_position(('outward', 10))
# Hide the right and top spines
ax.spines['right'].set_visible(False)
ax.spines['top'].set_visible(False)

# Only show ticks on the left and bottom spines
ax.yaxis.set_ticks_position('left')
ax.xaxis.set_ticks_position('bottom')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

### 110.4 ticks_and_spines example code: tick-formatters.py

- `NullFormatter()`

- `FixedFormatter(['', '0', '1', ...])`

- `FuncFormatter(lambda x, pos: "[%.2f]" % x)`

- `FormatStrFormatter('>%d<')`

- `ScalarFormatter()`

- `StrMethodFormatter('{x}')`
Show the different tick formatters.

---

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker

# Setup a plot such that only the bottom spine is shown

def setup(ax):
    ax.spines['right'].set_color('none')
    ax.spines['left'].set_color('none')
    ax.yaxis.set_major_locator(ticker.NullLocator())
    ax.spines['top'].set_color('none')
    ax.xaxis.set_ticks_position('bottom')
    ax.tick_params(which='major', width=1.00, length=5)
    ax.tick_params(which='minor', width=0.75, length=2.5, labelsize=10)
    ax.set_xlim(0, 5)
    ax.set_ylim(0, 1)
    ax.patch.set_alpha(0.0)

plt.figure(figsize=(8, 5))
n = 6

# Null formatter
ax = plt.subplot(n, 1, 1)
setup(ax)
ax.xaxis.set_major_locator(ticker.MultipleLocator(1.0))
ax.xaxis.set_minor_locator(ticker.MultipleLocator(0.25))
ax.xaxis.set_major_formatter(ticker.NullFormatter())
ax.xaxis.set_minor_formatter(ticker.NullFormatter())
ax.text(0.0, 0.1, "NullFormatter()", fontsize=16, transform=ax.transAxes)

# Fixed formatter
ax = plt.subplot(n, 1, 2)
setup(ax)
majors = ["", "0", "1", "2", "3", "4", "5"]
ax.xaxis.set_major_formatter(ticker.FixedFormatter(majors))
ax.text(0.0, 0.1, "FixedFormatter(['', '0', '1', '2', '3', '4', '5'])", fontsize=15, transform=ax.transAxes)
```

---
# Func formatter

```python
def major_formatter(x, pos):
    return "%.2f" % x
```

```python
ax = plt.subplot(n, 1, 3)
setup(ax)
ax.xaxis.set_major_locator(ticker.MultipleLocator(1.00))
ax.xaxis.set_minor_locator(ticker.MultipleLocator(0.25))
ax.xaxis.set_major_formatter(ticker.FuncFormatter(major_formatter))
ax.text(0.0, 0.1, 'FuncFormatter(lambda x, pos: 
                          "%.2f" % x)
',
fontsize=15, transform=ax.transAxes)
```

# FormatStr formatter

```python
ax = plt.subplot(n, 1, 4)
setup(ax)
ax.xaxis.set_major_locator(ticker.MultipleLocator(1.00))
ax.xaxis.set_minor_locator(ticker.MultipleLocator(0.25))
ax.xaxis.set_major_formatter(ticker.FormatStrFormatter('>%d<'))
ax.text(0.0, 0.1, 'FormatStrFormatter('>%d<')',
        fontsize=15, transform=ax.transAxes)
```

# Scalar formatter

```python
ax = plt.subplot(n, 1, 5)
setup(ax)
ax.xaxis.set_major_locator(ticker.AutoLocator())
ax.xaxis.set_minor_locator(ticker.AutoMinorLocator())
ax.xaxis.set_major_formatter(ticker.ScalarFormatter(useMathText=True))
ax.text(0.0, 0.1, 'ScalarFormatter()', fontsize=15, transform=ax.transAxes)
```

# StrMethod formatter

```python
ax = plt.subplot(n, 1, 6)
setup(ax)
ax.xaxis.set_major_locator(ticker.MultipleLocator(1.00))
ax.xaxis.set_minor_locator(ticker.MultipleLocator(0.25))
ax.xaxis.set_major_formatter(ticker.StrMethodFormatter('{x}'))
ax.text(0.0, 0.1, 'StrMethodFormatter({x})',
        fontsize=15, transform=ax.transAxes)
```

# Push the top of the top axes outside the figure because we only show the bottom spine.

```python
plt.subplots_adjust(left=0.05, right=0.95, bottom=0.05, top=1.05)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
110.5 ticks_and_spines example code: tick-locators.py

NullLocator()

MultipleLocator(0.5)

FixedLocator([0, 1, 5])

LinearLocator(numticks=3)

IndexLocator(base=0.5, offset=0.25)

AutoLocator()

MaxNLocator(n=4)

LogLocator(base=10, numticks=15)

"""
===============
Tick locators
===============

Show the different tick locators.
"""

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker

# Setup a plot such that only the bottom spine is shown
def setup(ax):
    ax.spines['right'].set_color('none')
    ax.spines['left'].set_color('none')
    ax.yaxis.set_major_locator(ticker.NullLocator())
    ax.spines['top'].set_color('none')
    ax.xaxis.set_ticks_position('bottom')

"""
ax.tick_params(which='major', width=1.00)
ax.tick_params(which='major', length=5)
ax.tick_params(which='minor', width=0.75)
ax.tick_params(which='minor', length=2.5)
ax.set_xlim(0, 5)
ax.set_ylim(0, 1)
ax.patch.set_alpha(0.0)

plt.figure(figsize=(8, 6))
n = 8

# Null Locator
ax = plt.subplot(n, 1, 1)
skipup(ax)
ax.xaxis.set_major_locator(ticker.NullLocator())
ax.xaxis.set_minor_locator(ticker.NullLocator())
ax.text(0.0, 0.1, "NullLocator()", fontsize=14, transform=ax.transAxes)

# Multiple Locator
ax = plt.subplot(n, 1, 2)
skipup(ax)
ax.xaxis.set_major_locator(ticker.MultipleLocator(0.5))
ax.xaxis.set_minor_locator(ticker.MultipleLocator(0.1))
ax.text(0.0, 0.1, "MultipleLocator(0.5)", fontsize=14,
        transform=ax.transAxes)

# Fixed Locator
ax = plt.subplot(n, 1, 3)
skipup(ax)
majors = [0, 1, 5]
ax.xaxis.set_major_locator(ticker.FixedLocator(majors))
minors = np.linspace(0, 1, 11)[1:-1]
ax.xaxis.set_minor_locator(ticker.FixedLocator(minors))
ax.text(0.0, 0.1, "FixedLocator([0, 1, 5])", fontsize=14,
        transform=ax.transAxes)

# Linear Locator
ax = plt.subplot(n, 1, 4)
skipup(ax)
ax.axis.set_major_locator(ticker.LinearLocator(3))
ax.axis.set_minor_locator(ticker.LinearLocator(31))
ax.text(0.0, 0.1, "LinearLocator(numticks=3)",
        fontsize=14, transform=ax.transAxes)

# Index Locator
ax = plt.subplot(n, 1, 5)
skipup(ax)
ax.plot(range(0, 5), [0]*5, color='White')
ax.axis.set_major_locator(ticker.IndexLocator(base=.5, offset=.25))
ax.text(0.0, 0.1, "IndexLocator(base=0.5, offset=0.25)",
        fontsize=14, transform=ax.transAxes)
# Auto Locator
ax = plt.subplot(n, 1, 6)
setup(ax)
ax.xaxis.set_major_locator(ticker.AutoLocator())
ax.xaxis.set_minor_locator(ticker.AutoMinorLocator())
ax.text(0.0, 0.1, "AutoLocator()", fontsize=14, transform=ax.transAxes)

# MaxN Locator
ax = plt.subplot(n, 1, 7)
setup(ax)
ax.xaxis.set_major_locator(ticker.MaxNLocator(4))
ax.xaxis.set_minor_locator(ticker.MaxNLocator(40))
ax.text(0.0, 0.1, "MaxNLocator(n=4)", fontsize=14, transform=ax.transAxes)

# Log Locator
ax = plt.subplot(n, 1, 8)
setup(ax)
ax.set_xlim(10**3, 10**10)
ad.set_xscale('log')
ax.xaxis.set_major_locator(ticker.LogLocator(base=10.0, numticks=15))
ax.text(0.0, 0.1, "LogLocator(base=10, numticks=15)",
        fontsize=15, transform=ax.transAxes)

# Push the top of the top axes outside the figure because we only show the
# bottom spine.
plt.subplots_adjust(left=0.05, right=0.95, bottom=0.05, top=1.05)

plt.show()
Setting tick labels from a list of values

Using ax.set_xticks causes the tick labels to be set on the currently chosen ticks. However, you may want to allow matplotlib to dynamically choose the number of ticks and their spacing.

In this case it may be better to determine the tick label from the value at the tick. The following example shows how to do this.

**NB:** The MaxNLocator is used here to ensure that the tick values take integer values.

```python
import matplotlib.pyplot as plt
from matplotlib.ticker import FuncFormatter, MaxNLocator
fig = plt.figure()
ax = fig.add_subplot(111)
xs = range(26)
```
```
ys = range(26)
labels = list('abcdefghijklmnopqrstuvwxyz')

def format_fn(tick_val, tick_pos):
    if int(tick_val) in xs:
        return labels[int(tick_val)]
    else:
        return '

ax.xaxis.set_major_formatter(FuncFormatter(format_fn))
ax.xaxis.set_major_locator(MaxNLocator(integer=True))
ax.plot(xs, ys)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

110.7 ticks_and_spines example code: ticklabels_demo_rotation.py
Rotating custom tick labels
==========================

Demo of custom tick-labels with user-defined rotation.

```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4]
y = [1, 4, 9, 6]
labels = ['Frogs', 'Hogs', 'Bogs', 'Slogs']

plt.plot(x, y, 'ro')
# You can specify a rotation for the tick labels in degrees or with keywords.
plt.xticks(x, labels, rotation='vertical')
# Pad margins so that markers don't get clipped by the axes
plt.margins(0.2)
# Tweak spacing to prevent clipping of tick-labels
plt.subplots_adjust(bottom=0.15)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
111.1 units example code: annotate_with_units.py

The example illustrates how to create text and arrow annotations using a centimeter-scale plot.
```python
import matplotlib.pyplot as plt
from basic_units import cm

fig, ax = plt.subplots()

ax.annotate("Note 01", [0.5*cm, 0.5*cm])

# xy and text both unitized
ax.annotate('local max', xy=(3*cm, 1*cm), xycoords='data',
            xytext=(0.8*cm, 0.95*cm), textcoords='data',
            arrowprops=dict(facecolor='black', shrink=0.05),
            horizontalalignment='right', verticalalignment='top')

# mixing units w/ nonunits
ax.annotate('local max', xy=(3*cm, 1*cm), xycoords='data',
            xytext=(0.8, 0.95), textcoords='axes fraction',
            arrowprops=dict(facecolor='black', shrink=0.05),
            horizontalalignment='right', verticalalignment='top')

ax.set_xlim(0*cm, 4*cm)
ax.set_ylim(0*cm, 4*cm)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
111.2 units example code: artist_tests.py

import random
import matplotlib.lines as lines
import matplotlib.patches as patches
import matplotlib.text as text
import matplotlib.collections as collections
from basic_units import cm, inch
import numpy as np
import matplotlib.pyplot as plt

Test unit support with each of the Matplotlib primitive artist types.

The axis handles unit conversions and the artists keep a pointer to their axis parent. You must initialize the artists with the axis instance if you want to use them with unit data, or else they will not know how to convert the units to scalars.

import random
import matplotlib.lines as lines
import matplotlib.patches as patches
import matplotlib.text as text
import matplotlib.collections as collections

from basic_units import cm, inch
import numpy as np
import matplotlib.pyplot as plt
fig, ax = plt.subplots()
ax.xaxis.set_units(cm)
ax.yaxis.set_units(cm)

if 0:
    # test a line collection
    # Not supported at present.
    verts = []
    for i in range(10):
        # a random line segment in inches
        verts.append(zip(*inch*10*np.random.rand(2, random.randint(2, 15))))
    lc = collections.LineCollection(verts, axes=ax)
    ax.add_collection(lc)

# test a plain-ol-line
line = lines.Line2D([0*cm, 1.5*cm], [0*cm, 2.5*cm],
                     lw=2, color='black', axes=ax)
ax.add_line(line)

if 0:
    # test a patch
    # Not supported at present.
    rect = patches.Rectangle((1*cm, 1*cm), width=5*cm, height=2*cm,
                              alpha=0.2, axes=ax)
    ax.add_patch(rect)

t = text.Text(3*cm, 2.5*cm, 'text label', ha='left', va='bottom', axes=ax)
ax.add_artist(t)

ax.set_xlim(-1*cm, 10*cm)
ax.set_ylim(-1*cm, 10*cm)
# ax.xaxis.set_units(inch)
ax.grid(True)
ax.set_title('Artists with units')
plt.show()
A plot using a variety of centimetre and inch conversions. This example shows how default unit introspection works (ax1), how various keywords can be used to set the x and y units to override the defaults (ax2, ax3, ax4) and how one can set the xlimits using scalars (ax3, current units assumed) or units (conversions applied to get the numbers to current units).

```python
import numpy as np
from basic_units import cm, inch
import matplotlib.pyplot as plt

cms = cm * np.arange(0, 10, 2)
bottom = 0 * cm
width = 0.8 * cm

fig, axs = plt.subplots(2, 2)
```
axs[0, 0].bar(cms, cms, bottom=bottom)
axs[0, 1].bar(cms, cms, bottom=bottom, width=width, xunits=cm, yunits=inch)
axs[1, 0].bar(cms, cms, bottom=bottom, width=width, xunits=inch, yunits=cm)
axs[1, 0].set_xlim(2, 6)  # scalars are interpreted in current units
axs[1, 1].bar(cms, cms, bottom=bottom, width=width, xunits=inch, yunits=inch)
axs[1, 1].set_xlim(2 * cm, 6 * cm)  # cm are converted to inches
fig.tight_layout()
plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

111.4 units example code: bar_unit_demo.py
This is the same example as
<a href="http://matplotlib.org/examples/api/barchart_demo.html">the barchart demo</a> in centimeters.

```
import numpy as np
from basic_units import cm, inch
import matplotlib.pyplot as plt

N = 5
menMeans = (150*cm, 160*cm, 146*cm, 172*cm, 155*cm)
mens = (20*cm, 30*cm, 32*cm, 10*cm, 20*cm)

fig, ax = plt.subplots()
ind = np.arange(N) # the x locations for the groups
width = 0.35 # the width of the bars
p1 = ax.bar(ind, menMeans, width, color='r', bottom=0*cm, yerr=mens)

womenMeans = (145*cm, 149*cm, 172*cm, 165*cm, 200*cm)
womenStd = (30*cm, 25*cm, 20*cm, 31*cm, 22*cm)
p2 = ax.bar(ind + width, womenMeans, width,
            color='y', bottom=0*cm, yerr=womenStd)

ax.set_title('Scores by group and gender')
ax.set_xticks(ind + width / 2)
ax.set_xticklabels(('G1', 'G2', 'G3', 'G4', 'G5'))

ax.legend((p1[0], p2[0]), ('Men', 'Women'))
ax.yaxis.set_units(inch)
ax.autoscale_view()
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see <a href="Search examples">Search examples</a>)

### 111.5 units example code: basic_units.py

```
import math
import numpy as np
import matplotlib.units as units
import matplotlib.ticker as ticker
from matplotlib.axes import Axes
from matplotlib.cbook import iterable

class ProxyDelegate(object):
```

111.5. units example code: basic_units.py 2915
def __init__(self, fn_name, proxy_type):
    self.proxy_type = proxy_type
    self.fn_name = fn_name

def __get__(self, obj, objtype=None):
    return self.proxy_type(self.fn_name, obj)

class TaggedValueMeta(type):
    def __init__(cls, name, bases, dict):
        for fn_name in cls._proxies.keys():
            try:
                dummy = getattr(cls, fn_name)
            except AttributeError:
                setattr(cls, fn_name,
                        ProxyDelegate(fn_name, cls._proxies[fn_name]))

class PassThroughProxy(object):
    def __init__(self, fn_name, obj):
        self.fn_name = fn_name
        self.target = obj.proxy_target

    def __call__(self, *args):
        fn = getattr(self.target, self.fn_name)
        ret = fn(*args)
        return ret

class ConvertArgsProxy(PassThroughProxy):
    def __init__(self, fn_name, obj):
        PassThroughProxy.__init__(self, fn_name, obj)
        self.unit = obj.unit

    def __call__(self, *args):
        converted_args = []
        for a in args:
            try:
                converted_args.append(a.convert_to(self.unit))
            except AttributeError:
                converted_args.append(TaggedValue(a, self.unit))
        converted_args = tuple([c.get_value() for c in converted_args])
        return PassThroughProxy.__call__(self, *converted_args)

class ConvertReturnProxy(PassThroughProxy):
    def __init__(self, fn_name, obj):
        PassThroughProxy.__init__(self, fn_name, obj)
        self.unit = obj.unit

    def __call__(self, *args):
        ret = PassThroughProxy.__call__(self, *args)
        if (type(ret) == type(NotImplemented)):
return NotImplemented
return TaggedValue(ret, self.unit)

class ConvertAllProxy(PassThroughProxy):
    def __init__(self, fn_name, obj):
        PassThroughProxy.__init__(self, fn_name, obj)
        self.unit = obj.unit

    def __call__(self, *args):
        converted_args = []
        arg_units = [self.unit]
        for a in args:
            if hasattr(a, 'get_unit') and not hasattr(a, 'convert_to'):
                # if this arg has a unit type but no conversion ability,
                # this operation is prohibited
                return NotImplemented
            if hasattr(a, 'convert_to'):
                try:
                    a = a.convert_to(self.unit)
                except:
                    pass
                arg_units.append(a.get_unit())
                converted_args.append(a.get_value())
            else:
                converted_args.append(a)
                if hasattr(a, 'get_unit'):
                    arg_units.append(a.get_unit())
                else:
                    arg_units.append(None)
        converted_args = tuple(converted_args)
        ret = PassThroughProxy.__call__(self, *converted_args)
        if (type(ret) == type(NotImplemented)):
            return NotImplemented
        ret_unit = unit_resolver(self.fn_name, arg_units)
        if (ret_unit == NotImplemented):
            return NotImplemented
        return TaggedValue(ret, ret_unit)

class _TaggedValue(object):
    __proxies = {'__add__': ConvertAllProxy,
                 '__sub__': ConvertAllProxy,
                 '__mul__': ConvertAllProxy,
                 '__rmul__': ConvertAllProxy,
                 '__cmp__': ConvertAllProxy,
                 '__lt__': ConvertAllProxy,
                 '__gt__': ConvertAllProxy,
                 '__len__': PassThroughProxy}

    def __new__(cls, value, unit):
        ...
# generate a new subclass for value
value_class = type(value)

try:
    subcls = type('TaggedValue_of_%s' % (value_class.__name__),
                 tuple([cls, value_class]),
                 {})
    if subcls not in units.registry:
        units.registry[subcls] = basicConverter
    return object.__new__(subcls)
except TypeError:
    if cls not in units.registry:
        units.registry[cls] = basicConverter
    return object.__new__(cls)

def __init__(self, value, unit):
    self.value = value
    self.unit = unit
    self.proxy_target = self.value

def __getattribute__(self, name):
    if (name.startswith('_')):
        return object.__getattribute__(self, name)
    variable = object.__getattribute__(self, 'value')
    if (hasattr(variable, name) and name not in self.__class__.__dict__):
        return getattr(variable, name)
    return object.__getattribute__(self, name)

def __array__(self, t=None, context=None):
    if t is not None:
        return np.asarray(self.value).astype(t)
    else:
        return np.asarray(self.value, 'O')

def __array_wrap__(self, array, context):
    return TaggedValue(array, self.unit)

def __repr__(self):
    return 'TaggedValue(' + repr(self.value) + ', ' + repr(self.unit) + ')

def __str__(self):
    return str(self.value) + ' in ' + str(self.unit)

def __len__(self):
    return len(self.value)

def __iter__(self):
    class IteratorProxy(object):
        def __init__(self, iter, unit):
            self.iter = iter
            self.unit = unit

        def __next__(self):
            value = next(self.iter)
```python
    return TaggedValue(value, self.unit)
    next = __next__  # for Python 2
    return IteratorProxy(iter(self.value), self.unit)

def get_compressed_copy(self, mask):
    new_value = np.ma.masked_array(self.value, mask=mask).compressed()
    return TaggedValue(new_value, self.unit)

def convert_to(self, unit):
    if (unit == self.unit or not unit):
        return self
    new_value = self.unit.convert_value_to(self.value, unit)
    return TaggedValue(new_value, unit)

def get_value(self):
    return self.value

def get_unit(self):
    return self.unit

TaggedValue = TaggedValueMeta('TaggedValue', (_TaggedValue, ), {})

class BasicUnit(object):
    def __init__(self, name, fullname=None):
        self.name = name
        if fullname is None:
            fullname = name
        self.fullname = fullname
        self.conversions = dict()

    def __repr__(self):
        return 'BasicUnit(%s)' % self.name

    def __str__(self):
        return self.fullname

    def __call__(self, value):
        return TaggedValue(value, self)

    def __mul__(self, rhs):
        value = rhs
        unit = self
        if hasattr(rhs, 'get_unit'):
            value = rhs.get_value()
            unit = rhs.get_unit()
        unit = unit_resolver('__mul__', (self, unit))
        if (unit == NotImplemented):
            return NotImplemented
        return TaggedValue(value, unit)

    def __rmul__(self, lhs):
```

111.5. units example code: basic_units.py 2919
def __array_wrap__(self, array, context):
    return TaggedValue(array, self)

def __array__(self, t=None, context=None):
    ret = np.array([1])
    if t is not None:
        return ret.astype(t)
    else:
        return ret

def add_conversion_factor(self, unit, factor):
    def convert(x):
        return x*factor
    self.conversions[unit] = convert

def add_conversion_fn(self, unit, fn):
    self.conversions[unit] = fn

def get_conversion_fn(self, unit):
    return self.conversions[unit]

def convert_value_to(self, value, unit):
    conversion_fn = self.conversions[unit]
    ret = conversion_fn(value)
    return ret

def get_unit(self):
    return self

class UnitResolver(object):
    def addition_rule(self, units):
        for unit_1, unit_2 in zip(units[:-1], units[1:]):
            if (unit_1 != unit_2):
                return NotImplemented
        return units[0]

    def multiplication_rule(self, units):
        non_null = [u for u in units if u]
        if (len(non_null) > 1):
            return NotImplemented
        return non_null[0]

    op_dict = {
        '__mul__': multiplication_rule,
        '__rmul__': multiplication_rule,
        '__add__': addition_rule,
        '__radd__': addition_rule,
        '__sub__': addition_rule,
        '__rsub__': addition_rule
    }
def __call__(self, operation, units):
    if (operation not in self.op_dict):
        return NotImplemented

    return self.op_dict[operation](self, units)

unit_resolver = UnitResolver()

cm = BasicUnit('cm', 'centimeters')
inch = BasicUnit('inch', 'inches')
inch.add_conversion_factor(cm, 2.54)
cm.add_conversion_factor(inch, 1/2.54)

radians = BasicUnit('rad', 'radians')
degrees = BasicUnit('deg', 'degrees')
radians.add_conversion_factor(degrees, 180.0/np.pi)
degrees.add_conversion_factor(radians, np.pi/180.0)

secs = BasicUnit('s', 'seconds')
hertz = BasicUnit('Hz', 'Hertz')
minutes = BasicUnit('min', 'minutes')

secs.add_conversion_fn(hertz, lambda x: 1./x)
secs.add_conversion_factor(minutes, 1/60.0)

# radians formatting

def rad_fn(x, pos=None):
    n = int((x / np.pi) * 2.0 + 0.25)
    if n == 0:
        return '0'
    elif n == 1:
        return r'$\pi/2$'
    elif n == 2:
        return r'$\pi$
    elif n % 2 == 0:
        return r'$%s\pi$' % (n//2,)
    else:
        return r'$%s\pi/2$' % (n,)

class BasicUnitConverter(units.ConversionInterface):
    @staticmethod
    def axisinfo(unit, axis):
        'return AxisInfo instance for x and unit'

        if unit == radians:
            return units.AxisInfo(
                majloc=ticker.MultipleLocator(base=np.pi/2),
                majfmt=ticker.FuncFormatter(rad_fn),
                label=unitfullname,
            )
elif unit == degrees:
    return units.AxisInfo(
        majloc=ticker.AutoLocator(),
        majfmt=ticker.FormatStrFormatter(r'${%i}^\circ$'),
        label=unit.fullname,
    )
elif unit is not None:
    if hasattr(unit, 'fullname'):
        return units.AxisInfo(label=unit.fullname)
    elif hasattr(unit, 'unit'):
        return units.AxisInfo(label=unit.unit.fullname)
return None

@staticmethod
def convert(val, unit, axis):
    if units.ConversionInterface.is_numlike(val):
        return val
    if iterable(val):
        return [thisval.convert_to(unit).get_value() for thisval in val]
    else:
        return val.convert_to(unit).get_value()

@staticmethod
def default_units(x, axis):
    'return the default unit for x or None'
    if iterable(x):
        for thisx in x:
            return thisx.unit
    return x.unit

def cos(x):
    if iterable(x):
        return [math.cos(val.convert_to(radians).get_value()) for val in x]
    else:
        return math.cos(x.convert_to(radians).get_value())

basicConverter = BasicUnitConverter()
units.registry[BasicUnit] = basicConverter
units.registry[TaggedValue] = basicConverter

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
111.6 units example code: ellipse_with_units.py

![Diagram of an ellipse with units example code](image-url)
Compare the ellipse generated with arcs versus a polygonal approximation

```python
from basic_units import cm
import numpy as np
from matplotlib import patches
import matplotlib.pyplot as plt

xcenter, ycenter = 0.38*cm, 0.52*cm
width, height = 1e-1*cm, 3e-1*cm
angle = -30

theta = np.arange(0.0, 360.0, 1.0)*np.pi/180.0
x = 0.5 * width * np.cos(theta)
y = 0.5 * height * np.sin(theta)

rtheta = np.radians(angle)
R = np.array([  
    [np.cos(rtheta), -np.sin(rtheta)],
    [np.sin(rtheta), np.cos(rtheta)],
])

x, y = np.dot(R, np.array([x, y]))
```
x += xcenter
y += ycenter

fig = plt.figure()
ax = fig.add_subplot(211, aspect='auto')
ax.fill(x, y, alpha=0.2, facecolor='yellow', edgecolor='yellow', linewidth=1, zorder=1)
e1 = patches.Ellipse((xcenter, ycenter), width, height,
    angle=angle, linewidth=2, fill=False, zorder=2)
ax.add_patch(e1)

ax = fig.add_subplot(212, aspect='equal')
ax.fill(x, y, alpha=0.2, facecolor='green', edgecolor='green', zorder=1)
e2 = patches.Ellipse((xcenter, ycenter), width, height,
    angle=angle, linewidth=2, fill=False, zorder=2)
ax.add_patch(e2)

#fig.savefig('ellipse_compare.png')
fig.savefig('ellipse_compare')

fig = plt.figure()
ax = fig.add_subplot(211, aspect='auto')
ax.fill(x, y, alpha=0.2, facecolor='yellow', edgecolor='yellow', linewidth=1, zorder=1)
e1 = patches.Arc((xcenter, ycenter), width, height,
    angle=angle, linewidth=2, fill=False, zorder=2)
ax.add_patch(e1)

ax = fig.add_subplot(212, aspect='equal')
ax.fill(x, y, alpha=0.2, facecolor='green', edgecolor='green', zorder=1)
e2 = patches.Arc((xcenter, ycenter), width, height,
    angle=angle, linewidth=2, fill=False, zorder=2)
ax.add_patch(e2)

#fig.savefig('arc_compare.png')
fig.savefig('arc_compare')

plt.show()
```python
from matplotlib.cbook import iterable
import matplotlib.units as units
import matplotlib.ticker as ticker
import matplotlib.pyplot as plt

class Foo(object):
    def __init__(self, val, unit=1.0):
        self.unit = unit
        self._val = val * unit
```
```python
def value(self, unit):
    if unit is None:
        unit = self.unit
    return self._val / unit

class FooConverter(object):
    @staticmethod
    def axisinfo(unit, axis):
        'return the Foo AxisInfo'
        if unit == 1.0 or unit == 2.0:
            return units.AxisInfo(
                majloc=ticker.IndexLocator(8, 0),
                majfmt=ticker.FormatStrFormatter("VAL: %s"),
                label='foo',
            )
        else:
            return None

    @staticmethod
    def convert(obj, unit, axis):
        
        convert obj using unit. If obj is a sequence, return the
        converted sequence
        
        if units.ConversionInterface.is_numlike(obj):
            return obj
        if iterable(obj):
            return [o.value(unit) for o in obj]
        else:
            return obj.value(unit)

    @staticmethod
    def default_units(x, axis):
        'return the default unit for x or None'
        if iterable(x):
            for thisx in x:
                return thisx.unit
        else:
            return x.unit

units.registry[Foo] = FooConverter()

# create some Foos
x = []
for val in range(0, 50, 2):
    x.append(Foo(val, 1.0))

# and some arbitrary y data
y = [i for i in range(len(x))]
```

111.7. units example code: evans_test.py
# plot specifying units
fig = plt.figure()
fig.suptitle("Custom units")
fig.subplots_adjust(bottom=0.2)
ax = fig.add_subplot(1, 2, 2)
ax.plot(x, y, 'o', xunits=2.0)
for label in ax.get_xticklabels():
    label.set_rotation(30)
    label.set_ha('right')
ax.set_title("xunits = 2.0")

# plot without specifying units; will use the None branch for axisinfo
ax = fig.add_subplot(1, 2, 1)
ax.plot(x, y)  # uses default units
ax.set_title("default units")
for label in ax.get_xticklabels():
    label.set_rotation(30)
    label.set_ha('right')

plt.show()
Radian ticks

Plot with radians from the basic_units mockup example package.

This example shows how the unit class can determine the tick locating, formatting and axis labeling.

```python
import numpy as np
from basic_units import radians, degrees, cos
from matplotlib.pyplot import figure, show

x = [val*radians for val in np.arange(0, 15, 0.01)]

fig = figure()
fig.subplots_adjust(hspace=0.3)

ax = fig.add_subplot(211)
line1, = ax.plot(x, cos(x), xunits=radians)
```

111.8 units example code: radian_demo.py
ax = fig.add_subplot(212)
line2, = ax.plot(x, cos(x), xunits=degrees)

show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

111.9 units example code: units_sample.py

```
Inches and Centimeters

The example illustrates the ability to override default x and y units (ax1) to inches and centimeters using the `xunits` and `yunits` parameters for the `plot` function. Note that conversions are applied to get numbers to correct units.
```

```
from basic_units import cm, inch
import matplotlib.pyplot as plt
```
import numpy

cms = cm * numpy.arange(0, 10, 2)

fig = plt.figure()

ax1 = fig.add_subplot(2, 2, 1)
ax1.plot(cms, cms)

ax2 = fig.add_subplot(2, 2, 2)
ax2.plot(cms, cms, xunits=cm, yunits=inch)

ax3 = fig.add_subplot(2, 2, 3)
ax3.plot(cms, cms, xunits=inch, yunits=cm)
ax3.set_xlim(3, 6)  # scalars are interpreted in current units

ax4 = fig.add_subplot(2, 2, 4)
ax4.plot(cms, cms, xunits=inch, yunits=inch)
ax4.set_xlim(3*cm, 6*cm)  # cm are converted to inches

plt.show()
Unit handling

basic_units is a mockup of a true units package used for testing purposes, which illustrates the basic interface that a units package must provide to matplotlib.

The example below shows support for unit conversions over masked arrays.

```python
import numpy as np
import matplotlib.pyplot as plt
from basic_units import secs, hertz, minutes

# create masked array
data = (1, 2, 3, 4, 5, 6, 7, 8)
mask = (1, 0, 1, 0, 0, 0, 1, 0)
xsecs = secs * np.ma.MaskedArray(data, mask, float)

fig, (ax1, ax2, ax3) = plt.subplots(nrows=3, sharex=True)
```

111.10 units example code: units_scatter.py
ax1.scatter(xsecs, xsecs)
a1.yaxis.set_units(secs)
a1.axis([0, 10, 0, 10])

ax2.scatter(xsecs, xsecs, yunits=hertz)
a2.axis([0, 10, 0, 1])

ax3.scatter(xsecs, xsecs, yunits=hertz)
a3.axis([0, 10, 0, 1])

fig.tight_layout()
plt.show()
112.1 user_interfaces example code: embedding_in_gtk.py

```python
import gtk
from matplotlib.figure import Figure
from numpy import arange, sin, pi

# uncomment to select /GTK/GTKAgg/GTKCairo
# from matplotlib.backends.backend_gtk import FigureCanvasGTK as FigureCanvas
from matplotlib.backends.backend_gtkagg import FigureCanvasGTKAgg as FigureCanvas
# from matplotlib.backends.backend_gtkcairo import FigureCanvasGTKCairo as FigureCanvas

win = gtk.Window()
win.connect("destroy", lambda x: gtk.main_quit())
win.set_default_size(400, 300)
win.set_title("Embedding in GTK")

f = Figure(figsize=(5, 4), dpi=100)
a = f.add_subplot(111)
t = arange(0.0, 3.0, 0.01)
s = sin(2*pi*t)
a.plot(t, s)

canvas = FigureCanvas(f)  # a gtk.DrawingArea
win.add(canvas)

win.show_all()
gtk.main()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
112.2 user_interfaces example code: embedding_in_gtk2.py

[source code]

```python
""
show how to add a matplotlib FigureCanvasGTK or FigureCanvasGTKAgg widget and
a toolbar to a gtk.Window
""

import gtk

from matplotlib.figure import Figure
from numpy import arange, sin, pi

# uncomment to select /GTK/GTKAgg/GTKcairo
#from matplotlib.backends.backend_gtk import FigureCanvasGTK as FigureCanvas
from matplotlib.backends.backend_gtkagg import FigureCanvasGTKAgg as FigureCanvas
#from matplotlib.backends.backend_gtkcairo import FigureCanvasGTKCairo as FigureCanvas

# or NavigationToolbar for classic
#from matplotlib.backends.backend_gtk import NavigationToolbar2GTK as NavigationToolbar
from matplotlib.backends.backend_gtkagg import NavigationToolbar2GTKAgg as NavigationToolbar

# implement the default mpl key bindings
from matplotlib.backend_bases import key_press_handler

win = gtk.Window()
win.connect("destroy", lambda x: gtk.main_quit())
win.set_default_size(400, 300)
win.set_title("Embedding in GTK")

vbox = gtk.VBox()
win.add(vbox)

fig = Figure(figsize=(5, 4), dpi=100)
ax = fig.add_subplot(111)
t = arange(0.0, 3.0, 0.01)
s = sin(2*pi*t)

ax.plot(t, s)

canvas = FigureCanvas(fig)  # a gtk.DrawingArea
vbox.pack_start(canvas)
toolbar = NavigationToolbar(canvas, win)
vbox.pack_start(toolbar, False, False)

def on_key_event(event):
    print('you pressed %s' % event.key)
    key_press_handler(event, canvas, toolbar)

canvas.mpl_connect('key_press_event', on_key_event)
```
```python
import gi
from gi.repository import Gtk
from matplotlib.figure import Figure
from numpy import arange, sin, pi
from matplotlib.backends.backend_gtk3agg import FigureCanvasGTK3Agg as FigureCanvas

win = Gtk.Window()
win.connect("delete-event", Gtk.main_quit)
win.set_default_size(400, 300)
win.set_title("Embedding in GTK")

f = Figure(figsize=(5, 4), dpi=100)
a = f.add_subplot(111)
t = arange(0.0, 3.0, 0.01)
s = sin(2*pi*t)
a.plot(t, s)

sw = Gtk.ScrolledWindow()
win.add(sw)
# A scrolled window border goes outside the scrollbars and viewport
sw.set_border_width(10)

canvas = FigureCanvas(f)  # a Gtk.DrawingArea
canvas.set_size_request(800, 600)
sw.add_with_viewport(canvas)

win.show_all()
Gtk.main()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

### 112.4 user_interfaces example code: embedding_in_gtk3_panzoom.py

[source code]
```
# demonstrate NavigationToolbar with GTK3 accessed via pygobject
#
from gi.repository import Gtk

from matplotlib.figure import Figure
from numpy import arange, sin, pi
from matplotlib.backends.backend_gtk3agg import FigureCanvasGTK3Agg as FigureCanvas
from matplotlib.backends.backend_gtk3 import NavigationToolbar2GTK3 as NavigationToolbar

win = Gtk.Window()
win.connect("delete-event", Gtk.main_quit)
win.set_default_size(400, 300)
win.set_title("Embedding in GTK")

f = Figure(figsize=(5, 4), dpi=100)
a = f.add_subplot(1, 1, 1)
t = arange(0.0, 3.0, 0.01)
s = sin(2*pi*t)
a.plot(t, s)

vbox = Gtk.VBox()
win.add(vbox)

# Add canvas to vbox
canvas = FigureCanvas(f)  # a Gtk.DrawingArea
vbox.pack_start(canvas, True, True, 0)

# Create toolbar
toolbar = NavigationToolbar(canvas, win)
vbox.pack_start(toolbar, False, False, 0)

win.show_all()
Gtk.main()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

112.5 user_interface example code: embedding_in_qt4.py

[source code]

```
# embedding_in_qt4.py --- Simple Qt4 application embedding matplotlib canvases
#
# Copyright (C) 2005 Florent Rougon
# 2006 Darren Dale
#
# This file is an example program for matplotlib. It may be used and
# modified with no restriction; raw copies as well as modified versions
# may be distributed without limitation.
```
from __future__ import unicode_literals
import sys
import os
import random
from matplotlib.backends import qt_compat
use_pyside = qt_compat.QT_API == qt_compat.QT_API_PYSIDE
if use_pyside:
    from PySide import QtGui, QtCore
else:
    from PyQt4 import QtGui, QtCore
from numpy import arange, sin, pi
from matplotlib.backends.backend_qt4agg import FigureCanvasQTAgg as FigureCanvas
from matplotlib.figure import Figure

progname = os.path.basename(sys.argv[0])
progversion = "0.1"

class MyMplCanvas(FigureCanvas):
    """Ultimately, this is a QWidget (as well as a FigureCanvasAgg, etc.).""

    def __init__(self, parent=None, width=5, height=4, dpi=100):
        fig = Figure(figsize=(width, height), dpi=dpi)
        self.axes = fig.add_subplot(111)
        self.compute_initial_figure()

        FigureCanvas.__init__(self, fig)
        self.setParent(parent)

        FigureCanvas.setSizePolicy(self,
                                   QtGui.QSizePolicy.Expanding,
                                   QtGui.QSizePolicy.Expanding)
        FigureCanvas.updateGeometry(self)

    def compute_initial_figure(self):
        pass

class MyStaticMplCanvas(MyMplCanvas):
    """Simple canvas with a sine plot.""

    def compute_initial_figure(self):
        t = arange(0.0, 3.0, 0.01)
        s = sin(2*pi*t)
        self.axes.plot(t, s)

class MyDynamicMplCanvas(MyMplCanvas):
    """A canvas that updates itself every second with a new plot.""

    def __init__(self, *args, **kwargs):
        112.5. user_interfaces example code: embedding_in_qt4.py 2939
MyMplCanvas.__init__(self, *args, **kwargs)
timer = QtCore.QTimer(self)
timer.timeout.connect(self.update_figure)
timer.start(1000)

def compute_initial_figure(self):
    self.axes.plot([0, 1, 2, 3], [1, 2, 0, 4], 'r')

def update_figure(self):
    # Build a list of 4 random integers between 0 and 10 (both inclusive)
    l = [random.randint(0, 10) for i in range(4)]
    self.axes.cla()
    self.axes.plot([0, 1, 2, 3], l, 'r')
    self.draw()

class ApplicationWindow(QtGui.QMainWindow):
    def __init__(self):
        QtGui.QMainWindow.__init__(self)
        self.setAttribute(QtCore.Qt.WA_DeleteOnClose)
        self.setWindowTitle("application main window")

        self.file_menu = QtGui.QMenu('&File', self)
        self.file_menu.addAction('&Quit', self.fileQuit,
                                  QtCore.Qt.CTRL + QtCore.Qt.Key_Q)
        self.menuBar().addMenu(self.file_menu)

        self.help_menu = QtGui.QMenu('&Help', self)
        self.menuBar().addSeparator()
        self.menuBar().addMenu(self.help_menu)

        self.file_menu.addAction('&About', self.about)

        self.main_widget = QtGui.QWidget(self)
        l = QtGui.QVBoxLayout(self.main_widget)
        sc = MyStaticMplCanvas(self.main_widget, width=5, height=4, dpi=100)
        dc = MyDynamicMplCanvas(self.main_widget, width=5, height=4, dpi=100)
        l.addWidget(sc)
        l.addWidget(dc)
        self.main_widget.setFocus()
        self.setCentralWidget(self.main_widget)
        self.statusBar().showMessage("All hail matplotlib!", 2000)

    def fileQuit(self):
        self.close()

    def closeEvent(self, ce):
        self.fileQuit()

    def about(self):
        pass
Matplotlib, Release 2.0.2

```python
QtGui.QMessageBox.about(self, "About",
                        """"embedding_in_qt4.py example

Copyright 2005 Florent Rougon, 2006 Darren Dale

This program is a simple example of a Qt4 application embedding matplotlib canvases.

It may be used and modified with no restriction; raw copies as well as modified versions may be distributed without limitation.

```

qApp = QtGui.QApplication(sys.argv)

aw = ApplicationWindow()
aw.setWindowTitle("%s" % proiname)
aw.show()
sys.exit(qApp.exec_())

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

112.6 user_interfaces example code: embedding_in_qt4_wtoolbar.py

[source code]

```python
from __future__ import print_function
import sys
import numpy as np
from matplotlib.figure import Figure
from matplotlib.backend_bases import key_press_handler
from matplotlib.backends.backend_qt4agg import (FigureCanvasQTAgg
                                                as FigureCanvas,
                                                NavigationToolbar2QT
                                                as NavigationToolbar)
from matplotlib.backends import qt4_compat
use_pyside = qt4_compat.QT_API == qt4_compat.QT_API_PYSIDE

if use_pyside:
    from PySide.QtCore import *
    from PySide.QtGui import *
else:
    from PyQt4.QtCore import *
    from PyQt4.QtGui import *

class AppForm(QMainWindow):
    def __init__(self, parent=None):
        QMainWindow.__init__(self, parent)
        #self.x, self.y = self.get_data()
```

112.6. user_interfaces example code: embedding_in_qt4_wtoolbar.py 2941
self.data = self.get_data2()
self.create_main_frame()
self.on_draw()

def create_main_frame(self):
    self.main_frame = QWidget()
    self.fig = Figure((5.0, 4.0), dpi=100)
    self.canvas = FigureCanvas(self.fig)
    self.canvas.setParent(self.main_frame)
    self.canvas.setFocusPolicy(Qt.StrongFocus)
    self.canvas.setFocus()
    self.mpl_toolbar = NavigationToolbar(self.canvas, self.main_frame)

    self.canvas.mpl_connect('key_press_event', self.on_key_press)
    vbox = QVBoxLayout()
    vbox.addWidget(self.canvas)
    vbox.addWidget(self.mpl_toolbar)
    self.main_frame.setLayout(vbox)
    self.setCentralWidget(self.main_frame)

def get_data2(self):
    return np.arange(20).reshape([4, 5]).copy()

def on_draw(self):
    self.fig.clear()
    self.axes = self.fig.add_subplot(111)
    self.axes.imshow(self.data, interpolation='nearest')
    self.canvas.draw()

def on_key_press(self, event):
    print('you pressed', event.key)
    key_press_handler(event, self.canvas, self.mpl_toolbar)

def main():
    app = QApplication(sys.argv)
    form = AppForm()
    form.show()
    app.exec_()

if __name__ == '__main__':
    main()
112.7  user_interfaces example code: embedding_in_qt5.py

[source code]

```python
# embedding_in_qt5.py --- Simple Qt5 application embedding matplotlib canvases
#
# Copyright (C) 2005 Florent Rougon
# 2006 Darren Dale
# 2015 Jens H Nielsen
#
# This file is an example program for matplotlib. It may be used and
# modified with no restriction; raw copies as well as modified versions
# may be distributed without limitation.

from __future__ import unicode_literals
import sys
import os
import random
import matplotlib
# Make sure that we are using QT5
matplotlib.use('Qt5Agg')
from PyQt5 import QtCore, QtWidgets
from numpy import arange, sin, pi
from matplotlib.backends.backend_qt5agg import FigureCanvasQTAgg as FigureCanvas
from matplotlib.figure import Figure

progname = os.path.basename(sys.argv[0])
progversion = "0.1"

class MyMplCanvas(FigureCanvas):
    """Ultimately, this is a QWidget (as well as a FigureCanvasAgg, etc.)."""

    def __init__(self, parent=None, width=5, height=4, dpi=100):
        fig = Figure(figsize=(width, height), dpi=dpi)
        self.axes = fig.add_subplot(111)

        self.compute_initial_figure()

        FigureCanvas.__init__(self, fig)
        self.setParent(parent)

        FigureCanvas.setSizePolicy(self,
                                  QtWidgets.QSizePolicy.Expanding,
                                  QtWidgets.QSizePolicy.Expanding)
        FigureCanvas.updateGeometry(self)

    def compute_initial_figure(self):
        pass

class MyStaticMplCanvas(MyMplCanvas):
```

---

112.7.  user_interfaces example code: embedding_in_qt5.py 2943
```python
"""Simple canvas with a sine plot."""

def compute_initial_figure(self):
    t = arange(0.0, 3.0, 0.01)
    s = sin(2*pi*t)
    self.axes.plot(t, s)

class MyDynamicMplCanvas(MyMplCanvas):
    """A canvas that updates itself every second with a new plot."""

def __init__(self, *args, **kwargs):
    MyMplCanvas.__init__(self, *args, **kwargs)
    timer = QtCore.QTimer(self)
    timer.timeout.connect(self.update_figure)
    timer.start(1000)

def compute_initial_figure(self):
    self.axes.plot([0, 1, 2, 3], [1, 2, 0, 4], 'r')

def update_figure(self):
    # Build a list of 4 random integers between 0 and 10 (both inclusive)
    l = [random.randint(0, 10) for i in range(4)]
    self.axes.cla()
    self.axes.plot([0, 1, 2, 3], l, 'r')
    self.draw()

class ApplicationWindow(QtWidgets.QMainWindow):
    def __init__(self):
        QtWidgets.QMainWindow.__init__(self)
        self.setAttribute(QtCore.Qt.WA_DeleteOnClose)
        self.setWindowTitle("application main window")

        self.file_menu = QtWidgets.QMenu(&quot;File&quot;, self)
        self.file_menu.addAction(&quot;Quit&quot;, self.fileQuit, QtCore.Qt.CTRL + QtCore.Qt.Key_Q)
        self.menuBar().addMenu(self.file_menu)

        self.help_menu = QtWidgets.QMenu(&quot;Help&quot;, self)
        self.menuBar().addSeparator()
        self.menuBar().addMenu(self.help_menu)

        self.help_menu.addAction(&quot;About&quot;, self.about)

        self.main_widget = QtWidgets.QWidget(self)
        l = QtWidgets.QVBoxLayout(self.main_widget)
        sc = MyStaticMplCanvas(self.main_widget, width=5, height=4, dpi=100)
        dc = MyDynamicMplCanvas(self.main_widget, width=5, height=4, dpi=100)
        l.addWidget(sc)
        l.addWidget(dc)
```

2944 Chapter 112. user_interfaces Examples
```python
self.main_widget.setFocus()
self.setCentralWidget(self.main_widget)

self.statusBar().showMessage("All hail matplotlib!", 2000)

def fileQuit(self):
    self.close()

def closeEvent(self, ce):
    self.fileQuit()

def about(self):
    QtWidgets.QMessageBox.about(self, "About",
        "embedding_in_qt5.py example
        """embedding_in_qt5.py example

This program is a simple example of a Qt5 application embedding matplotlib canvases.

It may be used and modified with no restriction; raw copies as well as modified versions may be distributed without limitation.

This is modified from the embedding in qt4 example to show the difference between qt4 and qt5"
"
)

qApp = QtWidgets.QApplication(sys.argv)
aw = ApplicationWindow()
aw.setWindowTitle("%s" % proname)
aw.show()
sys.exit(qApp.exec_())
#qApp.exec_()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

### 112.8 user_interfaces example code: embedding_in_tk.py

[source code]

```python
import matplotlib
matplotlib.use('TkAgg')

from numpy import arange, sin, pi
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg, NavigationToolbar2TkAgg
# implement the default mpl key bindings
from matplotlib.backend_bases import key_press_handler

from matplotlib.figure import Figure
```
import sys
if sys.version_info[0] < 3:
    import Tkinter as Tk
else:
    import tkinter as Tk

root = Tk.Tk()
root.wm_title("Embedding in Tk")

f = Figure(figsize=(5, 4), dpi=100)
a = f.add_subplot(111)
t = arange(0.0, 3.0, 0.01)
s = sin(2*pi*t)

a.plot(t, s)

# a Tk.DrawingArea
canvas = FigureCanvasTkAgg(f, master=root)
canvas.show()
canvas.get_tk_widget().pack(side=Tk.TOP, fill=Tk.BOTH, expand=1)

toolbar = NavigationToolbar2TkAgg(canvas, root)
toolbar.update()
canvas._tkcanvas.pack(side=Tk.TOP, fill=Tk.BOTH, expand=1)

def on_key_event(event):
    print('you pressed %s' % event.key)
    key_press_handler(event, canvas, toolbar)

canvas.mpl_connect('key_press_event', on_key_event)

def _quit():
    root.quit()  # stops mainloop
    root.destroy()  # this is necessary on Windows to prevent
                   # Fatal Python Error: PyEval_RestoreThread: NULL tstate

button = Tk.Button(master=root, text='Quit', command=_quit)
button.pack(side=Tk.BOTTOM)

Tk.mainloop()
# If you put root.destroy() here, it will cause an error if
# the window is closed with the window manager.

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
import matplotlib
matplotlib.use('TkAgg')

from numpy import arange, sin, pi
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
from matplotlib.figure import Figure

import sys
if sys.version_info[0] < 3:
    import Tkinter as Tk
else:
    import tkinter as Tk

def destroy(e):
    sys.exit()

root = Tk.Tk()
root.wm_title("Embedding in TK")

f = Figure(figsize=(5, 4), dpi=100)
a = f.add_subplot(111)
t = arange(0.0, 3.0, 0.01)
s = sin(2*pi*t)
a.plot(t, s)
a.set_title('Tk embedding')
a.set_xlabel('X axis label')
a.set_ylabel('Y label')

# a tk.DrawingArea
canvas = FigureCanvasTkAgg(f, master=root)
canvas.show()
canvas.get_tk_widget().pack(side=Tk.TOP, fill=Tk.BOTH, expand=1)

canvas._tkcanvas.pack(side=Tk.TOP, fill=Tk.BOTH, expand=1)

button = Tk.Button(master=root, text='Quit', command=sys.exit)
button.pack(side=Tk.BOTTOM)

Tk.mainloop()
112.10 user_interfaces example code: embedding_in_tk_canvas.py

[source code]

```python
# -*- noplot -*-

import matplotlib as mpl
import numpy as np
import sys
if sys.version_info[0] < 3:
    import Tkinter as tk
else:
    import tkinter as tk
import matplotlib.backends.tkagg as tkagg
from matplotlib.backends.backend_agg import FigureCanvasAgg

def draw_figure(canvas, figure, loc=(0, 0)):
    """ Draw a matplotlib figure onto a Tk canvas

    loc: location of top-left corner of figure on canvas in pixels.
    
    Inspired by matplotlib source: lib/matplotlib/backends/backend_tkagg.py
    ""
    figure_canvas_agg = FigureCanvasAgg(figure)
    figure_canvas_agg.draw()
    figure_x, figure_y, figure_w, figure_h = figure.bbox.bounds
    figure_w, figure_h = int(figure_w), int(figure_h)
    photo = tk.PhotoImage(master=canvas, width=figure_w, height=figure_h)
    # Position: convert from top-left anchor to center anchor
    canvas.create_image(loc[0] + figure_w/2, loc[1] + figure_h/2, image=photo)

    # Unfortunately, there's no accessor for the pointer to the native renderer
    tkagg.blit(photo, figure_canvas_agg.get_renderer()._renderer, colormode=2)

    # Return a handle which contains a reference to the photo object
    # which must be kept live or else the picture disappears
    return photo

# Create a canvas
w, h = 300, 200
window = tk.Tk()
window.title("A figure in a canvas")
canvas = tk.Canvas(window, width=w, height=h)
canvas.pack()

# Generate some example data
X = np.linspace(0, 2.*np.pi, 50)
Y = np.sin(X)

# Create the figure we desire to add to an existing canvas
fig = mpl.figure.Figure(figsize=(2, 1))
```

Chapter 112. user_interfaces Examples
ax = fig.add_axes([0, 0, 1, 1])
ax.plot(X, Y)

# Keep this handle alive, or else figure will disappear
fig_x, fig_y = 100, 100
fig_photo = draw_figure(canvas, fig, loc=(fig_x, fig_y))
fig_w, fig_h = fig_photo.width(), fig_photo.height()

# Add more elements to the canvas, potentially on top of the figure
canvas.create_line(200, 50, fig_x + fig_w / 2, fig_y + fig_h / 2)
canvas.create_text(200, 50, text="Zero-crossing", anchor="s")

# Let Tk take over
tk.mainloop()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

112.11 user_interfaces example code: embedding_in_wx2.py

[source code]

"""
An example of how to use wx or wxagg in an application with the new
toolbar - comment out the setA_toolbar line for no toolbar
"""

# matplotlib requires wxPython 2.8+
# set the wxPython version in lib\site-packages\wx.pth file
# or if you have wxversion installed un-comment the lines below
import matplotlib
#matplotlib.use('WX')
from numpy import arange, sin, pi

import matplotlib

# uncomment the following to use wx rather than wxagg
#matplotlib.use('WX')
# from matplotlib.backends.backend_wx import FigureCanvasWx as FigureCanvas

# comment out the following to use wx rather than wxagg
matplotlib.use('WXAgg')
from matplotlib.backends.backend_wxagg import FigureCanvasWxAgg as FigureCanvas

canvas = FigureCanvas(fig)

import wx

import wx.lib.mixins.inspection as WIT
class CanvasFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1, 'CanvasFrame', size=(550, 350))

        self.figure = Figure()
        self.axes = self.figure.add_subplot(111)
        t = arange(0.0, 3.0, 0.01)
        s = sin(2 * pi * t)

        self.axes.plot(t, s)
        self.canvas = FigureCanvas(self, -1, self.figure)

        self.sizer = wx.BoxSizer(wx.VERTICAL)
        self.sizer.Add(self.canvas, 1, wx.LEFT | wx.TOP | wx.EXPAND)
        self.SetSizer(self.sizer)
        self.Fit()

        self.add_toolbar()  # comment this out for no toolbar

    def add_toolbar(self):
        self.toolbar = NavigationToolbar2Wx(self.canvas)
        self.toolbar.Realize()
        # By adding toolbar in sizer, we are able to put it at the bottom
        # of the frame - so appearance is closer to GTK version.
        self.sizer.Add(self.toolbar, 0, wx.LEFT | wx.TOP | wx.EXPAND)
        # update the axes menu on the toolbar
        self.toolbar.update()

# alternatively you could use
# class App(wx.App):
class App(WIT.InspectableApp):
    def OnInit(self):
        'Create the main window and insert the custom frame'
        self.Init()
        frame = CanvasFrame()
        frame.Show(True)

        return True

app = App(0)
app.MainLoop()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

112.12 user_interfaces example code: embedding_in_wx3.py

[source code]
This is yet another example of using matplotlib with wx. Hopefully this is pretty full-featured:

- both matplotlib toolbar and WX buttons manipulate plot
- full wxApp framework, including widget interaction
- XRC (XML wxWidgets resource) file to create GUI (made with XRced)

This was derived from embedding_in_wx and dynamic_image_wxagg.

Thanks to matplotlib and wx teams for creating such great software!

```
from __future__ import print_function

# matplotlib requires wxPython 2.8+
# set the wxPython version in lib\site-packages\wx.pth file
# or if you have wxversion installed un-comment the lines below
#import wxversion
#wxversion.ensureMinimal('2.8')

import sys
import time
import os
import gc
import matplotlib
matplotlib.use('WXAgg')
import matplotlib.cm as cm
import matplotlib.cbook as cbook
from matplotlib.backends.backend_wxagg import Toolbar, FigureCanvasWxAgg
from matplotlib.figure import Figure
import numpy as np

import wx
import wx.xrc as xrc

ERR_TOL = 1e-5  # floating point slop for peak-detection

matplotlib.rc('image', origin='lower')

class PlotPanel(wx.Panel):
    def __init__(self, parent):
        wx.Panel.__init__(self, parent, -1)
        self.fig = Figure((5, 4), 75)
self.canvas = FigureCanvasWxAgg(self, -1, self.fig)
self.toolbar = Toolbar(self.canvas)  # matplotlib toolbar
self.toolbar.Realize()
# self.toolbar.set_active([0,1])
# Now put all into a sizer
sizer = wx.BoxSizer(wx.VERTICAL)
# This way of adding to sizer allows resizing
sizer.Add(self.canvas, 1, wx.LEFT | wx.TOP | wx.GROW)
# Best to allow the toolbar to resize!
sizer.Add(self.toolbar, 0, wx.GROW)
self.SetSizer(sizer)
self.Fit()

def init_plot_data(self):
a = self.fig.add_subplot(111)

x = np.arange(120.0) * 2 * np.pi / 60.0
y = np.arange(100.0) * 2 * np.pi / 50.0
self.x, self.y = np.meshgrid(x, y)
z = np.sin(self.x) + np.cos(self.y)
self.im = a.imshow(z, cmap=cm.RdBu, interpolation='nearest')

zmax = np.amax(z) - ERR_TOL
ymax_i, xmax_i = np.nonzero(z >= zmax)
if self.im.origin == 'upper':
    ymax_i = z.shape[0] - ymax_i
self.lines = a.plot(xmax_i, ymax_i, 'ko')

self.toolbar.update()  # Not sure why this is needed - ADS

def GetToolBar(self):
    # You will need to override GetToolBar if you are using an
    # unmanaged toolbar in your frame
    return self.toolbar

def OnWhiz(self, evt):
    self.x += np.pi / 15
    self.y += np.pi / 20
    z = np.sin(self.x) + np.cos(self.y)
    self.im.set_array(z)
    zmax = np.amax(z) - ERR_TOL
    ymax_i, xmax_i = np.nonzero(z >= zmax)
    if self.im.origin == 'upper':
        ymax_i = z.shape[0] - ymax_i
    self.lines[0].set_data(xmax_i, ymax_i)
    self.canvas.draw()

def onEraseBackground(self, evt):
    # this is supposed to prevent redraw flicker on some X servers...
    pass
class MyApp(wx.App):
    def OnInit(self):
        xrcfile = cbook.get_sample_data('embedding_in_wx3.xrc',
                                         asfileobj=False)

        print('loading', xrcfile)

        self.res = xrc.XmlResource(xrcfile)

        # main frame and panel ---------
        self.frame = self.res.LoadFrame(None, "MainFrame")
        self.panel = xrc.XRCCTRL(self.frame, "MainPanel")

        # matplotlib panel ------------
        # container for matplotlib panel (I like to make a container
        # panel for our panel so I know where it'll go when in XRCed.)
        plot_container = xrc.XRCCTRL(self.frame, "plot_container_panel")
        sizer = wx.BoxSizer(wx.VERTICAL)

        # matplotlib panel itself
        self.plotpanel = PlotPanel(plot_container)
        self.plotpanel.init_plot_data()

        # wx boilerplate
        sizer.Add(self.plotpanel, 1, wx.EXPAND)
        plot_container.SetSizer(sizer)

        # whiz button -----------------
        whiz_button = xrc.XRCCTRL(self.frame, "whiz_button")
        whiz_button.Bind(wx.EVT_BUTTON, self.plotpanel.OnWhiz)

        # bang button ------------------
        bang_button = xrc.XRCCTRL(self.frame, "bang_button")
        bang_button.Bind(wx.EVT_BUTTON, self.OnBang)

        # final setup -----------------
        sizer = self.panel.GetSizer()
        self.frame.Show(1)

        self.SetTopWindow(self.frame)

        return True

    def OnBang(self, event):
        bang_count = xrc.XRCCTRL(self.frame, "bang_count")
        bangs = int(bangs) + 1
        bang_count.SetValue(str(bangs))

if __name__ == '__main__':
```python
app = MyApp(0)
app.MainLoop()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

### 112.13 user_interfaces example code: embedding_in_wx4.py

[Source code]

```python
""
An example of how to use wx or wxagg in an application with a custom toolbar
""

# matplotlib requires wxPython 2.8+
# set the wxPython version in lib\site-packages\wx.pth file
# or if you have wxversion installed un-comment the lines below
# import wxversion
# wxversion.ensureMinimal(‘2.8’)

from numpy import arange, sin, pi

import matplotlib

matplotlib.use(‘WXAgg’)

from matplotlib.backends.backend_wxagg import FigureCanvasWxAgg as FigureCanvas

from matplotlib.backends.backend_wxagg import NavigationToolbar2WxAgg

from matplotlib.backends.backend_wx import _load_bitmap

from matplotlib.figure import Figure

from numpy.random import rand

import wx

class MyNavigationToolbar(NavigationToolbar2WxAgg):
    ""
    Extend the default wx toolbar with your own event handlers
    ""
    ON_CUSTOM = wx.NewId()

    def __init__(self, canvas, cankill):
        NavigationToolbar2WxAgg.__init__(self, canvas)

        # for simplicity I'm going to reuse a bitmap from wx, you'll probably want to add your own.
        if 'phoenix' in wx.PlatformInfo:
            self.AddTool(self.ON_CUSTOM, 'Click me',
                        _load_bitmap('back.png'),
                        'Activate custom control')
            self.Bind(wx.EVT_TOOL, self._on_custom, id=self.ON_CUSTOM)
```

2954 Chapter 112. user_interfaces Examples
else:
    self.AddSimpleTool(self.ON_CUSTOM, _load_bitmap('back.png'),
                       'Click me', 'Activate custom control')
    self.Bind(wx.EVT_TOOL, self._on_custom, id=self.ON_CUSTOM)

def _on_custom(self, evt):
    # add some text to the axes in a random location in axes (0,1) coords with a random color

    # get the axes
    ax = self.canvas.figure.axes[0]

    # generate a random location in color
    x, y = tuple(rand(2))
    rgb = tuple(rand(3))

    # add the text and draw
    ax.text(x, y, 'You clicked me',
            transform=ax.transAxes,
            color=rgb)
    self.canvas.draw()
    evt.Skip()

class CanvasFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1, 'CanvasFrame', size=(550, 350))

        self.figure = Figure(figsize=(5, 4), dpi=100)
        self.axes = self.figure.add_subplot(111)
        t = arange(0.0, 3.0, 0.01)
        s = sin(2 * pi * t)

        self.axes.plot(t, s)

        self.canvas = FigureCanvas(self, -1, self.figure)

        self.sizer = wx.BoxSizer(wx.VERTICAL)
        self.sizer.Add(self.canvas, 1, wx.TOP | wx.LEFT | wx.EXPAND)

        # Capture the paint message
        self.Bind(wx.EVT_PAINT, self.OnPaint)

        self.toolbar = MyNavigationToolbar(self.canvas, True)
        self.toolbar.Realize()

        self.sizer.Add(self.toolbar, 0, wx.LEFT | wx.EXPAND)

        self.SetSizer(self.sizer)
        self.Fit()
```python
def OnPaint(self, event):
    self.canvas.draw()
    event.Skip()

class App(wx.App):
    def OnInit(self):
        # Create the main window and insert the custom frame
        frame = CanvasFrame()
        frame.Show(True)
        return True

app = App(0)
app.MainLoop()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

### 112.14 user_interfaces example code: embedding_in_wx5.py

[source code]

```python
# matplotlib requires wxPython 2.8+
# set the wxPython version in lib\site-packages\wx.pth file
# or if you have wxversion installed un-comment the lines below
# import wxversion
# wxversion.ensureMinimal('2.8')

import wx
import wx.lib.mixins.inspection as wit

if 'phoenix' in wx.PlatformInfo:
    import wx.lib.agw.aui as aui
else:
    import wx.aui as aui

import matplotlib as mpl
from matplotlib.backends.backend_wxagg import FigureCanvasWxAgg as Canvas
from matplotlib.backends.backend_wxagg import NavigationToolbar2Wx as Toolbar

class Plot(wx.Panel):
    def __init__(self, parent, id=-1, dpi=None, **kwargs):
        wx.Panel.__init__(self, parent, id=id, **kwargs)
        self.figure = mpl.figure.Figure(dpi=dpi, figsize=(2, 2))
        self.canvas = Canvas(self, -1, self.figure)
        self.toolbar = Toolbar(self.canvas)
        self.toolbar.Realize()

        sizer = wx.BoxSizer(wx.VERTICAL)
```
```python
sizer.Add(self.canvas, 1, wx.EXPAND)
sizer.Add(self.toolbar, 0, wx.LEFT | wx.EXPAND)
self.SetSizer(sizer)

class PlotNotebook(wx.Panel):
    def __init__(self, parent, id=-1):
        wx.Panel.__init__(self, parent, id=id)
        self.nb = aui.AuiNotebook(self)
        sizer = wx.BoxSizer()
        sizer.Add(self.nb, 1, wx.EXPAND)
        self.SetSizer(sizer)

    def add(self, name="plot"):
        page = Plot(self.nb)
        self.nb.AddPage(page, name)
        return page.figure

def demo():
    # alternatively you could use
    # app = wx.App()
    # InspectableApp is a great debug tool, see:
    # http://wiki.wxpython.org/Widget%20Inspection%20Tool
    app = wit.InspectableApp()
    frame = wx.Frame(None, -1, 'Plotter')
    plotter = PlotNotebook(frame)
    axes1 = plotter.add('figure 1').gca()
    axes1.plot([1, 2, 3], [2, 1, 4])
    axes2 = plotter.add('figure 2').gca()
    axes2.plot([1, 2, 3, 4, 5], [2, 1, 4, 2, 3])
    frame.Show()
    app.MainLoop()

if __name__ == "__main__":
    demo()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

### 112.15 user_interfaces example code: embedding_webagg.py

[source code]

```python
#
# This example demonstrates how to embed matplotlib WebAgg interactive
# plotting in your own web application and framework. It is not
# necessary to do all this if you merely want to display a plot in a
# browser or use matplotlib's built-in Tornado-based server "on the
# side".
#
# The framework being used must support web sockets.
```
import io

try:
    import tornado
except ImportError:
    raise RuntimeError("This example requires tornado.")
import tornado.web
import tornado.httpserver
import tornado.ioloop
import tornado.websocket

from matplotlib.backends.backend_webagg_core import (FigureManagerWebAgg, new_figure_manager_given_figure)
from matplotlib.figure import Figure

import numpy as np
import json

def create_figure():
    
    Creates a simple example figure.
    
    fig = Figure()
    a = fig.add_subplot(111)
    t = np.arange(0.0, 3.0, 0.01)
    s = np.sin(2 * np.pi * t)
    a.plot(t, s)
    return fig

# The following is the content of the web page. You would normally
# generate this using some sort of template facility in your web
# framework, but here we just use Python string formatting.
html_content = ""
<html>
<head>
    <!-- TODO: There should be a way to include all of the required javascript
    and CSS so matplotlib can add to the set in the future if it
    needs to. -->
    <link rel="stylesheet" href="_static/css/page.css" type="text/css">
    <link rel="stylesheet" href="_static/css/boilerplate.css" type="text/css" />
    <link rel="stylesheet" href="_static/css/fbm.css" type="text/css" />
    <link rel="stylesheet" href="_static/jquery/css/themes/base/jquery-ui.min.css">
    <script src="_static/jquery/js/jquery-1.11.3.min.js"></script>
    <script src="_static/jquery/js/jquery-ui.min.js"></script>
    <script src="mpl.js"></script>
</head>
""
/* This is a callback that is called when the user saves
(downloads) a file. Its purpose is really to map from a
figure and file format to a url in the application. */
function ondownload(figure, format) {
    window.open('download.' + format, '_blank');
};

$(document).ready(
    function() {
        /* It is up to the application to provide a websocket that the figure
will use to communicate to the server. This websocket object can
also be a "fake" websocket that underneath multiplexes messages
from multiple figures, if necessary. */
        var websocket_type = mpl.get_websocket_type();
        var websocket = new websocket_type("%(ws_uri)s");

        // mpl.figure creates a new figure on the webpage.
        var fig = new mpl.figure(
            // A unique numeric identifier for the figure
            %(fig_id)s,
            // A websocket object (or something that behaves like one)
            websocket,
            // A function called when a file type is selected for download
            ondownload,
            // The HTML element in which to place the figure
            $('div#figure'));
    }
);
class MplJs(tornado.web.RequestHandler):
    """
    Serves the generated matplotlib javascript file. The content is
dynamically generated based on which toolbar functions the
user has defined. Call `FigureManagerWebAgg` to get its content.
    """

def get(self):
    self.set_header('Content-Type', 'application/javascript')
    js_content = FigureManagerWebAgg.get_javascript()
    self.write(js_content)

class Download(tornado.web.RequestHandler):
    """
    Handles downloading of the figure in various file formats.
    """

def get(self, fmt):
    manager = self.application.manager
    mimetypes = {
        'ps': 'application/postscript',
        'eps': 'application/postscript',
        'pdf': 'application/pdf',
        'svg': 'image/svg+xml',
        'png': 'image/png',
        'jpeg': 'image/jpeg',
        'tif': 'image/tiff',
        'emf': 'application/emf'
    }
    self.set_header('Content-Type', mimetypes.get(fmt, 'binary'))
    buff = io.BytesIO()
    manager.canvas.print_figure(buff, format=fmt)
    self.write(buff.getvalue())

class WebSocket(tornado.websocket.WebSocketHandler):
    """
    A websocket for interactive communication between the plot in
    the browser and the server.
    
    In addition to the methods required by tornado, it is required to
    have two callback methods:
    """
    - `send_json(json_content)` is called by matplotlib when
      it needs to send json to the browser. `json_content` is
      a JSON tree (Python dictionary), and it is the responsibility
      of this implementation to encode it as a string to send over
      the socket.
`send_binary(blob)` is called to send binary image data to the browser.

```
supports_binary = True

def open(self):
    # Register the websocket with the FigureManager.
    manager = self.application.manager
    manager.add_web_socket(self)
    if hasattr(self, 'set_nodelay'):
        self.set_nodelay(True)

def on_close(self):
    # When the socket is closed, deregister the websocket with
    # the FigureManager.
    manager = self.application.manager
    manager.remove_web_socket(self)

def on_message(self, message):
    # The 'supports_binary' message is relevant to the
    # websocket itself. The other messages get passed along
    # to matplotlib as-is.
    # Every message has a "type" and a "figure_id".
    message = json.loads(message)
    if message['type'] == 'supports_binary':
        self.supports_binary = message['value']
    else:
        manager = self.application.manager
        manager.handle_json(message)

def send_json(self, content):
    self.write_message(json.dumps(content))

def send_binary(self, blob):
    if self.supports_binary:
        self.write_message(blob, binary=True)
    else:
        data_uri = "data:image/png;base64,\{0}\".format(
            blob.encode('base64').replace('\n', ''))
        self.write_message(data_uri)

def __init__(self, figure):
    self.figure = figure
    self.manager = new_figure_manager_given_figure(
        id(figure), figure)

super(MyApplication, self).__init__(
    # Static files for the CSS and JS
    (r'/_static/(.*)',
     tornado.web.StaticFileHandler,
     {'path': FigureManagerWebAgg.get_static_file_path()}),
# The page that contains all of the pieces
('/', self.MainPage),

('/mpl.js', self.MplJs),

# Sends images and events to the browser, and receives
# events from the browser
('ws', self.WebSocket),

# Handles the downloading (i.e., saving) of static images
(r'/download.([a-z0-9.]+)', self.Download),
]

def _run_[__name__] == '__main__':
    figure = create_figure()
    application = MyApplication(figure)

    http_server = tornado.httpserver.HTTPServer(application)
    http_server.listen(8080)

    print("http://127.0.0.1:8080/")
    print("Press Ctrl+C to quit")

    tornado.ioloop.IOLoop.instance().start

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

112.16 user_interfaces example code: fourier_demo_wx.py

[source code]

```python
import numpy as np

# matplotlib requires wxPython 2.8+
# set the wxPython version in lib\site-packages\wx.pth file
# or if you have wxversion installed un-comment the lines below
# import wxversion
# wxversion.ensureMinimal('2.8')

import wx
import matplotlib
matplotlib.interactive(False)
matplotlib.use('WXAgg')
from matplotlib.backends.backend_wxagg import FigureCanvasWxAgg
from matplotlib.figure import Figure
from matplotlib.pyplot import gcf, setp

class Knob(object):
```
Knob - simple class with a "setKnob" method.
A Knob instance is attached to a Param instance, e.g., param.attach(knob)
Base class is for documentation purposes.

```python
def setKnob(self, value):
    pass
```

class Param(object):
    """
The idea of the "Param" class is that some parameter in the GUI may have several knobs that both control it and reflect the parameter's state, e.g. a slider, text, and dragging can all change the value of the frequency in the waveform of this example.
The class allows a cleaner way to update/"feedback" to the other knobs when one is being changed. Also, this class handles min/max constraints for all the knobs.
Idea - knob list - in "set" method, knob object is passed as well - the other knobs in the knob list have a "set" method which gets called for the others.
    """
    def __init__(self, initialValue=None, minimum=0., maximum=1.):
        self.minimum = minimum
        self.maximum = maximum
        if initialValue != self.constrain(initialValue):
            raise ValueError('illegal initial value')
        self.value = initialValue
        self.knobs = []
    def attach(self, knob):
        self.knobs += [knob]
    def set(self, value, knob=None):
        self.value = value
        self.value = self.constrain(value)
        for feedbackKnob in self.knobs:
            if feedbackKnob != knob:
                feedbackKnob.setKnob(self.value)
        return self.value
    def constrain(self, value):
        if value <= self.minimum:
            value = self.minimum
        if value >= self.maximum:
            value = self.maximum
        return value

class SliderGroup(Knob):
    def __init__(self, parent, label, param):
        """
self.sliderLabel = wx.StaticText(parent, label=label)
self.sliderText = wx.TextCtrl(parent, -1, style=wx.TE_PROCESS_ENTER)
self.slider = wx.Slider(parent, -1)
# self.slider.SetMax(param.maximum*1000)
self.slider.SetRange(0, param.maximum * 1000)
self.setKnob(param.value)

sizer = wx.BoxSizer(wx.HORIZONTAL)
sizer.Add(self.sliderLabel, 0,
wx.EXPAND | wx.ALIGN_CENTER | wx.ALL,
border=2)
sizer.Add(self.sliderText, 0,
wx.EXPAND | wx.ALIGN_CENTER | wx.ALL,
border=2)
sizer.Add(self.slider, 1, wx.EXPAND)
self.sizer = sizer

self.slider.Bind(wx.EVT_SLIDER, self.sliderHandler)
self.sliderText.Bind(wx.EVT_TEXT_ENTER, self.sliderTextHandler)

self.param = param
self.param.attach(self)

def sliderHandler(self, evt):
    value = evt.GetInt() / 1000.
    self.param.set(value)

def sliderTextHandler(self, evt):
    value = float(self.sliderText.GetValue())
    self.param.set(value)

def setKnob(self, value):
    self.sliderText.SetValue('%g' % value)
    self.slider.SetValue(value * 1000)

class FourierDemoFrame(wx.Frame):
    def __init__(self, *args, **kwargs):
        wx.Frame.__init__(self, *args, **kwargs)

        self.fourierDemoWindow = FourierDemoWindow(self)
        self.frequencySliderGroup = SliderGroup(
            self,
            label='Frequency f0:',
            param=self.fourierDemoWindow.f0)
        self.amplitudeSliderGroup = SliderGroup(self, label='Amplitude a:',
            param=self.fourierDemoWindow.A)

        sizer = wx.BoxSizer(wx.VERTICAL)
sizer.Add(self.fourierDemoWindow, 1, wx.EXPAND)
sizer.Add(self.frequencySliderGroup.sizer, 0,
wx.EXPAND | wx.ALIGN_CENTER | wx.ALL, border=5)
sizer.Add(self.amplitudeSliderGroup.sizer, 0,
class FourierDemoWindow(wx.Window, Knob):
    def __init__(self, *args, **kwargs):
        wx.Window.__init__(self, *args, **kwargs)
        self.lines = []
        self.figure = Figure()
        self.canvas = FigureCanvasWxAgg(self, -1, self.figure)
        self.canvas.callbacks.connect('button_press_event', self.mouseDown)
        self.canvas.callbacks.connect('motion_notify_event', self.mouseMotion)
        self.canvas.callbacks.connect('button_release_event', self.mouseUp)
        self.state = ''
        self.mouseInfo = (None, None, None, None)
        self.f0 = Param(2., minimum=0., maximum=6.)
        self.A = Param(1., minimum=0.01, maximum=2.)
        self.draw()

        # Not sure I like having two params attached to the same Knob,
        # but that is what we have here... it works but feels kludgy -
        # although maybe it's not too bad since the knob changes both params
        # at the same time (both f0 and A are affected during a drag)
        self.f0.attach(self)
        self.A.attach(self)
        self.Bind(wx.EVT_SIZE, self.sizeHandler)
        self.Bind(wx.EVT_PAINT, self.OnPaint)

    def OnPaint(self, event):
        self.canvas.draw()
        event.Skip()

    def sizeHandler(self, *args, **kwargs):
        self.canvas.SetSize(self.GetSize())

    def mouseDown(self, evt):
        if self.lines[0] in self.figure.hitlist(evt):
            self.state = 'frequency'
        elif self.lines[1] in self.figure.hitlist(evt):
            self.state = 'time'
        else:
            self.state = ''
        self.mouseInfo = (evt.xdata, evt.ydata, max(self.f0.value, .1), self.A.value)

    def mouseMotion(self, evt):
        if self.state == '':
            return
        x, y = evt.xdata, evt.ydata
        if x is None:  # outside the axes
            return
```python
x0, y0, f0Init, AInit = self.mouseInfo
self.A.set(AInit + (AInit * (y - y0) / y0), self)
if self.state == 'frequency':
    self.f0.set(f0Init + (f0Init * (x - x0) / x0))
elif self.state == 'time':
    if (x - x0) / x0 != -1.:
        self.f0.set(1. / (1. / f0Init + (1. / f0Init * (x - x0) / x0)))

def mouseUp(self, evt):
    self.state = ''
def draw(self):
    if not hasattr(self, 'subplot1'):
        self.subplot1 = self.figure.add_subplot(211)
        self.subplot2 = self.figure.add_subplot(212)
    x1, y1, x2, y2 = self.compute(self.f0.value, self.A.value)
    color = (1., 0., 0.)
    self.lines += self.subplot1.plot(x1, y1, color=color, linewidth=2)
    self.lines += self.subplot2.plot(x2, y2, color=color, linewidth=2)
    # Set some plot attributes
    self.subplot1.set_title("Click and drag waveforms to change frequency and amplitude", fontsize=12)
    self.subplot1.set_ylabel("Frequency Domain Waveform X(f)", fontsize=8)
    self.subplot1.set_xlabel("frequency f", fontsize=8)
    self.subplot2.set_ylabel("Time Domain Waveform x(t)", fontsize=8)
    self.subplot2.set_xlabel("time t", fontsize=8)
    self.subplot1.set_xlim([-6, 6])
    self.subplot1.set_ylim([0, 1])
    self.subplot2.set_xlim([-2, 2])
    self.subplot2.set_ylim([-2, 2])
    self.subplot1.text(0.05, .95, r'$X(f) = \mathcal{F}\{x(t)\}$', verticalalignment='top', transform=self.subplot1.transAxes)
    self.subplot2.text(0.05, .95, r'$x(t) = a \cdot \cos(2\pi f_0 t) e^{-\pi t^2}$', verticalalignment='top', transform=self.subplot2.transAxes)

def compute(self, f0, A):
    f = np.arange(-6., 6., 0.02)
    t = np.arange(-2., 2., 0.01)
    x = A * np.cos(2 * np.pi * f0 * t) * np.exp(-np.pi * t ** 2)
    X = A / 2 * \n        (np.exp(-np.pi * (f - f0) ** 2) + np.exp(-np.pi * (f + f0) ** 2))
    return f, X, t, x

def repaint(self):
    self.canvas.draw()

def setKnob(self, value):
    # Note, we ignore value arg here and just go by state of the params
```
\[ x_1, y_1, x_2, y_2 = \text{self}.\text{compute}(\text{self}.f0.\text{value}, \text{self}.A.\text{value}) \]
\[ \text{setp(self}.\text{lines}[0], \text{xdata}=x_1, \text{ydata}=y_1) \]
\[ \text{setp(self}.\text{lines}[1], \text{xdata}=x_2, \text{ydata}=y_2) \]
\[ \text{self}.\text{repaint()} \]

class \text{App}(\text{wx.App}):  
  \text{def} \text{OnInit(self):}  
    \text{self}.\text{frame1} = \text{FourierDemoFrame(parent=None, title=\"Fourier Demo\", size=(640, 480))}  
    \text{self}.\text{frame1}.\text{Show()}  
    \text{return True}  

\text{app} = \text{App()}  
\text{app}.\text{MainLoop()} 

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

112.17 user\_interfaces example code: gtk\_spreadsheet.py

[source code]

"""
Example of embedding matplotlib in an application and interacting with a treeview to store data. Double click on an entry to update plot data
"""

import pygtk
pygtk.require('2.0')
import gtk
from gtk import gdk

import matplotlib
matplotlib.use('GTKAgg')  # or 'GTK'
from matplotlib.backends.backend_gtk import FigureCanvasGTK as FigureCanvas

from numpy.random import random
from matplotlib.figure import Figure

class DataManager(gtk.Window):
    numRows, numCols = 20, 10

    data = random((numRows, numCols))

    def __init__(self):
        gtk.Window.__init__(self)
        self.set_default_size(600, 600)
        self.connect('destroy', lambda win: gtk.main_quit())
self.set_title('GtkListStore demo')
self.set_border_width(8)

vbox = gtk.VBox(False, 8)
self.add(vbox)

label = gtk.Label('Double click a row to plot the data')

vbox.pack_start(label, False, False)

sw = gtk.ScrolledWindow()
sw.set_shadow_type(gtk.SHADOW_ETCHED_IN)
sw.set_policy(gtk.POLICY_NEVER,
gtk.POLICY_AUTOMATIC)
vbox.pack_start(sw, True, True)

model = self.create_model()

self.treeview = gtk.TreeView(model)
self.treeview.set_rules_hint(True)

# matplotlib stuff
fig = Figure(figsize=(6, 4))
self.canvas = FigureCanvas(fig)
# a gtk.DrawingArea
vbox.pack_start(self.canvas, True, True)

ax = fig.add_subplot(111)
self.line, = ax.plot(self.data[0, :], 'go')  # plot the first row

self.treeview.connect('row-activated', self.plot_row)
sw.add(self.treeview)

self.add_columns()

self.add_events(gdk.BUTTON_PRESS_MASK |
gdk.KEY_PRESS_MASK |
gdk.KEY_RELEASE_MASK)

def plot_row(self, treeview, path, view_column):
    ind, = path  # get the index into data
    points = self.data[ind, :]
    self.line.set_ydata(points)
    self.canvas.draw()

def add_columns(self):
    for i in range(self.numCols):
        column = gtk.TreeViewColumn('%d' % i, gtk.CellRendererText(), text=i)
        self.treeview.append_column(column)

def create_model(self):
    types = [float]*self.numCols
    store = gtk.ListStore(*types)
for row in self.data:
    store.append(row)
return store

manager = DataManager()
manager.show_all()
gtk.main()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

112.18 user_interfaces example code: histogram_demo_canvasagg.py

[source code]

"""
This is an example that shows you how to work directly with the agg
figure canvas to create a figure using the pythonic API.

In this example, the contents of the agg canvas are extracted to a
string, which can in turn be passed off to PIL or put in a numeric
array

"""
from matplotlib.backends.backend_agg import FigureCanvasAgg
from matplotlib.figure import Figure
from matplotlib.mlab import normpdf
from numpy.random import randn
import numpy

fig = Figure(figsize=(5, 4), dpi=100)
ax = fig.add_subplot(111)
canvas = FigureCanvasAgg(fig)

mu, sigma = 100, 15
x = mu + sigma*randn(10000)

# the histogram of the data
n, bins, patches = ax.hist(x, 50, normed=1)

# add a 'best fit' line
y = normpdf(bins, mu, sigma)
line, = ax.plot(bins, y, 'r--')
line.set_linewidth(1)

ax.set_xlabel('Smarts')
ax.set_ylabel('Probability')
ax.set_title(r'$\text{Histogram of IQ: } \mu=100, \sigma=15$')

112.18. user_interfaces example code: histogram_demo_canvasagg.py 2969
ax.set_xlim((40, 160))
ax.set_ylim((0, 0.03))
canvas.draw()

s = canvas.tostring_rgb()  # save this and convert to bitmap as needed

# get the figure dimensions for creating bitmaps or numpy arrays, 
# etc.
l, b, w, h = fig.bbox.bounds
w, h = int(w), int(h)

if 0:
    # convert to a numpy array
    X = numpy.fromstring(s, numpy.uint8)
    X.shape = h, w, 3

if 0:
    # pass off to PIL
    from PIL import Image
    im = Image.fromstring("RGB", (w, h), s)
    im.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

112.19 user_interfaces example code: interactive.py

[source code]

"""Multithreaded interactive interpreter with GTK and Matplotlib support.

WARNING:
As of 2010/06/25, this is not working, at least on Linux.
I have disabled it as a runnable script. - EF

Usage:

    pyint-gtk.py  ->  starts shell with gtk thread running separately

    pyint-gtk.py  -pylab [filename]  ->  initializes matplotlib, optionally running
    the named file. The shell starts after the file is executed.

Threading code taken from:
http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/65109, by Brian
McErlean and John Finlay.

Matplotlib support taken from interactive.py in the matplotlib distribution.

Also borrows liberally from code.py in the Python standard library."""
from __future__ import print_function

__author__ = "Fernando Perez <Fernando.Perez@colorado.edu>"

import sys
import code
import threading

import gobject
import gtk

try:
    import readline
except ImportError:
    has_readline = False
else:
    has_readline = True

class MTConsole(code.InteractiveConsole):
    """Simple multi-threaded shell""

    def __init__(self, on_kill=None, *args, **kw):
        code.InteractiveConsole.__init__(self, *args, **kw)
        self.code_to_run = None
        self.ready = threading.Condition()
        self._kill = False
        if on_kill is None:
            on_kill = []
        # Check that all things to kill are callable:
        for _ in on_kill:
            if not callable(_):
                raise TypeError('on_kill must be a list of callables')
        self.on_kill = on_kill
        # Set up tab-completer
        if has_readline:
            import rlcompleter
            try:
                # this form only works with python 2.3
                self.completer = rlcompleter.Completer(self.locals)
            except:
                # simpler for py2.2
                self.completer = rlcompleter.Completer()
            readline.set_completer(self.completer.complete)
            # Use tab for completions
            readline.parse_and_bind('tab: complete')
            # This forces readline to automatically print the above list when tab
            # completion is set to 'complete'.
            readline.parse_and_bind('set show-all-if-ambiguous on')
            # Bindings for incremental searches in the history. These searches
            # use the string typed so far on the command line and search
            # anything in the previous input history containing them.
            readline.parse_and_bind('"\C-r": reverse-search-history')
            readline.parse_and_bind('"\C-s": forward-search-history')
def runsource(self, source, filename="<input>", symbol="single"):  
    """Compile and run some source in the interpreter.

    Arguments are as for compile_command().

    One several things can happen:

    1) The input is incorrect; compile_command() raised an
    exception (SyntaxError or OverflowError). A syntax traceback
    will be printed by calling the showsyntaxerror() method.

    2) The input is incomplete, and more input is required;
    compile_command() returned None. Nothing happens.

    3) The input is complete; compile_command() returned a code
    object. The code is executed by calling self.runode() (which
    also handles run-time exceptions, except for SystemExit).

    The return value is True in case 2, False in the other cases (unless
    an exception is raised). The return value can be used to
    decide whether to use sys.ps1 or sys.ps2 to prompt the next
    line.
    """
    try:
        code = self.compile(source, filename, symbol)
    except (OverflowError, SyntaxError, ValueError):
        # Case 1
        self.showsyntaxerror(filename)
        return False
    if code is None:
        # Case 2
        return True
    # Case 3
    # Store code in self, so the execution thread can handle it
    self.ready.acquire()
    self.code_to_run = code
    self.ready.wait()  # Wait until processed in timeout interval
    self.ready.release()
    return False

def runode(self):
    """Execute a code object.

    When an exception occurs, self.showtraceback() is called to display a
    traceback."""
    self.ready.acquire()
    if self._kill:
        print('Closing threads...')
sys.stdout.flush()
for tokill in self.on_kill:
    tokill()
    print('Done.')

if self.code_to_run is not None:
    self.ready.notify()
    code.InteractiveConsole.runcode(self, self.code_to_run)

    self.code_to_run = None
    self.ready.release()
    return True

def kill(self):
    """Kill the thread, returning when it has been shut down.""
    self.ready.acquire()
    self._kill = True
    self.ready.release()

class GTKInterpreter(threading.Thread):
    """Run gtk.main in the main thread and a python interpreter in a
    separate thread.
    Python commands can be passed to the thread where they will be executed.
    This is implemented by periodically checking for passed code using a
    GTK timeout callback.
    """
    TIMEOUT = 100  # Millisecond interval between timeouts.

    def __init__(self, banner=None):
        threading.Thread.__init__(self)
        self.banner = banner
        self.shell = MTConsole(on_kill=[gtk.main_quit])

    def run(self):
        self.pre_interact()
        self.shell.interact(self.banner)
        self.shell.kill()

    def mainloop(self):
        self.start()
        gobject.timeout_add(self.TIMEOUT, self.shell.runcode)
        try:
            if gtk.gtk_version[0] >= 2:
                gtk.gdk.threads_init()
        except AttributeError:
            pass
        gtk.main()
        self.join()

    def pre_interact(self):
        """This method should be overridden by subclasses."""
It gets called right before interact(), but after the thread starts. Typically used to push initialization code into the interpreter"

    pass

class MatplotlibInterpreter(GTKInterpreter):
    """Threaded interpreter with matplotlib support.

    Note that this explicitly sets GTKAgg as the backend, since it has specific GTK hooks in it.""

def __init__(self, banner=None):
    banner = "\nWelcome to matplotlib, a MATLAB-like python environment. help(matlab) -> help on matlab compatible commands from matplotlib. help(plotting) -> help on plotting commands. """
    GTKInterpreter.__init__(self, banner)

def pre_interact(self):
    """Initialize matplotlib before user interaction begins"""

    push = self.shell.push
    # Code to execute in user's namespace
    lines = ['"import matplotlib",
             "matplotlib.use('GTKAgg')",
             "matplotlib.interactive(1)",
             "import matplotlib.pylab as pylab",
             "from matplotlib.pylab import *
"
             ]

    map(push, lines)

    # Execute file if given.
    if len(sys.argv) > 1:
        import matplotlib
        matplotlib.interactive(0)  # turn off interaction
        fname = sys.argv[1]
        try:
            inFile = file(fname, 'r')
        except IOError:
            print('*** ERROR *** Could not read file <%s>' % fname)
        else:
            print('*** Executing file <%s>: ' % fname)
            for line in inFile:
                if line.lstrip().find('show()') == 0:
                    continue
                print('>>>', line)
                push(line)
            inFile.close()
            matplotlib.interactive(1)  # turn on interaction

if __name__ == '__main__':
    print("This demo is not presently functional, so running")
print("it as a script has been disabled.")
sys.exit()
# Quick sys.argv hack to extract the option and leave filenames in sys.argv.
# For real option handling, use optparse or getopt.
if len(sys.argv) > 1 and sys.argv[1] == '-pylab':
    sys.argv = [sys.argv[0]] + sys.argv[2:]
    MatplotlibInterpreter().mainloop()
else:
    GTKInterpreter().mainloop()
def __init__(self, locals):
    self.locals = locals

def complete(self, text, state):
    """Return the next possible completion for 'text'.
    This is called successively with state == 0, 1, 2, ... until it
    returns None. The completion should begin with 'text'.
    """
    if state == 0:
        if "." in text:
            self.matches = self.attr_matches(text)
        else:
            self.matches = self.global_matches(text)
        try:
            return self.matches[state]
        except IndexError:
            return None

def global_matches(self, text):
    """Compute matches when text is a simple name.
    Return a list of all keywords, built-in functions and names
    currently defines in __main__ that match.
    """
    import keyword
    matches = []
    n = len(text)
    for list in [keyword.kwlist, __builtin__.__dict__.keys(), __main__.__dict__.keys(), self.locals.keys()]:
        for word in list:
            if word[:n] == text and word != "__builtins__":
                matches.append(word)
    return matches

def attr_matches(self, text):
    """Compute matches when text contains a dot.
    Assuming the text is of the form NAME.NAME....[NAME], and is
    evaluable in the globals of __main__, it will be evaluated
    and its attributes (as revealed by dir()) are used as possible
    completions. (For class instances, class members are are also
    considered.)
    WARNING: this can still invoke arbitrary C code, if an object
    with a __getattr__ hook is evaluated."""
```python
import re
m = re.match(r"(\w+(\.\w+)*)\.(\w*)", text)
if not m:
    return
expr, attr = m.group(1, 3)
object = eval(expr, __main__.__dict__, self.locals)
words = dir(object)
if hasattr(object, '__class__'):
    words.append('__class__')
    words += get_class_members(object.__class__)
matches = []
n = len(attr)
for word in words:
    if word[:n] == attr and word != '__builtins__':
        matches.append('%s.%s % (expr, word))
return matches

def get_class_members(klass):
    ret = dir(klass)
    if hasattr(klass, '__bases__'):
        for base in klass.__bases__:
            ret += get_class_members(base)
    return ret

class OutputStream(object):
    """
    A Multiplexing output stream.
    It can replace another stream, and tee output to the original stream and too
    a GTK textview.
    """

def __init__(self, view, old_out, style):
    self.view = view
    self.buffer = view.get_buffer()
    self.mark = self.buffer.create_mark("End", self.buffer.get_end_iter(), False)
    self.out = old_out
    self.style = style
    self.tee = 1

def write(self, text):
    if self.tee:
        self.out.write(text)
    end = self.buffer.get_end_iter()
    if self.view is not None:
        self.view.scroll_to_mark(self.mark, 0, True, 1, 1)
    self.buffer.insert_with_tags(end, text, self.style)
```

---

112.20. user_interfaces example code: interactive2.py 2977
class GTKInterpreterConsole(gtk.ScrolledWindow):
    
    """
    An InteractiveConsole for GTK. It's an actual widget, so it can be dropped in just about anywhere.
    """

    def __init__(self):
        gtk.ScrolledWindow.__init__(self)
        self.set_policy(gtk.POLICY_AUTOMATIC, gtk.POLICY_AUTOMATIC)

        self.text = gtk.TextView()
        self.text.set_wrap_mode(True)

        self.interpreter = code.InteractiveInterpreter()

        self.completer = Completer(self.interpreter.locals)
        self.buffer = []
        self.history = []
        self.banner = banner
        self.ps1 = ">>> 
        self.ps2 = "... 

        self.text.add_events(gtk.gdk.KEY_PRESS_MASK)
        self.text.connect("key_press_event", self.key_pressed)

        self.current_history = -1

        self.mark = self.text.get_buffer().create_mark("End", self.text.get_buffer().get_end_iter(), False)

        # setup colors
        self.style_banner = gtk.TextTag("banner")
        self.style_banner.set_property("foreground", "saddle brown")

        self.style_ps1 = gtk.TextTag("ps1")
        self.style_ps1.set_property("foreground", "DarkOrchid4")
        self.style_ps1.set_property("editable", False)
        self.style_ps1.set_property("font", "courier")

        self.style_ps2 = gtk.TextTag("ps2")
        self.style_ps2.set_property("foreground", "DarkOliveGreen")
        self.style_ps2.set_property("editable", False)
        self.style_ps2.set_property("font", "courier")

        self.style_out = gtk.TextTag("stdout")
        self.style_out.set_property("foreground", "midnight blue")

        self.style_err = gtk.TextTag("stderr")
        self.style_err.set_property("style", pango.STYLE_ITALIC)
        self.style_err.set_property("foreground", "red")

        self.text.get_buffer().get_tag_table().add(self.style_banner)
        self.text.get_buffer().get_tag_table().add(self.style_ps1)
self.text.get_buffer().get_tag_table().add(self.style_ps2)
self.text.get_buffer().get_tag_table().add(self.style_out)
self.text.get_buffer().get_tag_table().add(self.style_err)

self.stdout = OutputStream(self.text, sys.stdout, self.style_out)
self.stderr = OutputStream(self.text, sys.stderr, self.style_err)

sys.stderr = self.stderr
sys.stdout = self.stdout

self.current_prompt = None

self.write_line(self.banner, self.style_banner)
self.prompt_ps1()

self.add(self.text)
self.text.show()

def reset_history(self):
    self.history = []

def reset_buffer(self):
    self.buffer = []

def prompt_ps1(self):
    self.current_prompt = self.prompt_ps1
    self.write_line(self.ps1, self.style_ps1)

def prompt_ps2(self):
    self.current_prompt = self.prompt_ps2
    self.write_line(self.ps2, self.style_ps2)

def write_line(self, text, style=None):
    start, end = self.text.get_buffer().get_bounds()
    if style is None:
        self.text.get_buffer().insert(end, text)
    else:
        self.text.get_buffer().insert_with_tags(end, text, style)

    self.text.scroll_to_mark(self.mark, 0, True, 1, 1)

def push(self, line):

    self.buffer.append(line)
    if len(line) > 0:
        self.history.append(line)

    source = 

    more = self.interpreter.runsource(source, "<<console>>")

    if not more:
        self.reset_buffer()}
return more

def key_pressed(self, widget, event):
    if event.keyval == gtk.gdk.keyval_from_name('Return'):
        return self.execute_line()

    if event.keyval == gtk.gdk.keyval_from_name('Up'):
        self.current_history = self.current_history - 1
        if self.current_history < - len(self.history):
            self.current_history = - len(self.history)
        return self.show_history()
    elif event.keyval == gtk.gdk.keyval_from_name('Down'):
        self.current_history = self.current_history + 1
        if self.current_history > 0:
            self.current_history = 0
        return self.show_history()
    elif event.keyval == gtk.gdk.keyval_from_name('Home'):
        l = self.text.get_buffer().get_line_count() - 1
        start = self.text.get_buffer().get_iter_at_line_offset(l, 4)
        self.text.get_buffer().place_cursor(start)
        return True
    elif event.keyval == gtk.gdk.keyval_from_name('space') and event.state & gtk.gdk.CONTROL_MASK:
        return self.complete_line()
    return False

def show_history(self):
    if self.current_history == 0:
        return True
    else:
        self.replace_line(self.history[self.current_history])
        return True

def current_line(self):
    start, end = self.current_line_bounds()
    return self.text.get_buffer().get_text(start, end, True)

def current_line_bounds(self):
    txt_buffer = self.text.get_buffer()
    l = txt_buffer.get_line_count() - 1
    start = txt_buffer.get_iter_at_line(l)
    if start.get_chars_in_line() >= 4:
        start.forward_chars(4)
    end = txt_buffer.get_end_iter()
    return start, end

def replace_line(self, txt):
    start, end = self.current_line_bounds()
    self.text.get_buffer().delete(start, end)
    self.write_line(txt)
def execute_line(self, line=None):
    if line is None:
        line = self.current_line()
        self.write_line("\n")
    else:
        self.write_line(line + "\n")

    more = self.push(line)

    self.text.get_buffer().place_cursor(self.text.get_buffer().get_end_iter())

    if more:
        self.prompt_ps2()
    else:
        self.prompt_ps1()

    self.current_history = 0
    self.window.raise_()
    return True

def complete_line(self):
    line = self.current_line()
    tokens = line.split()
    token = tokens[-1]

    completions = []
    p = self.completer.complete(token, len(completions))
    while p is not None:
        completions.append(p)
        p = self.completer.complete(token, len(completions))

    if len(completions) != 1:
        self.write_line("\n")
        self.write_line("\n".join(completions), self.style_ps1)
        self.write_line("\n")
        self.current_prompt()
        self.write_line(line)
    else:
        i = line.rfind(token)
        line = line[0:i] + completions[0]
        self.replace_line(line)

    return True

def main():
    w = gtk.Window()
    console = GTKInterpreterConsole()
    console.set_size_request(640, 480)
    w.add(console)
def destroy(arg=None):
    gtk.main_quit()

def key_event(widget, event):
    if gtk.gdk.keyval_name(event.keyval) == 'd' and event.state & gtk.gdk.CONTROL_MASK:
        destroy()
    return False

w.connect("destroy", destroy)

w.add_events(gtk.gdk.KEY_PRESS_MASK)
w.connect('key_press_event', key_event)
w.show_all()

console.execute_line('import matplotlib')
console.execute_line("matplotlib.use('GTKAgg')")
console.execute_line('matplotlib.interactive(1)')
console.execute_line('from pylab import *')

if len(sys.argv) > 1:
    fname = sys.argv[1]
    if not os.path.exists(fname):
        print('%s does not exist' % fname)
        for line in file(fname):
            line = line.strip()
            console.execute_line(line)
    gtk.main()

if __name__ == '__main__':
    main()
t1 = np.arange(0.0, 5.0, 0.1)
t2 = np.arange(0.0, 5.0, 0.02)
t3 = np.arange(0.0, 2.0, 0.01)

fig, ax = plt.subplots()
l1, = ax.plot(t1, f(t1), 'bo', label='line 1')
l2, = ax.plot(t2, f(t2), 'k--', label='line 2')

dlg = DialogLineprops([l1, l2])
dlg.show()
plt.show()
(r'$\frac{4}{3}\pi x^3$', lambda x: (4.0/3.0)*pi*x**3),
(r'$\cos(2 \pi x)$', lambda x: cos(2*pi*x)),
(r'$\log(x)$',  lambda x: log(x))
]

class CanvasFrame(wx.Frame):
    def __init__(self, parent, title):
        wx.Frame.__init__(self, parent, -1, title, size=(550, 350))
        self.SetBackgroundColour(wxc.NamedColour("WHITE"))

        self.figure = Figure()
        self.axes = self.figure.add_subplot(111)

        self.canvas = FigureCanvas(self, -1, self.figure)

        self.change_plot(0)

        self.sizer = wx.BoxSizer(wx.VERTICAL)
        self.add_buttonbar()

        self.canvas = FigureCanvas(self, -1, self.figure)

        self.change_plot(0)

        self.sizer = wx.BoxSizer(wx.VERTICAL)
        self.add_buttonbar()

        menuBar = wx.MenuBar()

        menu = wx.Menu()  # File Menu
        menu.Append(wx.ID_EXIT, "E&xit	Alt-X", "Exit this simple sample")
        menuBar.Append(menu, "&File")

        if IS_GTK or IS_WIN:
            # Equation Menu
            for i, (mt, func) in enumerate(functions):
                bm = mathtext_to_wxbitmap(mt)
                item = wx.MenuItem(menu, 1000 + i, "")
                item.SetBitmap(bm)
                menu.AppendItem(item)

                self.Bind(wx.EVT_MENU, self.OnChangePlot, item)
            menuBar.Append(menu, "&Functions")

        self.SetMenuBar(menuBar)

        self.SetSizer(self.sizer)
        self.Fit()

    def add_buttonbar(self):
        self.button_bar = wx.Panel(self)
        self.button_bar_sizer = wx.BoxSizer(wx.HORIZONTAL)

        self.sizer.Add(self.button_bar, 0, wx.LEFT | wx.TOP | wx.GROW)

        for i, (mt, func) in enumerate(functions):
            bm = mathtext_to_wxbitmap(mt)
button = wx.BitmapButton(self.button_bar, 1000 + i, bm)
self.button_bar_sizer.Add(button, 1, wx.GROW)
self.Bind(wx.EVT_BUTTON, self.OnChangePlot, button)

self.button_bar_sizer.SetSizer(self.button_bar_sizer)

def add_toolbar(self):
    """Copied verbatim from embedding.wx2.py""
    self.toolbar = NavigationToolbar2Wx(self.canvas)
    self.toolbar.Realize()
    # By adding toolbar in sizer, we are able to put it at the bottom
    # of the frame - so appearance is closer to GTK version.
    self.sizer.Add(self.toolbar, 0, wx.LEFT | wx.EXPAND)
    # update the axes menu on the toolbar
    self.toolbar.update()

def OnChangePlot(self, event):
    self.change_plot(event.GetId() - 1000)

def change_plot(self, plot_number):
    t = arange(1.0, 3.0, 0.01)
    s = functions[plot_number][1](t)
    self.axes.clear()
    self.axes.plot(t, s)
    self.canvas.draw()

class MyApp(wx.App):
    def OnInit(self):
        frame = CanvasFrame(None, "wxPython mathtext demo app")
        self.SetTopWindow(frame)
        frame.Show(True)
        return True

app = MyApp()
app.MainLoop()
from matplotlib.widgets import SpanSelector

from numpy import arange, sin, pi
import gtk
import gtk.glade

def simple_msg(msg, parent=None, title=None):
    dialog = gtk.MessageDialog(
        parent=None,
        type=gtk.MESSAGE_INFO,
        buttons=gtk.BUTTONS_OK,
        message_format=msg)
    if parent is not None:
        dialog.set_transient_for(parent)
    if title is not None:
        dialog.set_title(title)
    dialog.show()
    dialog.run()
    dialog.destroy()
    return None

class GladeHandlers(object):
    def on_buttonClickMe_clicked(event):
        simple_msg('Nothing to say, really', parent=widgets['windowMain'],
                   title='Thanks!')

class WidgetsWrapper(object):
    def __init__(self):
        self.widgets = gtk.glade.XML('mpl_with_glade.glade')
        self.widgets.signal_autoconnect(GladeHandlers.__dict__)
        self['windowMain'].connect('destroy', lambda x: gtk.main_quit())
        self['windowMain'].move(10, 10)
        self.figure = Figure(figsize=(8, 6), dpi=72)
        self.axis = self.figure.add_subplot(111)
        t = arange(0.0, 3.0, 0.01)
        s = sin(2*pi*t)
        self.axis.plot(t, s)
        self.axis.set_xlabel('time (s)')
        self.axis.set_ylabel('voltage')
        self.canvas = FigureCanvas(self.figure)  # a gtk.DrawingArea
        self.canvas.show()
        self.canvas.set_size_request(600, 400)
        self.canvas.set_events(gtk.gdk.BUTTON_PRESS_MASK |
                               gtk.gdk.KEY_PRESS_MASK |
                               gtk.gdk.KEY_RELEASE_MASK)
self.canvas.set_flags(gtk.HAS_FOCUS | gtk.CAN_FOCUS)
self.canvas.grab_focus()

def keypress(widget, event):
    print('key press')

def buttonpress(widget, event):
    print('button press')

self.canvas.connect('key_press_event', keypress)
self.canvas.connect('button_press_event', buttonpress)

def onselect(xmin, xmax):
    print(xmin, xmax)

span = SpanSelector(self.axis, onselect, 'horizontal', useblit=False,
                     rectprops=dict(alpha=0.5, facecolor='red'))

self['vboxMain'].pack_start(self.canvas, True, True)
self['vboxMain'].show()

# below is optional if you want the navigation toolbar
self.navToolbar = NavigationToolbar(self.canvas, self['windowMain'])
self.navToolbar.lastDir = '/var/tmp/
self['vboxMain'].pack_start(self.navToolbar)
self.navToolbar.show()

sep = gtk.HSeparator()
sep.show()
self['vboxMain'].pack_start(sep, True, True)

self['vboxMain'].reorder_child(self['buttonClickMe'], -1)

def __getitem__(self, key):
    return self.widgets.get_widget(key)

widgets = WidgetsWrapper()
gtk.main()
from matplotlib.backends.backend_gtk3agg import FigureCanvasGTK3Agg as FigureCanvas

class Window1Signals(object):
    def on_window1_destroy(self, widget):
        Gtk.main_quit()

def main:
    builder = Gtk.Builder()
    builder.add_objects_from_file("mpl_with_glade_316.glade", ("window1", ""))
    builder.connect_signals(Window1Signals())
    window = builder.get_object("window1")
    sw = builder.get_object("scrolledwindow1")

    # Start of Matplotlib specific code
    figure = Figure(figsize=(8, 6), dpi=71)
    axis = figure.add_subplot(111)
    t = arange(0.0, 3.0, 0.01)
    s = sin(2*pi*t)
    axis.plot(t, s)

    axis.set_xlabel('time [s]')
    axis.set_ylabel('voltage [V]')

    canvas = FigureCanvas(figure)  # a Gtk.DrawingArea
    canvas.set_size_request(800, 600)
    sw.add_with_viewport(canvas)
    # End of Matplotlib specific code

    window.show_all()
    Gtk.main()

if __name__ == '__main__':
    main()


Keywords: python, matplotlib, pylab, example, codex (see Search examples)

112.25 user_interfaces example code: pylab_with_gtk.py

[source code]

""
An example of how to use pylab to manage your figure windows, but modify the GUI by accessing the underlying gtk widgets
""

from __future__ import print_function
import matplotlib
matplotlib.use('GTKAgg')
import matplotlib.pyplot as plt
```python
fig, ax = plt.subplots()
plt.plot([1, 2, 3], 'ro-', label='easy as 1 2 3')
plt.plot([1, 4, 9], 'gs--', label='easy as 1 2 3 squared')
plt.legend()

manager = plt.get_current_fig_manager()
# you can also access the window or vbox attributes this way
toolbar = manager.toolbar

# now let's add a button to the toolbar
import gtk
next = 8  # where to insert this in the mpl toolbar
button = gtk.Button('Click me')
button.show()

def clicked(button):
    print('hi mom')
button.connect('clicked', clicked)

toolitem = gtk.ToolItem()
toolitem.show()
toolitem.set_tooltip(
    toolbar.tooltips,
    'Click me for fun and profit')

toolitem.add(button)
toolbar.insert(toolitem, next)
next += 1

# now let's add a widget to the vbox
label = gtk.Label()
label.set_markup('Drag mouse over axes for position')
label.show()

vbox = manager.vbox
vbox.pack_start(label, False, False)
vbox.reorder_child(manager.toolbar, -1)

def update(event):
    if event.xdata is None:
        label.set_markup('Drag mouse over axes for position')
    else:
        label.set_markup('<span color="#ef0000">x,y=(%f, %f)</span>' % (event.xdata, event.ydata))

plt.connect('motion_notify_event', update)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
generate an editable gtk treeview widget for record arrays with custom
formatting of the cells and show how to limit string entries to a list
of strings

```python
from __future__ import print_function
import gtk
import numpy as np
import matplotlib.mlab as mlab
import matplotlib.cbook as cbook
import mpl_toolkits.gtktools as gtktools

datafile = cbook.get_sample_data('demodata.csv', asfileobj=False)
r = mlab.csv2rec(datafile, converterd={'weekdays': str})

formatd = mlab.get_formatd(r)
formatd['date'] = mlab.FormatDate('%Y-%m-%d')
formatd['prices'] = mlab.FormatMillions(precision=1)
formatd['gain'] = mlab.FormatPercent(precision=2)

# use a drop down combo for weekdays
stringd = dict(weekdays=['Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat'])
constant = ['clientid']  # block editing of this field

liststore = gtktools.RecListStore(r, formatd=formatd, stringd=stringd)
treeview = gtktools.RecTreeView(liststore, constant=constant)

def mycallback(liststore, rownum, colname, oldval, newval):
    print('verify: old=%s, new=%s, rec=%s' % (oldval, newval, liststore.
    →r[rownum][colname]))

liststore.callbacks.connect('cell_changed', mycallback)

win = gtk.Window()
win.set_title('click to edit')
win.add(treeview)
win.show_all()
win.connect('delete-event', lambda *args: gtk.main_quit())
gtk.main()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
**112.27 user_interfaces example code: rec_edit_gtk_simple.py**

[source code]

```
""
Load a CSV file into a record array and edit it in a gtk treeview
""

    import gtk
    import numpy as np
    import matplotlib.mlab as mlab
    import matplotlib.cbook as cbook
    import mpl_toolkits.gtktools as gtktools

datafile = cbook.get_sample_data('demodata.csv', asfileobj=False)
r = mlab.csv2rec(datafile, converterd={'weekdays': str})

liststore, treeview, win = gtktools.edit_recarray(r)
win.set_title('click to edit')
win.connect('delete-event', lambda *args: gtk.main_quit())
gtk.main()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

**112.28 user_interfaces example code: svg_histogram.py**

[source code]

```
""
Demonstrate how to create an interactive histogram, in which bars are hidden or shown by clicking on legend markers.

The interactivity is encoded in ecmascript (javascript) and inserted in the SVG code in a post-processing step. To render the image, open it in a web browser. SVG is supported in most web browsers used by Linux and OSX users. Windows IE9 supports SVG, but earlier versions do not.

Notes
-----
The matplotlib backend lets us assign ids to each object. This is the mechanism used here to relate matplotlib objects created in python and the corresponding SVG constructs that are parsed in the second step. While flexible, ids are cumbersome to use for large collection of objects. Two mechanisms could be used to simplify things:

* systematic grouping of objects into SVG <g> tags,
* assigning classes to each SVG object according to its origin.

For example, instead of modifying the properties of each individual bar, the bars from the 'hist' function could either be grouped in a PatchCollection, or be assigned a class="hist_##" attribute.
```
CSS could also be used more extensively to replace repetitive markup throughout the generated SVG.

Author: david.huard@gmail.com

---

```python
import numpy as np
import matplotlib.pyplot as plt
import xml.etree.ElementTree as ET
from io import BytesIO
import json

plt.rcParams['svg.fonttype'] = 'none

# Apparently, this `register_namespace` method works only with
# python 2.7 and up and is necessary to avoid garbling the XML name
# space with ns0.
ET.register_namespace('', 'http://www.w3.org/2000/svg')

# --- Create histogram, legend and title ---
plt.figure()
r = np.random.randn(100)
r1 = r + 1
labels = ['Rabbits', 'Frogs']
H = plt.hist([r, r1], label=labels)
containers = H[-1]
leg = plt.legend(frameon=False)
plt.title("From a web browser, click on the legend
"  "marker to toggle the corresponding histogram.")

# --- Add ids to the svg objects we'll modify
hist_patches = {}
for ic, c in enumerate(containers):
    hist_patches['hist_%d' % ic] = []
    for il, element in enumerate(c):
        element.set_gid('hist_patch_%d' % (ic, il))
        hist_patches['hist_patch_%d' % (ic, il)].append('hist_patch_%d' % (ic, il))

# Set ids for the legend patches
for i, t in enumerate(leg.get_patches()):
    t.set_gid('leg_patch_%d' % i)

# Set ids for the text patches
for i, t in enumerate(leg.get_texts()):
    t.set_gid('leg_text_%d' % i)

# Save SVG in a fake file object.
```
```python
f = BytesIO()
plt.savefig(f, format="svg")

# Create XML tree from the SVG file.
tree, xmlid = ET.XMLID(f.getvalue())

# --- Add interactivity ---

# Add attributes to the patch objects.
for i, t in enumerate(leg.get_patches()):
    el = xmlid['leg_patch_%d' % i]
el.set('cursor', 'pointer')
el.set('onclick', "toggle_hist(this)")

# Add attributes to the text objects.
for i, t in enumerate(leg.get_texts()):
    el = xmlid['leg_text_%d' % i]
el.set('cursor', 'pointer')
el.set('onclick', "toggle_hist(this)")

# Create script defining the function `toggle_hist`.
# We create a global variable `container` that stores the patches id
# belonging to each histogram. Then a function "toggle_element" sets the
# visibility attribute of all patches of each histogram and the opacity
# of the marker itself.

script = ""
<script type="text/ecmascript">
<![CDATA[
var container = %s

function toggle(oid, attribute, values) {
    /* Toggle the style attribute of an object between two values.

    Parameters
    ----------
    oid : str
        Object identifier.
    attribute : str
        Name of style attribute.
    values : [on state, off state]
        The two values that are switched between.
    */
    var obj = document.getElementById(oid);
    var a = obj.style[attribute];

    a = (a == values[0] || a == "") ? values[1] : values[0];
    obj.style[attribute] = a;
}

function toggle_hist(obj) {
```

112.28. user_interfaces example code: svg_histogram.py 2993
```
var num = obj.id.slice(-1);

toggle('leg_patch_' + num, 'opacity', [1, 0.3]);
toggle('leg_text_' + num, 'opacity', [1, 0.5]);

var names = container['hist_'+num]

for (var i=0; i < names.length; i++) {
    toggle(names[i], 'opacity', [1,0])
};
```
import matplotlib.pyplot as plt
import xml.etree.ElementTree as ET
from io import BytesIO

ET.register_namespace('', 'http://www.w3.org/2000/svg')

fig, ax = plt.subplots()

# Create patches to which tooltips will be assigned.
rect1 = plt.Rectangle((10, -20), 10, 5, fc='blue')
rect2 = plt.Rectangle((-20, 15), 10, 5, fc='green')

shapes = [rect1, rect2]
labels = ['This is a blue rectangle.', 'This is a green rectangle']

for i, (item, label) in enumerate(zip(shapes, labels)):
    patch = ax.add_patch(item)
    annotate = ax.annotate(labels[i], xy=item.get_xy(), xytext=(0, 0),
                            textcoords='offset points', color='w', ha='center',
                            fontsize=8, bbox=dict(boxstyle='round', pad=.5,
                                                  fc=(.1, .1, .1, .92),
                                                  ec=(1., 1., 1.), lw=1,
                                                  zorder=1))

    ax.add_patch(patch)
    patch.set_gid('mypatch_{:03d}'.format(i))
    annotate.set_gid('mytooltip_{:03d}'.format(i))

# Save the figure in a fake file object
ax.set_xlim(-30, 30)
ax.set_ylim(-30, 30)
ax.set_aspect('equal')

f = BytesIO()
plt.savefig(f, format='svg')

# --- Add interactivity ---

# Create XML tree from the SVG file.
tree, xmlid = ET.XMLID(f.getvalue())
tree.set('onload', 'init(evt)')

for i in shapes:
    # Get the index of the shape
    index = shapes.index(i)
    # Hide the tooltips
    tooltip = xmlid['mytooltip_{:03d}'.format(index)]
    tooltip.set('visibility', 'hidden')
    # Assign onmouseover and onmouseout callbacks to patches.
mypatch = xmlid['mypatch_{:03d}'.format(index)]
mypatch.set('onmouseover', "ShowTooltip(this)"
)mypatch.set('onmouseout', "HideTooltip(this)"

# This is the script defining the ShowTooltip and HideTooltip functions.
script = ""
<script type="text/ecmascript">
<![CDATA[
    function init(evt) {
        if ( window.svgDocument == null ) {
            svgDocument = evt.target.ownerDocument;
        }
    }

    function ShowTooltip(obj) {
        var cur = obj.id.split('_')[1];
        var tip = svgDocument.getElementById('mytooltip_' + cur);
        tip.setAttribute('visibility','visible')
    }

    function HideTooltip(obj) {
        var cur = obj.id.split('_')[1];
        var tip = svgDocument.getElementById('mytooltip_' + cur);
        tip.setAttribute('visibility','hidden')
    }
]]>
</script>
"

# Insert the script at the top of the file and save it.
tree.insert(0, ET.XML(script))
ET.ElementTree(tree).write('svg_tooltip.svg')

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

112.30 user_interfaces example code: toolmanager.py

[source code]

"This example demonstrates how to:
* Modify the Toolbar
* Create tools
* Add tools
* Remove tools
Using `matplotlib.backend_managers.ToolManager`
"

from __future__ import print_function
import matplotlib
matplotlib.use('GTK3Cairo')
matplotlib.rcParams['toolbar'] = 'toolmanager'
import matplotlib.pyplot as plt
from matplotlib.backend_tools import ToolBase, ToolToggleBase
from gi.repository import Gtk, Gdk

class ListTools(ToolBase):
    """List all the tools controlled by the ToolManager""
    # keyboard shortcut
default_keymap = 'm'
description = 'List Tools'
def trigger(self, *args, **kwargs):
    print('_' * 80)
    print('{0:12} {1:45} {2}'.format('Name (id)', 'Tool description', 'Keymap'))
    print('_' * 80)
tools = self.toolmanager.tools
for name in sorted(tools.keys()):
    if not tools[name].description:
        continue
    keys = ', '.join(sorted(self.toolmanager.get_tool_keymap(name)))
    print('{0:12} {1:45} {2}'.format(name, tools[name].description, keys))
    print('_' * 80)
print("Active Toggle tools")
print('{0:12} {1:45}'.format("Group", "Active"))
print('_' * 80)
for group, active in self.toolmanager.active_toggle.items():
    print('{0:12} {1:45}'.format(group, active))

class GroupHideTool(ToolToggleBase):
    """Hide lines with a given gid""
default_keymap = 'G'
description = 'Hide by gid'
def __init__(self, *args, **kwargs):
    self.gid = kwargs.pop('gid')
    ToolToggleBase.__init__(self, *args, **kwargs)
def enable(self, *args):
    self.set_lines_visibility(False)
def disable(self, *args):
    self.set_lines_visibility(True)
def set_lines_visibility(self, state):
    gr_lines = []
for ax in self.figure.get_axes():
    for line in ax.get_lines():
        if line.get_gid() == self.gid:
            line.set_visible(state)
self.figure.canvas.draw()

fig = plt.figure()
plt.plot([1, 2, 3], gid='mygroup')
plt.plot([2, 3, 4], gid='unknown')
plt.plot([3, 2, 1], gid='mygroup')

# Add the custom tools that we created
fig.canvas.manager.toolmanager.add_tool('List', ListTools)
fig.canvas.manager.toolmanager.add_tool('Hide', GroupHideTool, gid='mygroup')

# Add an existing tool to new group 'foo'.
# It can be added as many times as we want
fig.canvas.manager.toolbar.add_tool('zoom', 'foo')

# Remove the forward button
fig.canvas.manager.toolmanager.remove_tool('forward')

# To add a custom tool to the toolbar at specific location inside
# the navigation group
fig.canvas.manager.toolbar.add_tool('Hide', 'navigation', 1)

plt.show()
from numpy import arange, sin, pi

import wx

class CanvasFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1, 'CanvasFrame', size=(550, 350))

        self.SetBackgroundColour(wxc.NamedColour("WHITE"))

        self.figure = Figure()
        self.axes = self.figure.add_subplot(111)
        t = arange(0.0, 3.0, 0.01)
        s = sin(2*pi*t)

        self.axes.plot(t, s)
        self.axes.set_xlabel('t')
        self.axes.set_ylabel('sin(t)')

        self.figure_canvas = FigureCanvas(self, -1, self.figure)

        self.figure_canvas.mpl_connect('motion_notify_event', self.UpdateStatusBar)
        self.figure_canvas.Bind(wx.EVT_ENTER_WINDOW, self.ChangeCursor)

        self.sizer = wx.BoxSizer(wx.VERTICAL)
        self.sizer.Add(self.figure_canvas, 1, wx.LEFT | wx.TOP | wx.GROW)
        self.SetSizer(self.sizer)
        self.Fit()

        self.statusBar = wx.StatusBar(self, -1)
        self.SetStatusBar(self.statusBar)

        self.toolbar = NavigationToolbar2Wx(self.figure_canvas)
        self.sizer.Add(self.toolbar, 0, wx.LEFT | wx.EXPAND)
        self.toolbar.Show()

    def ChangeCursor(self, event):
        self.figure_canvas.SetCursor(wxc.StockCursor(wx.CURSOR_BULLSEYE))

    def UpdateStatusBar(self, event):
        if event.inaxes:
            x, y = event.xdata, event.ydata
            self.statusBar.SetStatusText(('x= ' + str(x) + ' y=' + str(y)), 0)

class App(wx.App):
    def OnInit(self):
        'Create the main window and insert the custom frame'
        frame = CanvasFrame()
self.SetTopWindow(frame)
frame.Show(True)
return True

if __name__ == '__main__':
    app = App(0)
    app.MainLoop()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)
113.1 widgets example code: buttons.py

```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.widgets import Button

freqs = np.arange(2, 20, 3)

fig, ax = plt.subplots()
plt.subplots_adjust(bottom=0.2)
t = np.arange(0.0, 1.0, 0.001)
s = np.sin(2*np.pi*freqs[0]*t)
l, = plt.plot(t, s, lw=2)

class Index(object):
    ind = 0

    def next(self, event):
        self.ind += 1
        i = self.ind % len(freqs)
ydata = np.sin(2*np.pi*freqs[i]*t)
l.set_ydata(ydata)
plt.draw()

    def prev(self, event):
        self.ind -= 1
        i = self.ind % len(freqs)
ydata = np.sin(2*np.pi*freqs[i]*t)
l.set_ydata(ydata)
plt.draw()

callback = Index()
axprev = plt.axes([0.7, 0.05, 0.1, 0.075])
axnext = plt.axes([0.81, 0.05, 0.1, 0.075])
bnext = Button(axnext, 'Next')
bnext.on_clicked(callback.next)
```

```python
bprev = Button(axprev, 'Previous')
bprev.on_clicked(callback.prev)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

### 113.2 widgets example code: check_buttons.py

[source code]

```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.widgets import CheckButtons

t = np.arange(0.0, 2.0, 0.01)
s0 = np.sin(2*np.pi*t)
s1 = np.sin(4*np.pi*t)
s2 = np.sin(6*np.pi*t)

fig, ax = plt.subplots()
l0, = ax.plot(t, s0, visible=False, lw=2)
l1, = ax.plot(t, s1, lw=2)
l2, = ax.plot(t, s2, lw=2)
plt.subplots_adjust(left=0.2)

rax = plt.axes([0.05, 0.4, 0.1, 0.15])
check = CheckButtons(rax, ('2 Hz', '4 Hz', '6 Hz'), (False, True, True))

def func(label):
    if label == '2 Hz':
        l0.set_visible(not l0.get_visible())
    elif label == '4 Hz':
        l1.set_visible(not l1.get_visible())
    elif label == '6 Hz':
        l2.set_visible(not l2.get_visible())
    plt.draw()
check.on_clicked(func)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

### 113.3 widgets example code: cursor.py

[source code]
from matplotlib.widgets import Cursor
import numpy as np
import matplotlib.pyplot as plt

fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, facecolor='#FFFFCC')

x, y = 4*(np.random.rand(2, 100) - .5)
ax.plot(x, y, 'o')
ax.set_xlim(-2, 2)
ax.set_ylim(-2, 2)

# set useblit = True on gtkagg for enhanced performance
cursor = Cursor(ax, useblit=True, color='red', linewidth=2)

plt.show()

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

113.4 widgets example code: lasso_selector_demo.py

[source code]

class SelectFromCollection(object):
    """Select indices from a matplotlib collection using 'LassoSelector'.

    Selected indices are saved in the 'ind' attribute. This tool highlights
    selected points by fading them out (i.e., reducing their alpha values).
    If your collection has alpha < 1, this tool will permanently alter them.

    Note that this tool selects collection objects based on their *origins*
    (i.e., 'offsets').

    Parameters
    ----------
    ax : :class:`matplotlib.axes.Axes`
        Axes to interact with.

    collection : :class:`matplotlib.collections.Collection` subclass
        Collection you want to select from.
alpha_other : $0 \leq \text{float} \leq 1$

To highlight a selection, this tool sets all selected points to an
alpha value of 1 and non-selected points to `alpha_other`.

```python
def __init__(self, ax, collection, alpha_other=0.3):
    self.canvas = ax.figure.canvas
    self.collection = collection
    self.alpha_other = alpha_other

    self.xys = collection.get_offsets()
    self.Npts = len(self.xys)

    # Ensure that we have separate colors for each object
    self.fc = collection.get_facecolors()
    if len(self.fc) == 0:
        raise ValueError('Collection must have a facecolor')
    elif len(self.fc) == 1:

    self.lasso = LassoSelector(ax, onselect=self.onselect)
    self.ind = []

def onselect(self, verts):
    path = Path(verts)
    self.ind = np.nonzero([path.contains_point(xy) for xy in self.xys])[0]
    self.fc[:, -1] = self.alpha_other
    self.fc[self.ind, -1] = 1
    self.collection.set_facecolors(self.fc)
    self.canvas.draw_idle()

def disconnect(self):
    self.lasso.disconnect_events()
    self.fc[:, -1] = 1
    self.collection.set_facecolors(self.fc)
    self.canvas.draw_idle()

if __name__ == '__main__':
    import matplotlib.pyplot as plt

    plt.ion()
    data = np.random.rand(100, 2)

    subplot_kw = dict(xlim=(0, 1), ylim=(0, 1), autoscale_on=False)
    fig, ax = plt.subplots(subplot_kw=subplot_kw)

    pts = ax.scatter(data[:, 0], data[:, 1], s=80)
    selector = SelectFromCollection(ax, pts)

    plt.draw()
    input('Press Enter to accept selected points')
```
print("Selected points:")
print(selector.xys[selector.ind])
selector.disconnect()

# Block end of script so you can check that the lasso is disconnected.
input('Press Enter to quit')

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

### 113.5 widgets example code: menu.py

[source code]

```python
from __future__ import division, print_function
import numpy as np
import matplotlib
import matplotlib.colors as colors
import matplotlib.patches as patches
import matplotlib.mathtext as mathtext
import matplotlib.pyplot as plt
import matplotlib.artist as artist
import matplotlib.image as image

class ItemProperties(object):
    def __init__(self, fontsize=14, labelcolor='black', bgcolor='yellow',
                 alpha=1.0):
        self.fontsize = fontsize
        self.labelcolor = labelcolor
        self.bgcolor = bgcolor
        self.alpha = alpha

        self.labelcolor_rgb = colors.to_rgba(labelcolor)[:3]
        self.bgcolor_rgb = colors.to_rgba(bgcolor)[:3]

class MenuItem(artist.Artist):
    parser = mathtext.MathTextParser("Bitmap")
    padx = 5
    pady = 5

    def __init__(self, fig, labelstr, props=None, hoverprops=None,
                 on_select=None):
        artist.Artist.__init__(self)

        self.set_figure(fig)
        self.labelstr = labelstr

        if props is None:
            props = ItemProperties()

    def _parse(self, s):
        self.parser.parse(s)

    def _update(self, s):
        self.parser.update(s)
```

113.5. widgets example code: menu.py 3005
if hoverprops is None:
    hoverprops = ItemProperties()

self.props = props
self.hoverprops = hoverprops
self.on_select = on_select

x, self.depth = self.parser.to_mask(
    labelstr, fontsize=props.fontsize, dpi=fig.dpi)

if props.fontsize != hoverprops.fontsize:
    raise NotImplementedError('support for different font sizes not implemented')

self.labelwidth = x.shape[1]
self.labelheight = x.shape[0]

self.labelArray = np.zeros((x.shape[0], x.shape[1], 4))
self.labelArray[:, :, -1] = x/255.

self.label = image.FigureImage(fig, origin='upper')
self.label.set_array(self.labelArray)

# we'll update these later
self.rect = patches.Rectangle((0, 0), 1, 1)
self.set_hover_props(False)

fig.canvas.mpl_connect('button_release_event', self.check_select)

def check_select(self, event):
    over, junk = self.rect.contains(event)
    if not over:
        return

    if self.on_select is not None:
        self.on_select(self)

def set_extent(self, x, y, w, h):
    print(x, y, w, h)
    self.rect.set_x(x)
    self.rect.set_y(y)
    self.rect.set_width(w)
    self.rect.set_height(h)

    self.label.ox = x + self.padx
    self.label.oy = y - self.depth + self.pady/2.

    self.rect._update_patch_transform()
    self.hover = False

def draw(self, renderer):

```python
def set_hover_props(self, b):
    if b:
        props = self.hoverprops
    else:
        props = self.props

    r, g, b = props.labelcolor_rgb
    self.labelArray[:, :, 0] = r
    self.labelArray[:, :, 1] = g
    self.labelArray[:, :, 2] = b
    self.label.set_array(self.labelArray)
    self.rect.set(facecolor=props.bgcolor, alpha=props.alpha)

def set_hover(self, event):
    'check the hover status of event and return true if status is changed'
    b, junk = self.rect.contains(event)

    changed = (b != self.hover)

    if changed:
        self.set_hover_props(b)

    self.hover = b
    return changed

class Menu(object):
    def __init__(self, fig, menuitems):
        self.figure = fig
        fig.suppressComposite = True

        self.menuitems = menuitems
        self.numitems = len(menuitems)

        maxw = max([item.labelwidth for item in menuitems])
        maxh = max([item.labelheight for item in menuitems])

        totalh = self.numitems * maxh + (self.numitems + 1) * 2 * MenuItem.pady

        x0 = 100
        y0 = 400

        width = maxw + 2 * MenuItem.padx
        height = maxh + MenuItem.padx

        for item in menuitems:
            left = x0
            bottom = y0 - maxh - MenuItem.pady

            item.set_extent(left, bottom, width, height)
```

113.5. widgets example code: menu.py
fig.artists.append(item)
y0 -= maxh + MenuItem.pady

fig.canvas.mpl_connect('motion_notify_event', self.on_move)

def on_move(self, event):
    draw = False
    for item in self.menuitems:
        draw = item.set_hover(event)
        if draw:
            self.figure.canvas.draw()
            break

fig = plt.figure()
fig.subplots_adjust(left=0.3)
props = ItemProperties(labelcolor='black', bgcolor='yellow',
                        fontsize=15, alpha=0.2)
hoverprops = ItemProperties(labelcolor='white', bgcolor='blue',
                             fontsize=15, alpha=0.2)

menuitems = []
for label in ('open', 'close', 'save', 'save as', 'quit'):
    def on_select(item):
        print('you selected %s' % item.labelstr)
    item = MenuItem(fig, label, props=props, hoverprops=hoverprops,
                     on_select=on_select)
    menuitems.append(item)

menu = Menu(fig, menuitems)
plt.show()
```python
ax2 = fig.add_subplot(212, sharex=ax1)
ax2.plot(t, s2)

multi = MultiCursor(fig.canvas, (ax1, ax2), color='r', lw=1)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

### 113.7 widgets example code: radio_buttons.py

[source code]

```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.widgets import RadioButtons

t = np.arange(0.0, 2.0, 0.01)
s0 = np.sin(2*np.pi*t)
s1 = np.sin(4*np.pi*t)
s2 = np.sin(8*np.pi*t)

fig, ax = plt.subplots()
l, = ax.plot(t, s0, lw=2, color='red')
plt.subplots_adjust(left=0.3)

axcolor = 'lightgoldenrodyellow'

rax = plt.axes([0.05, 0.7, 0.15, 0.15], facecolor=axcolor)
radio = RadioButtons(rax, ('2 Hz', '4 Hz', '8 Hz'))

def hzfunc(label):
    hzdict = {'2 Hz': s0, '4 Hz': s1, '8 Hz': s2}
ydata = hzdict[label]
l.set_ydata(ydata)
plt.draw()
radio.on_clicked(hzfunc)

rax = plt.axes([0.05, 0.4, 0.15, 0.15], facecolor=axcolor)
radio2 = RadioButtons(rax, ('red', 'blue', 'green'))

def colorfunc(label):
    l.set_color(label)
    plt.draw()
radio2.on_clicked(colorfunc)

rax = plt.axes([0.05, 0.1, 0.15, 0.15], facecolor=axcolor)
radio3 = RadioButtons(rax, ('-', '--', '-.', 'steps', ':'))

def stylefunc(label):
```

113.7. widgets example code: radio_buttons.py

3009
```python
from __future__ import print_function

Do a mouseclick somewhere, move the mouse to some destination, release the button. This class gives click- and release-events and also draws a line or a box from the click-point to the actual mouseposition (within the same axes) until the button is released. Within the method 'self.ignore()' it is checked whether the button from eventpress and eventrelease are the same.

```from matplotlib.widgets import RectangleSelector
import numpy as np
import matplotlib.pyplot as plt

def line_select_callback(eclick, erelease):
    'eclick and erelease are the press and release events'
    x1, y1 = eclick.xdata, eclick.ydata
    x2, y2 = erelease.xdata, erelease.ydata
    print("(%3.2f, %3.2f) --> (%3.2f, %3.2f)" % (x1, y1, x2, y2))
    print(" The button you used were: %s %s" % (eclick.button, erelease.button))

def toggle_selector(event):
    print(' Key pressed. ')
    if event.key in ['Q', 'q'] and toggle_selector.RS.active:
        print(' RectangleSelector deactivated. ')
        toggle_selector.RS.set_active(False)
    if event.key in ['A', 'a'] and not toggle_selector.RS.active:
        print(' RectangleSelector activated. ')
        toggle_selector.RS.set_active(True)

fig, current_ax = plt.subplots()
N = 100000
x = np.linspace(0.0, 10.0, N)
plt.plot(x, +np.sin(.2*np.pi*x), lw=3.5, c='b', alpha=.7)  # plot something
plt.plot(x, +np.cos(.2*np.pi*x), lw=3.5, c='r', alpha=.5)
```

Keywords: python, matplotlib, pylab, example, codex (see Search examples)

113.8 widgets example code: rectangle_selector.py

[source code]
Keywords: python, matplotlib, pylab, example, codex (see Search examples)

113.9 widgets example code: slider_demo.py

```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.widgets import Slider, Button, RadioButtons

fig, ax = plt.subplots()
plt.subplots_adjust(left=0.25, bottom=0.25)
t = np.arange(0.0, 1.0, 0.001)
a0 = 5
f0 = 3
s = a0*np.sin(2*np.pi*f0*t)
l, = plt.plot(t, s, lw=2, color='red')
plt.axis([0, 1, -10, 10])
axcolor = 'lightgoldenrodyellow'
axfreq = plt.axes([0.25, 0.1, 0.65, 0.03], facecolor=axcolor)
sfreq = Slider(axfreq, 'Freq', 0.1, 30.0, valinit=f0)
samp = Slider(axamp, 'Amp', 0.1, 10.0, valinit=a0)

def update(val):
    amp = samp.val
    freq = sfreq.val
    l.set_ydata(amp*np.sin(2*np.pi*freq*t))
    fig.canvas.draw_idle()
sfreq.on_changed(update)
samp.on_changed(update)

resetax = plt.axes([0.8, 0.025, 0.1, 0.04])
button = Button(resetax, 'Reset', color=axcolor, hovercolor='0.975')
```

113.9. widgets example code: slider_demo.py 3011
def reset(event):
    sfreq.reset()
    samp.reset()
button.on_clicked(reset)

rax = plt.axes([0.025, 0.5, 0.15, 0.15], facecolor=axcolor)
radio = RadioButtons(rax, ('red', 'blue', 'green'), active=0)

def colorfunc(label):
    l.set_color(label)
    fig.canvas.draw_idle()
radio.on_clicked(colorfunc)
plt.show()
thisy = y[indmin:indmax]
line2.set_data(thisx, thisy)
ax2.set_xlim(thisx[0], thisx[-1])
ax2.set_ylim(thisy.min(), thisy.max())
fig.canvas.draw()

# set useblit True on gtkagg for enhanced performance
span = SpanSelector(ax, onselect, 'horizontal', useblit=True,
rectprops=dict(alpha=0.5, facecolor='red'))

plt.show()
Part X

Glossary
AGG  The Anti-Grain Geometry (Agg) rendering engine, capable of rendering high-quality images

Cairo  The Cairo graphics engine

dateutil  The dateutil library provides extensions to the standard datetime module

EPS  Encapsulated Postscript (EPS)

FreeType  FreeType is a font rasterization library used by matplotlib which supports TrueType, Type 1, and OpenType fonts.

GDK  The Gimp Drawing Kit for GTK+

GTK  The GIMP Toolkit (GTK) graphical user interface library

JPG  The Joint Photographic Experts Group (JPEG) compression method and file format for photographic images

numpy  numpy is the standard numerical array library for python, the successor to Numeric and numarray. numpy provides fast operations for homogeneous data sets and common mathematical operations like correlations, standard deviation, fourier transforms, and convolutions.

PDF  Adobe’s Portable Document Format (PDF)

PNG  Portable Network Graphics (PNG), a raster graphics format that employs lossless data compression which is more suitable for line art than the lossy jpg format. Unlike the gif format, png is not encumbered by requirements for a patent license.

PS  Postscript (PS) is a vector graphics ASCII text language widely used in printers and publishing. Postscript was developed by adobe systems and is starting to show its age: for example is does not have an alpha channel. PDF was designed in part as a next-generation document format to replace postscript

pygtk  pygtk provides python wrappers for the GTK widgets library for use with the GTK or GTKAgg backend. Widely used on linux, and is often packages as ‘python-gtk2’

pyqt  pyqt provides python wrappers for the Qt widgets library and is required by the matplotlib Qt5Agg and Qt4Agg backends. Widely used on linux and windows; many linux distributions package this as ‘python-qt5’ or ‘python-qt4’.

python  python is an object oriented interpreted language widely used for scripting, application development, web application servers, scientific computing and more.

pytz  pytz provides the Olson tz database in Python. it allows accurate and cross platform timezone calculations and solves the issue of ambiguous times at the end of daylight savings

Qt  Qt is a cross-platform application framework for desktop and embedded development.

Qt4  Qt4 is the previous, but most widely used, version of Qt cross-platform application framework for desktop and embedded development.

Qt5  Qt5 is the current version of Qt cross-platform application framework for desktop and embedded development.

raster graphics  Raster graphics, or bitmaps, represent an image as an array of pixels which is resolution dependent. Raster graphics are generally most practical for photo-realistic images, but do not scale easily without loss of quality.
SVG  The Scalable Vector Graphics format (SVG). An XML based vector graphics format supported by many web browsers.

TIFF  Tagged Image File Format (TIFF) is a file format for storing images, including photographs and line art.

Tk  Tk is a graphical user interface for Tcl and many other dynamic languages. It can produce rich, native applications that run unchanged across Windows, Mac OS X, Linux and more.

vector graphics  vector graphics use geometrical primitives based upon mathematical equations to represent images in computer graphics. Primitives can include points, lines, curves, and shapes or polygons. Vector graphics are scalable, which means that they can be resized without suffering from issues related to inherent resolution like are seen in raster graphics. Vector graphics are generally most practical for typesetting and graphic design applications.

wxpython  wxpython provides python wrappers for the wxWidgets library for use with the WX and WX-Agg backends. Widely used on linux, OS-X and windows, it is often packaged by linux distributions as ‘python-wxgtk’

wxWidgets  WX is cross-platform GUI and tools library for GTK, MS Windows, and MacOS. It uses native widgets for each operating system, so applications will have the look-and-feel that users on that operating system expect.
BIBLIOGRAPHY

[colorcet] https://github.com/bokeh/colorcet


[list-colormaps] https://gist.github.com/endolith/2719900#id7

[mycarta-banding] https://mycarta.wordpress.com/2012/10/14/the-rainbow-is-deadlong-live-the-rainbow-part-4-cie-lab-heated-body/


[mycarta-lablinear] https://mycarta.wordpress.com/2012/12/06/the-rainbow-is-deadlong-live-the-rainbow-part-5-cie-lab-linear/

[mycarta-cubelaw] https://mycarta.wordpress.com/2013/02/21/perceptual-rainbow-palette-the-method/


matplotlib.afm, 487
matplotlib.animation, 491
matplotlib.artist, 536
matplotlib.axis, 794
matplotlib.backend_bases, 913
matplotlib.backend_managers, 934
matplotlib.backend_tools, 937
matplotlib.backends.backend_pdf, 948
matplotlib.backends.backend_qt4agg, 946
matplotlib.backends.backend_qt5agg, 946
matplotlib.backends.backend_svg, 951
matplotlib.backends.backend_wxagg, 947
matplotlib.cbook, 953
matplotlib.cm, 971
matplotlib.collections, 975
matplotlib.colorbar, 1145
matplotlib.colors, 1151
matplotlib.container, 1165
matplotlib.dates, 1167
matplotlib.dviread, 1181
matplotlib.figure, 1185
matplotlib.finance, 1207
matplotlib.font_manager, 1219
matplotlib.fontconfig_pattern, 1225
matplotlib.gridspec, 1227
matplotlib.image, 1231
matplotlib.legend, 1237
matplotlib.legend_handler, 1241
matplotlib.lines, 1245
matplotlib.markers, 1255
matplotlib.mathtext, 1261
matplotlib.mlab, 1279
matplotlib.offsetbox, 1313
matplotlib.patches, 1325
matplotlib.path, 1367
matplotlib.pathEffects, 92
matplotlib.projections, 1379
matplotlib.projections.polar, 1380
matplotlib.pyplot, 1387
matplotlib.rcsetup, 1593
matplotlib.sankey, 1597
matplotlib.scale, 1605
matplotlib.sphinxext.plot_directive, 1922
matplotlib.spines, 1615
matplotlib.style, 1619
matplotlib.text, 1621
matplotlib.ticker, 1635
matplotlib.tight_layout, 1647
matplotlib.transforms, 1649
matplotlib.tri, 1671
matplotlib.type1font, 1683
matplotlib.units, 1685
matplotlib.widgets, 1687
mpl_toolkits.axes_grid.axes_size, 1795
mpl_toolkits.axes_grid1.anchored_artists, 1805
mpl_toolkits.axes_grid1.inset_locator, 1811
mpl_toolkits.mplot3d.art3d, 1871
mpl_toolkits.mplot3d.axes3d, 1848
mpl_toolkits.mplot3d.axes3d, 1870
mpl_toolkits.mplot3d.proj3d, 1876
Symbols

__call__() (matplotlib.tri.CubicTriInterpolator method), 1675
__call__() (matplotlib.tri.LinearTriInterpolator method), 1673
__call__() (matplotlib.tri.TrapezoidMapTriFinder method), 1672
__init__() (matplotlib.animation.AVConvBase method), 530
__init__() (matplotlib.animation.AVConvFileWriter method), 521
__init__() (matplotlib.animation.AVConvWriter method), 517
__init__() (matplotlib.animation.Animation method), 525
__init__() (matplotlib.animation.ArtistAnimation method), 493
__init__() (matplotlib.animation.FFMpegBase method), 530
__init__() (matplotlib.animation.FFMpegFileWriter method), 519
__init__() (matplotlib.animation.FFMpegWriter method), 515
__init__() (matplotlib.animation.FileMovieWriter method), 529
__init__() (matplotlib.animation.FuncAnimation method), 493
__init__() (matplotlib.animation.ImageMagickBase method), 530
__init__() (matplotlib.animation.ImageMagickFileWriter method), 516
__init__() (matplotlib.animation.ImageMagickWriter method), 520
__init__() (matplotlib.animation.MencoderBase method), 532
__init__() (matplotlib.animation.MencoderFileWriter method), 532
__init__() (matplotlib.animation.MencoderWriter method), 533
__init__() (matplotlib.animation.MovieWriter method), 528
__init__() (matplotlib.animation.MovieWriterRegistry method), 526
__init__() (matplotlib.animation.TimedAnimation method), 526
__init__() (matplotlib.artist.ArtistInspector method), 553

A

AbstractPathEffect (class in matplotlib.path.effects), 1375
Accent (class in matplotlib.mathtext), 1261
accent() (matplotlib.mathtext.Parser method), 1272
acorr() (in module matplotlib.pyplot), 1387
acorr() (matplotlib.axes.Axes method), 608
active (matplotlib.widgets.Widget attribute), 1701
active_toggle (matplotlib.figure.Figure attribute), 934
add() (matplotlib.figure.AxesStack method), 1185
add() (matplotlib.sankey.Sankey method), 1601
add_artist() (matplotlib.figure.Figure method), 1188
add_artist() (matplotlib.axis.Axis method), 878
add_artist() (mpl_toolkits.mplot3d.proj3d.Perspective method), 1796
add_auto_adjustable_area() (mpl_toolkits.axes_grid.anchors.Anchors method), 1318
add_axis() (mpl_toolkits.axes_grid1.anchoring.AxesAnchoring method), 1796
add_callback() (matplotlib.artist.Artist method), 537
add_callback() (matplotlib.figure.Figure method), 1188
add_callback() (matplotlib.figure.Figure method), 1188
add_callback() (matplotlib.figure.Figure method), 1188
add_callback() (matplotlib.figure.Figure method), 1188
add_callback() (matplotlib.figure.Figure method), 1188
add_callback() (matplotlib.axis.Tick method), 883
add_callback() (matplotlib.axis.XAxis method), 889
add_callback() (matplotlib.axis.YAxis method), 900
add_callback() (matplotlib.axis.YTick method), 861
add_callback() (matplotlib.backend_bases.TimerBase method), 932
add_callback() (matplotlib.collections.AsteriskPolygonCollection method), 976
add_callback() (matplotlib.collections.BrokenBarHCollection method), 988
add_callback() (matplotlib.collections.CircleCollection method), 1000
add_callback() (matplotlib.collections.EllipseCollection method), 1024
add_callback() (matplotlib.collections.EventCollection method), 1037
add_callback() (matplotlib.collections.LineCollection method), 1050
add_callback() (matplotlib.collections.PatchCollection method), 1061
add_callback() (matplotlib.collections.PolyCollection method), 1085
add_callback() (matplotlib.collections.QuadMesh method), 1097
add_callback() (matplotlib.collections.RegularPolyCollection method), 1109
add_callback() (matplotlib.collections.StarPolygonCollection method), 1122
add_callback() (matplotlib.collections.TriMesh method), 1133
add_collection() (mpl_toolkits.mplot3d.Axes3D method), 1840
add_collection3d() (mpl_toolkits.mplot3d.Axes3D method), 1848
add_container() (matplotlib.axes.Axes method), 765
add_checker() (matplotlib.cm.ScalarMappable method), 971
add_checker() (matplotlib.collections.AsteriskPolygonCollection method), 976
add_checker() (matplotlib.collections.BrokenBarHCollection method), 988
add_checker() (matplotlib.collections.CircleCollection method), 1000
add_checker() (matplotlib.collections.EllipseCollection method), 1024
add_checker() (matplotlib.collections.EventCollection method), 1037
add_checker() (matplotlib.collections.LineCollection method), 1050
add_checker() (matplotlib.collections.PatchCollection method), 1061
add_checker() (matplotlib.collections.PolyCollection method), 1085
add_checker() (matplotlib.collections.QuadMesh method), 1097
add_checker() (matplotlib.collections.RegularPolyCollection method), 1109
add_checker() (matplotlib.collections.StarPolygonCollection method), 1122
add_checker() (matplotlib.collections.TriMesh method), 1133
Index
auto_scale_xyz() (mpl_toolkits.mplot3d.axes3d.Axes3D
method), 1848
AutoDateFormatter (class in matplotlib.dates), 1170
AutoDateLocator (class in matplotlib.dates), 1172
autofmt_xdate() (matplotlib.figure.Figure method),
1190
AutoHeightChar (class in matplotlib.mathtext), 1261
AutoLocator (class in matplotlib.ticker), 1644
AutoMinorLocator (class in matplotlib.ticker), 1645
autoscale() (in module matplotlib.pyplot), 1396
autoscale() (matplotlib.axes.Axes method), 752
autoscale() (matplotlib.cm.ScalarMappable method),
971
autoscale() (matplotlib.collections.AsteriskPolygonCollection
method), 976
autoscale() (matplotlib.collections.BrokenBarHCollections
autoscale() (matplotlib.collections.CircleCollection
method), 1000
autoscale() (matplotlib.collections.Collection method), 1013
autoscale() (matplotlib.collections.EllipseCollection
method), 1024
autoscale() (matplotlib.collections.EventCollection
method), 1037
autoscale() (matplotlib.collections.LineCollection
method), 1050
autoscale() (matplotlib.collections.PatchCollection
method), 1061
autoscale() (matplotlib.collections.PathCollection
method), 1073
autoscale() (matplotlib.collections.PolyCollection
method), 1085
autoscale() (matplotlib.collections.QuadMesh method), 1097
autoscale() (matplotlib.collections.RegularPolyCollection
method), 1109
autoscale() (matplotlib.collections.StarPolygonCollection
method), 1122
autoscale() (matplotlib.collections.TriMesh method), 1133
autoscale() (matplotlib.colors.LogNorm method), 1159
autoscale() (matplotlib.colors.Normalize method), 1160
autoscale() (matplotlib.colors.PowerNorm method), 1161
autoscale() (matplotlib.colors.SymLogNorm method), 1161
autoscale() (matplotlib.dates.AutoDateLocator method), 1172
autoscale() (matplotlib.dates.RRuleLocator method),
1171
autoscale() (matplotlib.dates.YearLocator method),
1173
autoscale() (matplotlib.dates, 1642
autoscale() (mpl_toolkits.mplot3d.axes3d.Axes3D
autoscale() (matplotlib.colors.SymLogNorm
method), 1161
autoscale() (matplotlib.colors.Normalize method),
1160
autoscale() (matplotlib.colors.PowerNorm method),
1161
autoscale(autoscale(autoscale(None) (matplotlib.collections.AsteriskPolygonCollection
method), 976
autoscale(autoscale(autoscale(None) (matplotlib.collections.BrokenBarHCollections
autoscale(autoscale(autoscale(None) (matplotlib.collections.CircleCollection
method), 988
autoscale(autoscale(autoscale(None) (matplotlib.collections.Collection method), 1013
autoscale(autoscale(autoscale(None) (matplotlib.collections.EllipseCollection
method), 1024
autoscale(autoscale(autoscale(None) (matplotlib.collections.EventCollection
method), 1037
autoscale(autoscale(autoscale(None) (matplotlib.collections.LineCollection
method), 1050
autoscale(autoscale(autoscale(None) (matplotlib.collections.PatchCollection
method), 1061
autoscale(autoscale(autoscale(None) (matplotlib.collections.PathCollection
method), 1073
autoscale(autoscale(autoscale(None) (matplotlib.collections.PolyCollection
method), 1085
autoscale(autoscale(autoscale(None) (matplotlib.collections.QuadMesh method), 1097
autoscale(autoscale(autoscale(None) (matplotlib.collections.RegularPolyCollection
method), 1109
autoscale(autoscale(autoscale(None) (matplotlib.collections.StarPolygonCollection
method), 1122
autoscale(autoscale(autoscale(None) (matplotlib.collections.TriMesh method), 1133
autoscale(autoscale(autoscale(None) (matplotlib.colors.LogNorm method), 1159
autoscale(autoscale(autoscale(None) (matplotlib.colors.Normalize method),
autoscale(autoscale(autoscale(None) (matplotlib.colors.PowerNorm method),
autoscale(autoscale(autoscale(None) (matplotlib.colors.SymLogNorm
 method), 1161
<table>
<thead>
<tr>
<th>Function/Method</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>autoscale_None() (matplotlib.collections.TriMesh method)</td>
<td>1133</td>
</tr>
<tr>
<td>autoscale_None() (matplotlib.colors.LogNorm method)</td>
<td>1160</td>
</tr>
<tr>
<td>autoscale_None() (matplotlib.colors.Normalize method)</td>
<td>1160</td>
</tr>
<tr>
<td>autoscale_None() (matplotlib.colors.PowerNorm method)</td>
<td>1161</td>
</tr>
<tr>
<td>autoscale_None() (matplotlib.colors.SymLogNorm method)</td>
<td>1161</td>
</tr>
<tr>
<td>autoscale_view() (matplotlib.axes.Axes method)</td>
<td>753</td>
</tr>
<tr>
<td>autoscale_view() (mpl_toolkits.mplot3d.axes3d.Axes3D method)</td>
<td>1849</td>
</tr>
<tr>
<td>AutoWidthChar (class in matplotlib.mathtext)</td>
<td>1261</td>
</tr>
<tr>
<td>autumn() (in module matplotlib.pyplot)</td>
<td>1396</td>
</tr>
<tr>
<td>AuxTransformBox (class in matplotlib.offsetbox)</td>
<td>1316</td>
</tr>
<tr>
<td>available() (matplotlib.widgets.LockDraw method)</td>
<td>1693</td>
</tr>
<tr>
<td>AVConvBase (class in matplotlib.animation)</td>
<td>530</td>
</tr>
<tr>
<td>AVConvFileWriter (class in matplotlib.animation)</td>
<td>521</td>
</tr>
<tr>
<td>AVConvWriter (class in matplotlib.animation)</td>
<td>517</td>
</tr>
<tr>
<td>ax (matplotlib.colorbar.ColorbarBase attribute)</td>
<td>1146</td>
</tr>
<tr>
<td>Axes (class in matplotlib.axes)</td>
<td>555</td>
</tr>
<tr>
<td>axes (matplotlib.artist.Artist attribute)</td>
<td>547</td>
</tr>
<tr>
<td>axes (matplotlib.axes.Axes attribute)</td>
<td>785</td>
</tr>
<tr>
<td>axes (matplotlib.axis.Axis attribute)</td>
<td>878</td>
</tr>
<tr>
<td>axes (matplotlib.axis.Tick attribute)</td>
<td>838</td>
</tr>
<tr>
<td>axes (matplotlib.axis.XAxis attribute)</td>
<td>890</td>
</tr>
<tr>
<td>axes (matplotlib.axis.XTick attribute)</td>
<td>850</td>
</tr>
<tr>
<td>axes (matplotlib.axis.YAxis attribute)</td>
<td>901</td>
</tr>
<tr>
<td>axes (matplotlib.axis.YTick attribute)</td>
<td>861</td>
</tr>
<tr>
<td>axes (matplotlib.collections.AsteriskPolygonCollection attribute)</td>
<td>976</td>
</tr>
<tr>
<td>axes (matplotlib.collections.BrokenBarHCollection attribute)</td>
<td>988</td>
</tr>
<tr>
<td>axes (matplotlib.collections.CircleCollection attribute)</td>
<td>1000</td>
</tr>
<tr>
<td>axes (matplotlib.collections.Collection attribute)</td>
<td>1013</td>
</tr>
<tr>
<td>axes (matplotlib.collections.EllipseCollection attribute)</td>
<td>1024</td>
</tr>
<tr>
<td>axes (matplotlib.collections.EventCollection attribute)</td>
<td>1037</td>
</tr>
<tr>
<td>axes (matplotlib.collections.LineCollection attribute)</td>
<td>1050</td>
</tr>
<tr>
<td>axes (matplotlib.collections.PatchCollection attribute)</td>
<td>1062</td>
</tr>
<tr>
<td>axes (matplotlib.collections.PathCollection attribute)</td>
<td>1073</td>
</tr>
<tr>
<td>axes (matplotlib.collections.PolyCollection attribute)</td>
<td>1085</td>
</tr>
<tr>
<td>axes (matplotlib.collections.QuadMesh attribute)</td>
<td>1097</td>
</tr>
<tr>
<td>axes (matplotlib.collections.RegularPolyCollection attribute)</td>
<td>1109</td>
</tr>
<tr>
<td>axes (matplotlib.collections.StarPolygonCollection attribute)</td>
<td>1122</td>
</tr>
<tr>
<td>axes (matplotlib.collections.TriMesh attribute)</td>
<td>1133</td>
</tr>
<tr>
<td>axes (matplotlib.figure.Figure attribute)</td>
<td>1246</td>
</tr>
<tr>
<td>axes (matplotlib.figure.FigureAttribute)</td>
<td>1190</td>
</tr>
<tr>
<td>axes (matplotlib.figure.Lines2DAttribute)</td>
<td>1190</td>
</tr>
<tr>
<td>axes (matplotlib.figure.OffsetBoxAttribute)</td>
<td>1190</td>
</tr>
<tr>
<td>axes (matplotlib.figure.PyplotAttribute)</td>
<td>1190</td>
</tr>
<tr>
<td>axes (matplotlib.figure.PyplotAttribute)</td>
<td>1190</td>
</tr>
<tr>
<td>AxesDivider (class in mpl_toolkits.axes_grid.axes_divider)</td>
<td>1799</td>
</tr>
<tr>
<td>AxesImage (class in matplotlib.image)</td>
<td>1231</td>
</tr>
<tr>
<td>AxesLocator (class in mpl_toolkits.axes_grid.axes_divider)</td>
<td>1798</td>
</tr>
<tr>
<td>AxesStack (class in matplotlib.figure)</td>
<td>1848</td>
</tr>
<tr>
<td>AxesWidget (class in matplotlib.widgets)</td>
<td>1848</td>
</tr>
<tr>
<td>AxesX (class in mpl_toolkits.axes_grid.axes_size)</td>
<td>1795</td>
</tr>
<tr>
<td>AxesY (class in mpl_toolkits.axes_grid.axes_size)</td>
<td>1795</td>
</tr>
<tr>
<td>axhline() (module matplotlib.pyplot)</td>
<td>1397</td>
</tr>
<tr>
<td>axhspan() (module matplotlib.pyplot)</td>
<td>1399</td>
</tr>
<tr>
<td>axhspan() (matplotlib.axes.Axes method)</td>
<td>601</td>
</tr>
<tr>
<td>Axis (class in matplotlib.axis)</td>
<td>749</td>
</tr>
<tr>
<td>Axis (class in mpl_toolkits.mplot3d.axis3d)</td>
<td>1870</td>
</tr>
<tr>
<td>axis() (matplotlib.axes.Axes method)</td>
<td>732</td>
</tr>
<tr>
<td>axis() (matplotlib.axis.Axis method)</td>
<td>800</td>
</tr>
<tr>
<td>axis() (matplotlib.axis.XAxis method)</td>
<td>824</td>
</tr>
<tr>
<td>axis() (matplotlib.axis.YAxis method)</td>
<td>815</td>
</tr>
<tr>
<td>axis_name (matplotlib.axis.XAxis attribute)</td>
<td>805</td>
</tr>
</tbody>
</table>
axis_name (matplotlib.axis.YAxis attribute), 803
AxisArtist (class in mpl_toolkits.axes_grid1.axis_artist), 1802
AxisInfo (class in matplotlib.units), 1685
axisinfo() (matplotlib.units.ConversionInterface static method), 1686
AxisLabel (class in mpl_toolkits.axes_grid1.axis_artist), 1804
AxisScaleBase (class in matplotlib.backend_tools), 937
axvline() (in module matplotlib.pyplot), 1402
axvline() (matplotlib.axes.Axes method), 605
axvspan() (in module matplotlib.pyplot), 1404
axvspan() (matplotlib.axes.Axes method), 607
B
back() (matplotlib.backend_bases.NavigationToolbar2 method), 925
back() (matplotlib.backend_tools.ToolViewsPositions method), 943
back() (matplotlib.cbook.Stack method), 957
BakomaFonts (class in matplotlib.mathtext), 1261
bart() (in module matplotlib.pyplot), 1405
bar() (matplotlib.axes.Axes method), 583
bar() (mpl_toolkits.mplot3d.Axes3D method), 1841
bar() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 1849
bar3d() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 1849
barbs() (in module matplotlib.pyplot), 1408
barbs() (matplotlib.axes.Axes method), 721
BarContainer (class in matplotlib.container), 1165
barh() (in module matplotlib.pyplot), 1412
barh() (matplotlib.axes.Axes method), 586
base (matplotlib.scale.InvertedLog10Transform attribute), 1605
base (matplotlib.scale.InvertedLog2Transform attribute), 1605
base (matplotlib.scale.InvertedNaturalLogTransform attribute), 1608
base (matplotlib.scale.Log10Transform attribute), 1605
base (matplotlib.scale.Log2Transform attribute), 1608
base (matplotlib.scale.LogNaturalLogTransform attribute), 1609
base (matplotlib.scale.NaturalLogTransform attribute), 1611
base() (matplotlib.ticker.LogFormatter method), 1641
base() (matplotlib.ticker.LogLocator method), 1644
base_repr() (in module matplotlib.mlab), 1286
basepath (matplotlib.mathtext.StandardPsFonts attribute), 1275
Bbox (class in matplotlib.transforms), 1654
bbox_artist() (in module matplotlib.offsetbox), 1323
bbox_artist() (in module matplotlib.patches), 1365
BboxBase (class in matplotlib.transforms), 1651
BboxConnector (class in mpl_toolkits.axes_grid1.inset_locator), 1811
BboxConnectorPatch (class in mpl_toolkits.axes_grid1.inset_locator), 1814
BboxImage (class in matplotlib.image), 1231
BboxPatch (class in mpl_toolkits.axes_grid1.inset_locator), 1815
BboxTransform (class in matplotlib.transforms), 1668
BboxTransformFrom (class in matplotlib.transforms), 1669
BboxTransformTo (class in matplotlib.transforms), 1669
bin_boundaries() (matplotlib.ticker.MaxNLocator method), 1645
binary_repr() (in module matplotlib.mlab), 1286
binom() (matplotlib.mathtext.Parser method), 1272
bivariate_normal() (in module matplotlib.mlab), 1286
blend_hsv() (matplotlib.colors.LightSource method), 1154
blend_overlay() (matplotlib.colors.LightSource method), 1154
blend_soft_light() (matplotlib.colors.LightSource method), 1155
blended_transform_factory() (in module matplotlib.transforms), 1667
BlendedAffine2D (class in matplotlib.transforms), 1666
BlendedGenericTransform (class in matplotlib.transforms), 1666
blit() (matplotlib.backend_bases.FigureCanvasBase method), 914
blit() (matplotlib.backends.backend_qt5agg.FigureCanvas method), 947
blit() (matplotlib.backends.backend_wxagg.FigureCanvas method), 948
bone() (in module matplotlib.pyplot), 1414
BoundaryNorm (class in matplotlib.colors), 1152
bounds (matplotlib.transforms.BboxBase attribute), 1651
Box (class in matplotlib.mathtext), 1262
box() (in module matplotlib.pyplot), 1414
boxplot() (in module matplotlib.pyplot), 1415
boxplot() (matplotlib.axes.Axes method), 636
boxplot_stats() (in module matplotlib.cbook), 958
BoxStyle (class in matplotlib.patches), 1333
BoxStyle.Circle (class in matplotlib.patches), 1334
BoxStyle.DArrow (class in matplotlib.patches), 1335
BoxStyle.LArrow (class in matplotlib.patches), 1335
BoxStyle.RArrow (class in matplotlib.patches), 1335
BoxStyle.Round (class in matplotlib.patches), 1335
BoxStyle.Round4 (class in matplotlib.patches), 1335
BoxStyle.Roundtooth (class in matplotlib.patches), 1335
BoxStyle.Sawtooth (class in matplotlib.patches), 1336
BoxStyle.Square (class in matplotlib.patches), 1336
broken_barh() (in module matplotlib.pyplot), 1420
broken_barh() (matplotlib.axes.Axes method), 595
BrokenBarHCollection (class in matplotlib.collections), 988
bubble() (matplotlib.cbook.Stack method), 957
bubble() (matplotlib.figure.AxesStack method), 1185
Bunch (class in matplotlib.pyplot), 1414
Button (class in matplotlib.widgets), 1687
button (matplotlib.backend_bases.MouseEvent attribute), 924
button_press_event() (matplotlib.backend_bases.FigureCanvasBase method), 914
button_release_event() (matplotlib.backend_bases.FigureCanvasBase method), 914
bxp() (matplotlib.axes.Axes method), 644
byAttribute() (matplotlib.cbook.Sorter method), 956
byItem() (matplotlib.cbook.Sorter method), 956
C
c_over_c() (matplotlib.mathtext.Parser method), 1272
Cairo, 3017
calculate_plane_coefficients() (matplotlib.tri.Triangulation method), 1671
CallbackRegistry (class in matplotlib.cbook), 953
can_pan() (matplotlib.axes.Axes method), 770
can_pan() (matplotlib.projections.polar.PolarAxes method), 1382
can_pan() (mpl_toolkits.mplot3d.axes3d.axes3d.Axes3D method), 1849
can_zoom() (matplotlib.axes.Axes method), 771
can_zoom() (matplotlib.projections.polar.PolarAxes method), 1382
can_zoom() (mpl_toolkits.mplot3d.axes3d.axes3d.Axes3D method), 1850
candlestick2_ochl() (in module matplotlib.finance), 1207
candlestick2_ohlc() (in module matplotlib.finance), 1208
candlestick_ochl() (in module matplotlib.finance), 1209
candlestick_ohlc() (in module matplotlib.finance), 1209
center (matplotlib.widgets.RectangleSelector attribute), 1696
center() (matplotlib.mlab.PCA method), 1284
center_matrix() (in module matplotlib.mlab), 1286
change_geometry() (mpl_toolkits.axes_grid.axes_divider.SubplotDivider method), 1799
changed() (matplotlib.cm.ScalarMappable method), 971
changed() (matplotlib.collections.AsteriskPolygonCollection method), 976
changed() (matplotlib.collections.BrokenBarHCollection method), 988
changed() (matplotlib.collections.CircleCollection method), 1000
changed() (matplotlib.collections.Collection method), 1013
<table>
<thead>
<tr>
<th>Function/Method</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>clear() (matplotlib.transforms.Affine2D method)</td>
<td>1662</td>
</tr>
<tr>
<td>clear() (matplotlib.widgets.Cursor method)</td>
<td>1690</td>
</tr>
<tr>
<td>clear() (matplotlib.widgets.MultiCursor method)</td>
<td>1694</td>
</tr>
<tr>
<td>clf() (in module matplotlib.pyplot)</td>
<td>1429</td>
</tr>
<tr>
<td>clf() (matplotlib.figure.Figure method)</td>
<td>1190</td>
</tr>
<tr>
<td>clim() (in module matplotlib.pyplot)</td>
<td>1429</td>
</tr>
<tr>
<td>clip_children (matplotlib.offsetbox.DrawingArea attribute)</td>
<td>1318</td>
</tr>
<tr>
<td>clip_to_bbox() (matplotlib.path.Path method)</td>
<td>1369</td>
</tr>
<tr>
<td>close() (in module matplotlib.pyplot)</td>
<td>1429</td>
</tr>
<tr>
<td>close() (matplotlib.backends.backend_pdf.PdfPages method)</td>
<td>950</td>
</tr>
<tr>
<td>close() (matplotlib.dviread.Dvi method)</td>
<td>1181</td>
</tr>
<tr>
<td>close_event() (matplotlib.backend_bases.FigureCanvasBase method)</td>
<td>914</td>
</tr>
<tr>
<td>close_group() (matplotlib.backend_bases.RendererBase method)</td>
<td>927</td>
</tr>
<tr>
<td>CloseEvent (class in matplotlib.backend_bases)</td>
<td>913</td>
</tr>
<tr>
<td>CLOSEPOLY (matplotlib.path.Path attribute)</td>
<td>1368</td>
</tr>
<tr>
<td>closest() (matplotlib.widgets.ToolHandles method)</td>
<td>1700</td>
</tr>
<tr>
<td>cm_fallback (matplotlib.mathtext.StixFonts attribute)</td>
<td>1275</td>
</tr>
<tr>
<td>cmap (matplotlib.cm.ScalarMappable attribute)</td>
<td>971</td>
</tr>
<tr>
<td>code_type (matplotlib.path.Path attribute)</td>
<td>1370</td>
</tr>
<tr>
<td>codes (matplotlib.legend.Legend attribute)</td>
<td>1239</td>
</tr>
<tr>
<td>codes (matplotlib.path.Path attribute)</td>
<td>1370</td>
</tr>
<tr>
<td>cohere() (in module matplotlib.mlab)</td>
<td>1286</td>
</tr>
<tr>
<td>cohere() (in module matplotlib.pyplot)</td>
<td>1430</td>
</tr>
<tr>
<td>cohore() (in module matplotlib.axes.Axes method)</td>
<td>613</td>
</tr>
<tr>
<td>cohore_pairs() (in module matplotlib.mlab)</td>
<td>1287</td>
</tr>
<tr>
<td>Collection (class in matplotlib.collections)</td>
<td>1011</td>
</tr>
<tr>
<td>Colorbar (class in matplotlib.colorbar)</td>
<td>1145</td>
</tr>
<tr>
<td>colorbar (matplotlib.cm.ScalarMappable attribute)</td>
<td>971</td>
</tr>
<tr>
<td>colorbar() (in module matplotlib.pyplot)</td>
<td>1433</td>
</tr>
<tr>
<td>colorbar() (matplotlib.figure.Figure method)</td>
<td>1190</td>
</tr>
<tr>
<td>colorbar_extend (matplotlib.colors.Colormap attribute)</td>
<td>1153</td>
</tr>
<tr>
<td>colorbar_factory() (in module matplotlib.colorbar)</td>
<td>1147</td>
</tr>
<tr>
<td>ColorbarBase (class in matplotlib.colorbar)</td>
<td>1146</td>
</tr>
<tr>
<td>ColorbarPatch (class in matplotlib.colorbar)</td>
<td>1147</td>
</tr>
<tr>
<td>Colormap (class in matplotlib.colors)</td>
<td>1152</td>
</tr>
<tr>
<td>colormaps() (in module matplotlib.pyplot)</td>
<td>428</td>
</tr>
<tr>
<td>colors() (in module matplotlib.pyplot)</td>
<td>1438</td>
</tr>
<tr>
<td>complex_spectrum() (in module matplotlib.mlab)</td>
<td>1289</td>
</tr>
<tr>
<td>composite_images() (in module matplotlib.image)</td>
<td>1233</td>
</tr>
<tr>
<td>composite_transform_factory() (in module matplotlib.transforms)</td>
<td>1668</td>
</tr>
<tr>
<td>CompositeAffine2D (class in matplotlib.transforms)</td>
<td>1668</td>
</tr>
<tr>
<td>CompositeGenericTransform (class in matplotlib.transforms)</td>
<td>1667</td>
</tr>
<tr>
<td>ComputerModernFontConstants (class in matplotlib.mathtext)</td>
<td>1262</td>
</tr>
<tr>
<td>config_axis() (matplotlib.colorbar.ColorbarBase method)</td>
<td>1146</td>
</tr>
<tr>
<td>ConfigureSubplotsBase (class in matplotlib.backend_tools)</td>
<td>937</td>
</tr>
<tr>
<td>connect() (in module matplotlib.pyplot)</td>
<td>1436</td>
</tr>
<tr>
<td>connect() (matplotlib.cbook.CallbackRegistry method)</td>
<td>954</td>
</tr>
<tr>
<td>connect() (matplotlib.patches.ConnectionStyle.Angle method)</td>
<td>1340</td>
</tr>
<tr>
<td>connect() (matplotlib.patches.ConnectionStyle.Angle3 method)</td>
<td>1341</td>
</tr>
<tr>
<td>connect() (matplotlib.patches.ConnectionStyle.Arc method)</td>
<td>1341</td>
</tr>
<tr>
<td>connect() (matplotlib.patches.ConnectionStyle.Arc3 method)</td>
<td>1341</td>
</tr>
<tr>
<td>connect() (matplotlib.patches.ConnectionStyle.Bar method)</td>
<td>1342</td>
</tr>
<tr>
<td>connect() (matplotlib.widgets.MultiCursor method)</td>
<td>1694</td>
</tr>
<tr>
<td>connect_bbox() (mpl_toolkits.axes_grid1.inset_locator.BboxConnector static method)</td>
<td>1813</td>
</tr>
<tr>
<td>connect_event() (matplotlib.widgets.AxesWidget method)</td>
<td>1687</td>
</tr>
<tr>
<td>ConnectionPatch (class in matplotlib.patches)</td>
<td>1338</td>
</tr>
<tr>
<td>ConnectionStyle (class in matplotlib.patches)</td>
<td>1340</td>
</tr>
<tr>
<td>ConnectionStyle.Angle (class in matplotlib.patches)</td>
<td>1340</td>
</tr>
<tr>
<td>ConnectionStyle.Angle3 (class in matplotlib.patches)</td>
<td>1341</td>
</tr>
<tr>
<td>ConnectionStyle.Arc (class in matplotlib.patches)</td>
<td>1341</td>
</tr>
<tr>
<td>ConnectionStyle.Arc3 (class in matplotlib.patches)</td>
<td>1341</td>
</tr>
<tr>
<td>ConnectionStyle.Bar (class in matplotlib.patches)</td>
<td>1341</td>
</tr>
</tbody>
</table>
Index
Matplotlib, Release 2.0.2

convert_mesh_to_triangles() (matplotlib.collections.TriMesh static method), 1134
convert_units() (matplotlib.collections.QuadMesh method), 1098
convert_units() (matplotlib.axis.Axis method), 802
convert_units() (matplotlib.axis.XAxis method), 825
convert_units() (matplotlib.axis.YAxis method), 815
convert_units() (matplotlib.artist.Artist method), 549
convert_units() (matplotlib.axes.Axes method), 764
convert_units() (matplotlib.axis.Tick method), 839
convert_units() (matplotlib.axis.XAxis method), 890
convert_units() (matplotlib.axis.XTick method), 850
convert_units() (matplotlib.axis.YAxis method), 901
convert_units() (matplotlib.axis.YTick method), 861
convert_units() (matplotlib.collections.AsteriskPolygonCollection method), 977
convert_units() (matplotlib.collections.BrokenBarHCollection method), 989
convert_units() (matplotlib.collections.CircleCollection method), 1001
convert_units() (matplotlib.collections.Collection method), 1013
convert_units() (matplotlib.collections.EllipseCollection method), 1025
convert_units() (matplotlib.collections.EventCollection method), 1037
convert_units() (matplotlib.collections.LineCollection method), 1050
convert_units() (matplotlib.collections.PatchCollection method), 1062
convert_units() (matplotlib.collections.PolyCollection method), 1085
convert_units() (matplotlib.collections.QuadMesh method), 1098
convert_units() (matplotlib.collections.RegularPolyCollection method), 1110
convert_units() (matplotlib.collections.StarPolygonCollection method), 1122
convert_units() (matplotlib.collections.TriMesh method), 1134
convert_units() (matplotlib.artist.Artist method), 549
convert_units() (matplotlib.axes.Axes method), 764
convert_units() (matplotlib.axis.Tick method), 839
convert_units() (matplotlib.axis.XAxis method), 890
convert_units() (matplotlib.axis.XTick method), 850
convert_units() (matplotlib.axis.YAxis method), 901
convert_units() (matplotlib.axis.YTick method), 862
convert_units() (matplotlib.collections.AsteriskPolygonCollection method), 977
convert_units() (matplotlib.collections.BrokenBarHCollection method), 989
convert_units() (matplotlib.collections.CircleCollection method), 1001
convert_units() (matplotlib.collections.Collection method), 1013
convert_units() (matplotlib.collections.EllipseCollection method), 1025
convert_units() (matplotlib.collections.EventCollection method), 1037
convert_units() (matplotlib.collections.LineCollection method), 1050
convert_units() (matplotlib.collections.PatchCollection method), 1062
convert_yunits() (matplotlib.collections.PathCollection method), 1074
convert_yunits() (matplotlib.collections.PolyCollection method), 1086
convert_yunits() (matplotlib.collections.QuadMesh method), 1098
convert_yunits() (matplotlib.collections.RegularPolyCollection method), 1110
convert_yunits() (matplotlib.collections.StarPolygonCollection method), 1122
convert_yunits() (matplotlib.collections.TriMesh method), 1134
covariance_factor() (matplotlib.mlab.GaussianKDE method), 1283
cross_from_above() (in module matplotlib.mlab), 1290
cross_from_below() (in module matplotlib.mlab), 1290
csv2rec() (in module matplotlib.mlab), 1292
csvformat_factory() (in module matplotlib.mlab), 1293
cubicTriInterpolator (class in matplotlib.tri), 1674
current_key_axes() (matplotlib.figure.AxesStack method), 3035
Cursor (class in matplotlib.widgets), 1689
cursor (matplotlib.backend_tools.ToolPan attribute), 941
cursor (matplotlib.backend_tools.ToolToggleBase attribute), 942
cursor (matplotlib.backend_tools.ToolZoom attribute), 944

Cursors (class in matplotlib.backend_tools), 937
CURVE3 (matplotlib.path.Path attribute), 1368
CURVE4 (matplotlib.path.Path attribute), 1368
customspace() (matplotlib.mathtext.Parser method), 1272
cycler() (in module matplotlib.mathtext), 1593

D
data2num() (in module matplotlib.dates), 1169
DateFormatter (class in matplotlib.dates), 1169
DateLocator (class in matplotlib.dates), 1171
dateutil, 3017
DayLocator (class in matplotlib.dates), 1174
dblclick (matplotlib.backend_bases.MouseEvent attribute), 924
dedent() (in module matplotlib.cbook), 959
deepecopy() (matplotlib.path.Path method), 1370
default_keymap (matplotlib.backend_tools.SaveFigureBase attribute), 938
default_keymap (matplotlib.backend_tools.ToolBack attribute), 938
default_keymap (matplotlib.backend_tools.ToolBase attribute), 939
default_keymap (matplotlib.backend_tools.ToolEnableNavigation attribute), 940
default_keymap (matplotlib.backend_tools.ToolForward attribute), 940
default_keymap (matplotlib.backend_tools.ToolFullScreen attribute), 941
default_keymap (matplotlib.backend_tools.ToolGrid attribute), 941
default_keymap (matplotlib.backend_tools.ToolHome attribute), 941
default_keymap (matplotlib.backend_tools.ToolPan attribute), 941
default_keymap (matplotlib.backend_tools.ToolQuit attribute), 942
default_keymap (matplotlib.backend_tools.ToolXScale attribute), 943
default_keymap (matplotlib.backend_tools.ToolYScale attribute), 944
default_keymap (matplotlib.backend_tools.ToolZoom attribute), 944
default_params (matplotlib.ticker.MaxNLocator attribute), 1645
default_toolbar_tools (in module matplotlib.backend_tools), 946
default_tools (in module matplotlib.backend_tools), 946
default_units() (matplotlib.units.ConversionInterface static method), 1686

DejaVuFonts (class in matplotlib.mathtext), 1263
DejaVuSansFontConstants (class in matplotlib.mathtext), 1263
DejaVuSansFonts (class in matplotlib.mathtext), 1263
DejaVuSerifFontConstants (class in matplotlib.mathtext), 1263
DejaVuSerifFonts (class in matplotlib.mathtext), 1263
deletes() (in module matplotlib.pyplot), 1466
deletes() (matplotlib.figure.Figure method), 1193
delete_masked_points() (in module matplotlib.cbook), 959
delta (matplotlib.mathtext.ComputerModernFontConstants attribute), 1262
delta (matplotlib.mathtext.FontConstantsBase attribute), 1263
delta (matplotlib.mathtext.STIXFontConstants attribute), 1274
delta_integral (matplotlib.mathtext.ComputerModernFontConstants attribute), 1274
Index
disconnect() (matplotlib.offsetbox.DraggableBase method), 1317
disconnect() (matplotlib.widgets.Button method), 1688
disconnect() (matplotlib.widgets.CheckButtons method), 1694
disconnect() (matplotlib.widgets.MultiCursor method), 1694
disconnect() (matplotlib.widgets.RadioButtons method), 1695
disconnect() (matplotlib.widgets.Slider method), 1698
disconnect_events() (matplotlib.widgets.AxesWidget method), 1687
dist() (in module matplotlib.mlab), 1295
dist_point_to_segment() (in module matplotlib.mlab), 1295
distances_along_curve() (in module matplotlib.mlab), 1295
Divider (class in mpl_toolkits.axes_grid.axes_divider), 1796
do_3d_projection() (mpl_toolkits.mplot3d.art3d.Line3DCollection method), 1871
do_3d_projection() (mpl_toolkits.mplot3d.art3d.Patch3DCollection method), 1872
do_3d_projection() (mpl_toolkits.mplot3d.art3d.Patch3DCollection method), 1872
do_3d_projection() (mpl_toolkits.mplot3d.art3d.Patch3DCollection method), 1873
do_3d_projection() (mpl_toolkits.mplot3d.art3d.PathPatch3D method), 1873
do_3d_projection() (mpl_toolkits.mplot3d.art3d.PathPatch3D method), 1873
do_3d_projection() (mpl_toolkits.mplot3d.art3d.Poly3DCollection method), 1873
donething_callback() (in module matplotlib.mlab), 1295
dpi (matplotlib.figure.Figure attribute), 1193
drag_pan() (matplotlib.axes.Axes method), 772
drag_pan() (matplotlib.backend_bases.NavigationToolbar2 method), 925
drag_zoom() (matplotlib.backend_bases.NavigationToolbar2 method), 925
draggable() (matplotlib.legend.Legend method), 1239
DraggableAnnotation (class in matplotlib.offsetbox), 1316
DraggableBase (class in matplotlib.offsetbox), 1316
DraggableLegend (class in matplotlib.legend), 1237
DraggableOffsetBox (class in matplotlib.offsetbox), 1317
drange() (in module matplotlib.dates), 1169
draw() (in module matplotlib.pyplot), 1466
draw() (matplotlib.artist.Artist method), 543
draw() (matplotlib.axes.Axes method), 777
draw() (matplotlib.axis.Axes method), 879
draw() (matplotlib.axis.Tick method), 839
draw() (matplotlib.axis.XAxis method), 890
draw() (matplotlib.axis.XTick method), 850
draw() (matplotlib.axis.YAxis method), 901
draw() (matplotlib.axis.YTick method), 862
draw() (matplotlib.backend_bases.NavigationToolbar2 method), 914
draw() (matplotlib.backend_bases.NavigationToolbar2 method), 925
draw() (matplotlib.backends.backend_qt5agg.FigureCanvasQTAggBase method), 947
draw() (matplotlib.backends.backend_wxagg.FigureCanvasWxAgg method), 948
draw() (matplotlib.collections.AsteriskPolygonCollection method), 977
draw() (matplotlib.collections.BrokenBarHCollection method), 989
draw() (matplotlib.collections.CircleCollection method), 1001
draw() (matplotlib.collections.Collection method), 1013
draw() (matplotlib.collections.EllipseCollection method), 1025
draw() (matplotlib.collections.EventCollection method), 1037
draw() (matplotlib.collections.LineCollection method), 1050
draw() (matplotlib.collections.Matcher method), 1062
draw() (matplotlib.collections.PatchCollection method), 1074
draw() (matplotlib.collections.PolyCollection method), 1086
draw() (matplotlib.collections.QuadMesh method), 1098
draw() (matplotlib.collections.RegularPolyCollection method), 1110
draw() (matplotlib.collections.StarPolygonCollection method), 1122
Matplotlib, Release 2.0.2

draw() (matplotlib.collections.TriMesh method), 1134
draw() (matplotlib.figure.Figure method), 1193
draw() (matplotlib.legend.Legend method), 1240
draw() (matplotlib.lines.Line2D method), 1246
draw() (matplotlib.offsetbox.AnchoredOffsetbox method), 1314
draw() (matplotlib.offsetbox.AnnotationBbox method), 1315
draw() (matplotlib.offsetbox.AuxTransformBox method), 1316
draw() (matplotlib.offsetbox.DrawingArea method), 1318
draw() (matplotlib.offsetbox.OffsetBox method), 1319
draw() (matplotlib.offsetbox.OffsetImage method), 1320
draw() (matplotlib.offsetbox.PaddedBox method), 1321
draw() (matplotlib.offsetbox.TextArea method), 1322
draw() (matplotlib.patches.Arc method), 1326
draw() (matplotlib.patches.ConnectionPatch method), 1339
draw() (matplotlib.patches.FancyArrowPatch method), 1346
draw() (matplotlib.patches.Patch method), 1352
draw() (matplotlib.patches.Shadow method), 1362
draw() (matplotlib.spines.Spine method), 1616
draw() (matplotlib.text.Annotation method), 1624
draw() (matplotlib.text.Text method), 1626
draw() (matplotlib.text.TextWithDash method), 1632
draw() (mpl_toolkits.axes_grid.axis_artist.AxisArtist method), 1802
draw() (mpl_toolkits.mplot3d.axes_grid1.inset_locator.AnchoredInsetPosition method), 1811
draw() (mpl_toolkits.mplot3d.art3d.Line3D method), 1871
draw() (mpl_toolkits.mplot3d.art3d.Line3DCollection method), 1871
draw() (mpl_toolkits.mplot3d.art3d.Patch3D method), 1872
draw() (mpl_toolkits.mplot3d.art3d.Poly3DCollection method), 1873
draw() (mpl_toolkits.mplot3d.art3d.Text3D method), 1874
draw() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 1851
draw() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 1851
draw() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 1851
draw_all() (matplotlib.colorbar.ColorbarBase method), 1146
draw_artist() (matplotlib.axes.Axes method), 777
draw_artist() (matplotlib.figure.Figure method), 1193
draw_bbox() (in module matplotlib.patches), 1365
draw_cursor() (matplotlib.backend_bases.FigureCanvasBase method), 914
draw_event() (matplotlib.backend_bases.FigureCanvasBase method), 914
draw_frame() (matplotlib.legend.Legend method), 1240
draw_frame() (matplotlib.offsetbox.PaddedBox method), 1321
draw_gouraud_triangle() (matplotlib.backend_bases.RendererBase method), 928
draw_gouraud_triangles() (matplotlib.backend_bases.RendererBase method), 928
draw_idle() (matplotlib.backend_bases.FigureCanvasBase method), 914
draw_idle() (matplotlib.backends.backend_qt5agg.FigureCanvasQTAgg method), 947
draw_image() (matplotlib.backend_bases.RendererBase method), 928
draw_markers() (matplotlib.backend_bases.RendererBase method), 928
draw_marked() (matplotlib.path_effects.PathEffectRenderer method), 1376
draw_pan() (mpl_toolkits.mplot3d.art3d.Axes3D method), 1870
draw_path() (matplotlib.backend_bases.RendererBase method), 928
draw_path() (matplotlib.path_effects.AbstractPathEffect method), 1375
draw_path() (matplotlib.path_effects.PathEffectRenderer method), 1376
draw_path() (matplotlib.path_effects.PathPatchEffect method), 1376
draw_path() (matplotlib.path_effects.SimpleLineShadow method), 1376
method), 1377
draw_path() (matplotlib.path_effects.SimplePatchShadow method), 1377
draw_path() (matplotlib.path_effects.Stroke method), 1377
draw_path() (matplotlib.path_effects.withSimplePatchShadow method), 1378
draw_path() (matplotlib.path_effects.withStroke method), 1378
draw_path_collection() (matplotlib.path_effects.PathEffectRenderer method), 1376
draw_quad_mesh() (matplotlib.backend_bases.RendererBase method), 929
draw_rectangle() (matplotlib.backends.backend_qt5agg.FigureCanvasQTAggBase method), 947
drawShapes (matplotlib.lines.Line2D attribute), 1247
draw_stylekeys (matplotlib.lines.Line2D attribute), 1247
drawStyles (matplotlib.lines.Line2D attribute), 1247
drawText() (matplotlib.dviread.DviFont attribute), 1181
drawText() (matplotlib.dviread.DviFont attribute), 1181
drawOn() (matplotlib.widgets.Widget attribute), 1701
drawRectangle() (matplotlib.backends.backend_qt5agg.FigureCanvasQTAggBase method), 947
drawStyleKeys (matplotlib.widgets.RectangleSelector method), 1690
drawPath() (in module matplotlib.patches), 1342
drawPathCollection() (in module matplotlib.collections), 1023
Ellipses (in module matplotlib.widgets), 1690
embedTTF() (matplotlib.backends.backend_pdf.PdfFile method), 949
empty() (matplotlib.cbook.Stack method), 957
enable() (matplotlib.backend_tools.AxisScaleBase method), 937
enable() (matplotlib.backend_tools.ToolFullScreen method), 941
enable() (matplotlib.backend_tools.ToolGrid method), 941
enable() (matplotlib.backend_tools.ToolToggleBase method), 942
enable() (matplotlib.backend_tools.ZoomPanBase method), 944
Encoding (in module matplotlib.dviread), 1182
coding (matplotlib.dviread.Encoding attribute), 1182
end() (matplotlib.backends.backend_pdf.Stream method), 951
end_group() (matplotlib.mathtext.Parser method), 1272
end_pan() (matplotlib.axes.Axes method), 772
enter_notify_event() (matplotlib.backend_bases.FigureCanvasBase method), 914
entropy() (in module matplotlib.mlab), 1295
environment variable
HOME, 1742, 1745
MPLBACKEND, 325, 1719, 1720, 1745, 1909
MPLCONFIGDIR, 1742, 1745
PATH, 135, 139, 140, 143, 1745
PYTHONPATH, 1745, 1909
epoch2num() (in module matplotlib.dates), 1169
EPS, 3017
Error() (in module matplotlib.mathtext), 1263
errorbar() (in module matplotlib.pyplot), 1467
errorbar() (matplotlib.axes.Axes method), 560
ErrorbarContainer (in module matplotlib.container), 1165
evaluate() (matplotlib.mlab.GaussianKDE method), 1283
Event (class in matplotlib.backend_bases), 913
EventCollection (class in matplotlib.collections), 1035
eventplot() (in module matplotlib.pyplot), 1470
eventplot() (matplotlib.axes.Axes method), 590
events (matplotlib.backend_bases.FigureCanvasBase attribute), 915
eventson (matplotlib.widgets.Widget attribute), 1701
exception_to_str() (in module matplotlib.cbook), 961
exp_safe() (in module matplotlib.mlab), 1295
expanded() (matplotlib.transforms.BboxBase method), 1652
extend_positions() (matplotlib.collections.EventCollection method), 1037
extents (matplotlib.transforms.BboxBase attribute), 1652
extents (matplotlib.widgets.RectangleSelector attribute), 1697
factory() (matplotlib.mathtext.GlueSpec class method), 1266
family_escape() (in module matplotlib.fontconfig_pattern), 1225
family_name (matplotlib.afm.AFM attribute), 487
family_unescape() (in module matplotlib.fontconfig_pattern), 1225
FancyArrow (class in matplotlib.patches), 1343
FancyArrowPatch (class in matplotlib.patches), 1344
FancyBboxPatch (class in matplotlib.patches), 1347
fetch_historical_yahoo() (in module matplotlib.finance), 1210
FFMpegBase (class in matplotlib.animation), 530
FFMpegFileWriter (class in matplotlib.animation), 519
FFMpegWriter (class in matplotlib.animation), 514
fitsurr() (in module matplotlib.mlab), 1296
figaspect() (in module matplotlib.figure), 1205
figimage() (in module matplotlib.pyplot), 1425
figimage() (matplotlib.figure.Figure method), 1193
figlegend() (in module matplotlib.pyplot), 1474
fignum_exists() (in module matplotlib.pyplot), 1474
figtext() (in module matplotlib.pyplot), 1474
Figure (class in matplotlib.figure), 1186
Figure (matplotlib.backend_tools.ToolBase attribute), 939
figure() (in module matplotlib.pyplot), 1476
FigureCanvas (in module matplotlib.backends.backend_pdf), 948
FigureCanvas (in module matplotlib.backends.backend_qt4agg), 946
FigureCanvas (in module matplotlib.backends.backend_qt5agg), 946
FigureCanvas (in module matplotlib.backends.backend_wxagg), 947
FigureCanvasBase (class in matplotlib.backend_bases), 914
FigureCanvasPdf (class in matplotlib.backends.backend_pdf), 948
FigureCanvasQTAgg (class in matplotlib.backends.backend_qt4agg), 946
FigureCanvasQTAgg (class in matplotlib.backends.backend_qt5agg), 946
FigureCanvasQTAggBase (class in matplotlib.backends.backend_qt4agg), 946
FigureCanvasQTAggBase (class in matplotlib.backends.backend_qt5agg), 947
FigureCanvasWxAgg (class in matplotlib.backends.backend_wxagg), 947
FigureFrameWxAgg (class in matplotlib.backends.backend_wxagg), 948
FigureImage (class in matplotlib.image), 1232
FigureManagerBase (class in matplotlib.backend_bases), 919
Fil (class in matplotlib.mathtext), 1263
file_requires_unicode() (in module matplotlib.cbook), 961
FileMovieWriter (class in matplotlib.animation), 529
filetypes (matplotlib.backend_bases.FigureCanvasBase attribute), 915
filetypes (matplotlib.backends.backend_wxagg.FigureCanvasWxAgg attribute), 948
Fill (class in matplotlib.mathtext), 1263
fill (matplotlib.patches.Patch attribute), 1352
fill() (in module matplotlib.backends.backend_pdf), 951
fill() (in module matplotlib.pyplot), 1476
fill() (matplotlib.axes.Axes method), 599
fill_between() (in module matplotlib.pyplot), 1478
fill_between() (matplotlib.axes.Axes method), 575
fill_betweenx() (in module matplotlib.pyplot), 1482
Index 3043

1221 Fonts (class in matplotlib.mathtext), 1264
format_coord() (matplotlib.axes.Axes method), 772
format_coord() (matplotlib.projections.polar.PolarAxes method), 1382
format_coord() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 1851
format_cursor_data() (matplotlib.artist.Artist method), 537
format_cursor_data() (matplotlib.axes.Axes method), 772
format_cursor_data() (matplotlib.axis.Axis method), 879
format_cursor_data() (matplotlib.axis.Tick method), 839
format_cursor_data() (matplotlib.axis.XAxis method), 891
format_cursor_data() (matplotlib.axis.XTick method), 851
format_cursor_data() (matplotlib.axis.YAxis method), 902
format_cursor_data() (matplotlib.axis.YTick method), 862
format_cursor_data() (matplotlib.collections.AsteriskPolygonCollection method), 977
format_cursor_data() (matplotlib.collections.BrokenBarHCollection method), 989
format_cursor_data() (matplotlib.collections.CircleCollection method), 1001
format_cursor_data() (matplotlib.collections.Collection method), 1013
format_cursor_data() (matplotlib.collections.EllipseCollection method), 1025
format_cursor_data() (matplotlib.collections.EventCollection method), 1038
format_cursor_data() (matplotlib.collections.LineCollection method), 1050
format_cursor_data() (matplotlib.collections.PatchCollection method), 1062
format_cursor_data() (matplotlib.collections.PathCollection method), 1074
format_cursor_data() (matplotlib.collections.PolyCollection method), 1086
format_cursor_data() (matplotlib.collections.QuadMesh method), 1098
format_cursor_data() (matplotlib.collections.RegularPolyCollection method), 1110
format_cursor_data() (matplotlib.collections.StarPolygonCollection method), 1122
format_cursor_data() (matplotlib.collections.TriMesh method), 1134
format_data_short() (matplotlib.axes.Axes method), 772
format_data_short() (matplotlib.ticker.Formatter method), 1638
format_data_short() (matplotlib.ticker.LogFormatter method), 1641
format_data_short() (matplotlib.ticker.ScalarFormatter method), 1639
format_data_short() (matplotlib.ticker.ScalarFormatter method), 1639
format_xdata() (matplotlib.axes.Axes method), 772
format_ydata() (matplotlib.axes.Axes method), 773
format_zdata() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 1851
FormatBool (class in matplotlib.mlab), 1281
FormatDate (class in matplotlib.mlab), 1281
FormatDatetime (class in matplotlib.mlab), 1281
FormatFloat (class in matplotlib.mlab), 1281
FormatFormatStr (class in matplotlib.mlab), 1281
FormatInt (class in matplotlib.mlab), 1281
FormatMillions (class in matplotlib.mlab), 1282
FormatObj (class in matplotlib.mlab), 1282
FormatPercent (class in matplotlib.mlab), 1282
FormatPercentFormatter (class in matplotlib.ticker), 1638
FormatString (class in matplotlib.mlab), 1282
Formatter (class in matplotlib.ticker), 1637
FormatThousands (class in matplotlib.mlab), 1282
null
get_adjustable() (matplotlib.axes.Axes method), 755
get_affine() (matplotlib.transforms.AffineBase method), 1660
get_affine() (matplotlib.transforms.BlendedGenericTransform method), 1666
get_affine() (matplotlib.transforms.CompositeGenericTransform method), 1667
get_affine() (matplotlib.transforms.IdentityTransform method), 1664
get_affine() (matplotlib.transforms.Transform method), 1658
get_agg_filter() (matplotlib.artist.Artist method), 543
get_agg_filter() (matplotlib.axes.Axes method), 782
get_agg_filter() (matplotlib.axis.Axis method), 879
get_agg_filter() (matplotlib.axis.Tick method), 840
get_agg_filter() (matplotlib.axis.XAxis method), 891
get_agg_filter() (matplotlib.axis.XTick method), 851
get_agg_filter() (matplotlib.axis.YAxis method), 902
get_agg_filter() (matplotlib.axis.YTick method), 862
get_agg_filter() (matplotlib.collections.AsteriskPolygonCollection method), 977
get_agg_filter() (matplotlib.collections.BrokenBarHCollection method), 989
get_agg_filter() (matplotlib.collections.CircleCollection method), 1001
get_agg_filter() (matplotlib.collections.Collection method), 1013
get_agg_filter() (matplotlib.collections.EllipseCollection method), 1025
get_agg_filter() (matplotlib.collections.EventCollection method), 1038
get_agg_filter() (matplotlib.collections.LineCollection method), 1051
get_agg_filter() (matplotlib.collections.PatchCollection method), 1062
get_agg_filter() (matplotlib.collections.PathCollection method), 1074
get_agg_filter() (matplotlib.collections.PolyCollection method), 1086
get_agg_filter() (matplotlib.collections.QuadMesh method), 1098
get_agg_filter() (matplotlib.collections.RegularPolyCollection method), 1110
get_agg_filter() (matplotlib.collections.StarPolygonCollection method), 1122
get_agg_filter() (matplotlib.collections.TriMesh method), 1134
get_alpha() (matplotlib.artist.Artist method), 543
get_alpha() (matplotlib.axes.Axes method), 783
get_alpha() (matplotlib.axis.Axis method), 880
get_alpha() (matplotlib.axis.Tick method), 840
get_alpha() (matplotlib.axis.XAxis method), 891
get_alpha() (matplotlib.axis.XTick method), 851
get_alpha() (matplotlib.axis.YAxis method), 902
get_alpha() (matplotlib.axis.YTick method), 862
get_alpha() (matplotlib.backend_bases.GraphicsContextBase method), 920
get_alpha() (matplotlib.collections.AsteriskPolygonCollection method), 977
get_alpha() (matplotlib.collections.BrokenBarHCollection method), 989
get_alpha() (matplotlib.collections.CircleCollection method), 1001
get_alpha() (matplotlib.collections.Collection method), 1013
get_alpha() (matplotlib.collections.EllipseCollection method), 1025
get_alpha() (matplotlib.collections.EventCollection method), 1038
get_alpha() (matplotlib.collections.LineCollection method), 1051
get_alpha() (matplotlib.collections.PatchCollection method), 1062
get_alpha() (matplotlib.collections.PathCollection method), 1074
get_alpha() (matplotlib.collections.PolyCollection method), 1086
get_alpha() (matplotlib.collections.QuadMesh method), 1098
get_alpha() (matplotlib.collections.RegularPolyCollection method), 1110
get_alpha() (matplotlib.collections.StarPolygonCollection method), 1123
method), 929
get_capheight() (matplotlib.afm.AFM method), 488
get_capstyle() (matplotlib.backend_bases.GraphicsContextBase method), 920
get_capstyle() (matplotlib.markers.MarkerStyle method), 1257
get_capstyle() (matplotlib.patches.Patch method), 1352
get_child() (matplotlib.offsetbox.AnchoredOffsetbox method), 1314
get_children() (matplotlib.artist.Artist method), 548
get_children() (matplotlib.axes.Axes method), 776
get_children() (matplotlib.axis.Axis method), 880
get_children() (matplotlib.axis.XAxis method), 891
get_children() (matplotlib.axis.YTick method), 852
get_children() (matplotlib.axis.YAxis method), 902
get_children() (matplotlib.axis.YTick method), 863
get_children() (matplotlib.collections.AsteriskPolygonCollection method), 977
get_children() (matplotlib.collections.BrokenBarHCollection method), 989
get_children() (matplotlib.collections.CircleCollection method), 1001
get_children() (matplotlib.collections.Collection method), 1014
get_children() (matplotlib.collections.EllipseCollection method), 1025
get_children() (matplotlib.collections.EventCollection method), 1038
get_children() (matplotlib.collections.LineCollection method), 1051
get_children() (matplotlib.collections.PatchCollection method), 1063
get_children() (matplotlib.collections.PathCollection method), 1074
get_children() (matplotlib.collections.PolyCollection method), 1086
get_children() (matplotlib.collections.QuadMesh method), 1098
get_children() (matplotlib.collections.RegularPolyCollection method), 1110
get_children() (matplotlib.collections.StarPolygonCollection method), 1123
get_children() (matplotlib.collections.TriMesh method), 1135
get_children() (matplotlib.container.Container method), 1165
get_children() (matplotlib.figure.Figure method), 1196
get_children() (matplotlib.legend.Legend method), 1240
get_children() (matplotlib.offsetbox.AnchoredOffsetbox method), 1314
get_children() (matplotlib.offsetbox.AnnotatedBbox method), 1315
get_clim() (matplotlib.cm.ScalarMappable method), 972
get_clim() (matplotlib.collections.AsteriskPolygonCollection method), 977
get_clim() (matplotlib.collections.BrokenBarHCollection method), 989
get_clim() (matplotlib.collections.CircleCollection method), 1001
get_clim() (matplotlib.collections.Collection method), 1014
get_clim() (matplotlib.collections.EllipseCollection method), 1025
get_clim() (matplotlib.collections.EventCollection method), 1038
get_clim() (matplotlib.collections.LineCollection method), 1051
get_clim() (matplotlib.collections.PatchCollection method), 1063
get_clim() (matplotlib.collections.PathCollection method), 1074
get_clim() (matplotlib.collections.PolyCollection method), 1086
get_clim() (matplotlib.collections.QuadMesh method), 1098
get_clim() (matplotlib.collections.RegularPolyCollection method), 1110
get_clim() (matplotlib.collections.StarPolygonCollection method), 1123
get_clim() (matplotlib.collections.TriMesh method), 1135
get_clim() (matplotlib.container.Container method), 1165
get_clim() (matplotlib.figure.Figure method), 1196
get_clim() (matplotlib.legend.Legend method), 1240
get_clim() (matplotlib.offsetbox.AnchoredOffsetbox method), 1314
get_clim() (matplotlib.offsetbox.AnnotatedBbox method), 1315
get_clim() (matplotlib.offsetbox.OffsetBox method), 1319
get_clim() (matplotlib.offsetbox.OffsetImage method), 1320
get_clim() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 1852
get_clim() (matplotlib.cm.ScalarMappable method), 972
get_clim() (matplotlib.collections.AsteriskPolygonCollection method), 977
get_clim() (matplotlib.collections.BrokenBarHCollection method), 989
get_clim() (matplotlib.collections.CircleCollection method), 1001
get_clim() (matplotlib.collections.Collection method), 1014
get_clim() (matplotlib.collections.EllipseCollection method), 1025
get_clim() (matplotlib.collections.EventCollection method), 1038
get_clim() (matplotlib.collections.LineCollection method), 1051
get_clim() (matplotlib.collections.PatchCollection method), 1063
get_clim() (matplotlib.collections.PathCollection method), 1074
get_clim() (matplotlib.collections.PolyCollection method), 1086
get_clim() (matplotlib.collections.QuadMesh method), 1098
get_clim() (matplotlib.collections.RegularPolyCollection method), 1110
get_clim() (matplotlib.collections.StarPolygonCollection method), 1123
get_clim() (matplotlib.collections.TriMesh method), 1135
get_clip_box() (matplotlib.artist.Artist method), 540
get_clip_box() (matplotlib.axes.Axes method), 783
get_clip_box() (matplotlib.axis.Axis method), 880
get_clip_box() (matplotlib.axis.Tick method), 840
get_clip_box() (matplotlib.axis.XAxis method), 891
get_clip_box() (matplotlib.axis.XTick method), 852
get_clip_box() (matplotlib.axis.YAxis method), 902
get_clip_box() (matplotlib.axis.YTick method), 863
get_clip_box() (matplotlib.collections.AsteriskPolygonCollection method), 978
get_clip_box() (matplotlib.collections.BrokenBarHCollection method), 990
get_clip_box() (matplotlib.collections.CircleCollection method), 1002
get_clip_box() (matplotlib.collections.Collection method), 1014
get_clip_box() (matplotlib.collections.EllipseCollection method), 1026
get_clip_box() (matplotlib.collections.EventCollection method), 1038
get_clip_box() (matplotlib.collections.LineCollection method), 1051
get_clip_box() (matplotlib.collections.PatchCollection method), 1063
getClipBox() (matplotlib.collections.PatchCollection method), 1074
get_clip_box() (matplotlib.collections.PolyCollection method), 1086
get_clip_box() (matplotlib.collections.QuadMesh method), 1099
get_clip_box() (matplotlib.collections.RegularPolyCollection method), 1111
get_clip_box() (matplotlib.collections.StarPolygonCollection method), 1123
get_clip_on() (matplotlib.artist.Artist method), 540
get_clip_on() (matplotlib.axes.Axes method), 783
get_clip_on() (matplotlib.axis.Axis method), 880
get_clip_on() (matplotlib.axis.Tick method), 840
get_clip_on() (matplotlib.axis.XAxis method), 891
get_clip_on() (matplotlib.axis.XTick method), 852
get_clip_on() (matplotlib.axis.YAxis method), 903
get_clip_on() (matplotlib.axis.YTick method), 863
get_clip_on() (matplotlib.collections.AsteriskPolygonCollection method), 978
get_clip_on() (matplotlib.collections.BrokenBarHCollection method), 990
get_clip_on() (matplotlib.collections.CircleCollection method), 1002
get_clip_on() (matplotlib.collections.Collection method), 1014
get_clip_on() (matplotlib.collections.EllipseCollection method), 1026
get_clip_on() (matplotlib.collections.EventCollection method), 1038
get_clip_on() (matplotlib.collections.LineCollection method), 1051
get_clip_on() (matplotlib.collections.PatchCollection method), 1063
get_clip_on() (matplotlib.collections.PolyCollection method), 1086
get_clip_on() (matplotlib.collections.QuadMesh method), 1099
get_clip_on() (matplotlib.collections.RegularPolyCollection method), 1111
get_clip_on() (matplotlib.collections.StarPolygonCollection method), 1123
get_clip_on() (matplotlib.collections.TriMesh method), 1135
get_clip_path() (matplotlib.artist.Artist method), 540
get_clip_path() (matplotlib.axes.Axes method), 783
get_clip_path() (matplotlib.axis.Axis method), 880
get_clip_path() (matplotlib.axis.Tick method), 840
get_clip_path() (matplotlib.axis.XAxis method), 892
get_clip_path() (matplotlib.axis.XTick method), 852
get_clip_path() (matplotlib.axis.YAxis method), 903
get_clip_path() (matplotlib.axis.YTick method), 863
get_clip_path() (matplotlib.backend_bases.GraphicsContextBase method), 920
get_clip_path() (matplotlib.collections.AsteriskPolygonCollection method), 978
get_clip_path() (matplotlib.collections.BrokenBarHCollection method), 990
get_clip_path() (matplotlib.collections.CircleCollection method), 1002
get_clip_path() (matplotlib.collections.Collection method), 1014
get_clip_path() (matplotlib.collections.EllipseCollection method), 1026
get_clip_path() (matplotlib.collections.EventCollection method), 1038
get_clip_path() (matplotlib.collections.LineCollection method), 1051
get_clip_path() (matplotlib.collections.PatchCollection method), 1063
get_clip_path() (matplotlib.collections.PathCollection method), 1074
get_clip_path() (matplotlib.collections.PolyCollection method), 1086
get_clip_path() (matplotlib.collections.QuadMesh method), 1099
get_clip_path() (matplotlib.collections.RegularPolyCollection method), 1111
get_clip_path() (matplotlib.collections.StarPolygonCollection method), 1123
get_clip_path() (matplotlib.collections.TriMesh method), 1135
get_clip_rectangle() (matplotlib.backend_bases.GraphicsContextBase method), 920
get_closed() (matplotlib.patches.Polygon method), 1357
get_cmap() (in module matplotlib.cm), 972
get_cmap() (matplotlib.cm.ScalarMappable method), 972
get_cmap() (matplotlib.collections.AsteriskPolygonCollection method), 978
get_cmap() (matplotlib.collections.BrokenBarHCollection method), 990
get_cmap() (matplotlib.collections.CircleCollection method), 1002
get_cmap() (matplotlib.collections.Collection method), 1014
get_cmap() (matplotlib.collections.EllipseCollection method), 1026
get_cmap() (matplotlib.collections.EventCollection method), 1038
get_cmap() (matplotlib.collections.LineCollection method), 1051
get_cmap() (matplotlib.collections.PatchCollection method), 1063
get_cmap() (matplotlib.collections.PathCollection method), 1074
get_cmap() (matplotlib.collections.PolyCollection method), 1086
get_cmap() (matplotlib.collections.QuadMesh method), 1099
get_cmap() (matplotlib.collections.RegularPolyCollection method), 1111
get_cmap() (matplotlib.collections.StarPolygonCollection method), 1123
get_cmap() (matplotlib.collections.TriMesh method), 1135
get_color() (matplotlib.collections.EventCollection method), 1038
get_color() (matplotlib.collections.LineCollection method), 1051
get_color() (matplotlib.collections.Line2D method), 1247

3050 Index
<table>
<thead>
<tr>
<th>Function Call</th>
<th>Module/Class</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_color() (matplotlib.text.Text method)</td>
<td></td>
<td>1626</td>
</tr>
<tr>
<td>get_colors() (mpl_toolkits.mplot3d.art3d)</td>
<td></td>
<td>1874</td>
</tr>
<tr>
<td>get_colors() (matplotlib.collections.EventCollection method)</td>
<td></td>
<td>1038</td>
</tr>
<tr>
<td>get_colors() (matplotlib.collections.LineCollection method)</td>
<td></td>
<td>1051</td>
</tr>
<tr>
<td>get_connectionstyle() (matplotlib.patches.FancyArrowPatch method)</td>
<td></td>
<td>1346</td>
</tr>
<tr>
<td>get_contains() (matplotlib.artist.Artist method)</td>
<td></td>
<td>537</td>
</tr>
<tr>
<td>get_contains() (matplotlib.axes.Axes method)</td>
<td></td>
<td>774</td>
</tr>
<tr>
<td>get_contains() (matplotlib.axis.Axis method)</td>
<td></td>
<td>880</td>
</tr>
<tr>
<td>get_contains() (matplotlib.axis.Tick method)</td>
<td></td>
<td>841</td>
</tr>
<tr>
<td>get_contains() (matplotlib.axis.XAxis method)</td>
<td></td>
<td>892</td>
</tr>
<tr>
<td>get_contains() (matplotlib.axis.XTick method)</td>
<td></td>
<td>852</td>
</tr>
<tr>
<td>get_contains() (matplotlib.axis.YAxis method)</td>
<td></td>
<td>903</td>
</tr>
<tr>
<td>get_contains() (matplotlib.axis.YTick method)</td>
<td></td>
<td>863</td>
</tr>
<tr>
<td>get_contains() (matplotlib.collections.AsteriskPolygonCollection method)</td>
<td></td>
<td>978</td>
</tr>
<tr>
<td>get_contains() (matplotlib.collections.BrokenBarHCollection method)</td>
<td></td>
<td>990</td>
</tr>
<tr>
<td>get_contains() (matplotlib.collections.CircleCollection method)</td>
<td></td>
<td>1002</td>
</tr>
<tr>
<td>get_contains() (matplotlib.collections.Collection method)</td>
<td></td>
<td>1014</td>
</tr>
<tr>
<td>get_contains() (matplotlib.collections.EllipseCollection method)</td>
<td></td>
<td>1026</td>
</tr>
<tr>
<td>get_contains() (matplotlib.collections.EventCollection method)</td>
<td></td>
<td>1038</td>
</tr>
<tr>
<td>get_contains() (matplotlib.collections.LineCollection method)</td>
<td></td>
<td>1051</td>
</tr>
<tr>
<td>get_contains() (matplotlib.collections.PatchCollection method)</td>
<td></td>
<td>1063</td>
</tr>
<tr>
<td>get_contains() (matplotlib.collections.PathCollection method)</td>
<td></td>
<td>1075</td>
</tr>
<tr>
<td>get_contains() (matplotlib.collections.PolyCollection method)</td>
<td></td>
<td>1087</td>
</tr>
<tr>
<td>get_contains() (matplotlib.collections.QuadMesh method)</td>
<td></td>
<td>1099</td>
</tr>
<tr>
<td>get_contains() (matplotlib.collections.RegularPolyCollection method)</td>
<td></td>
<td>1111</td>
</tr>
<tr>
<td>get_contains() (matplotlib.collections.StarPolygonCollection method)</td>
<td></td>
<td>1123</td>
</tr>
<tr>
<td>get_contains() (matplotlib.collections.TriMesh method)</td>
<td></td>
<td>1135</td>
</tr>
<tr>
<td>get_converter() (matplotlib.units.Registry method)</td>
<td></td>
<td>1686</td>
</tr>
<tr>
<td>get_current_fig_manager() (matplotlib.pyplot)</td>
<td></td>
<td>1486</td>
</tr>
<tr>
<td>get_cursor_data() (matplotlib.artist.Artist method)</td>
<td></td>
<td>538</td>
</tr>
<tr>
<td>get_cursor_data() (matplotlib.axes.Axes method)</td>
<td></td>
<td>775</td>
</tr>
<tr>
<td>get_cursor_data() (matplotlib.axis.Axis method)</td>
<td></td>
<td>881</td>
</tr>
<tr>
<td>get_cursor_data() (matplotlib.axis.Tick method)</td>
<td></td>
<td>841</td>
</tr>
<tr>
<td>get_cursor_data() (matplotlib.axis.XAxis method)</td>
<td></td>
<td>892</td>
</tr>
<tr>
<td>get_cursor_data() (matplotlib.axis.XTick method)</td>
<td></td>
<td>852</td>
</tr>
<tr>
<td>get_cursor_data() (matplotlib.axis.YAxis method)</td>
<td></td>
<td>903</td>
</tr>
<tr>
<td>get_cursor_data() (matplotlib.axis.YTick method)</td>
<td></td>
<td>863</td>
</tr>
<tr>
<td>get_cursor_data() (matplotlib.collections.AsteriskPolygonCollection method)</td>
<td></td>
<td>978</td>
</tr>
<tr>
<td>get_cursor_data() (matplotlib.collections.BrokenBarHCollection method)</td>
<td></td>
<td>990</td>
</tr>
<tr>
<td>get_cursor_data() (matplotlib.collections.CircleCollection method)</td>
<td></td>
<td>1002</td>
</tr>
<tr>
<td>get_cursor_data() (matplotlib.collections.Collection method)</td>
<td></td>
<td>1014</td>
</tr>
<tr>
<td>get_cursor_data() (matplotlib.collections.EllipseCollection method)</td>
<td></td>
<td>1026</td>
</tr>
<tr>
<td>get_cursor_data() (matplotlib.collections.EventCollection method)</td>
<td></td>
<td>1038</td>
</tr>
<tr>
<td>get_cursor_data() (matplotlib.collections.LineCollection method)</td>
<td></td>
<td>1051</td>
</tr>
<tr>
<td>get_cursor_data() (matplotlib.collections.PatchCollection method)</td>
<td></td>
<td>1063</td>
</tr>
<tr>
<td>get_cursor_data() (matplotlib.collections.PathCollection method)</td>
<td></td>
<td>1075</td>
</tr>
<tr>
<td>get_cursor_data() (matplotlib.collections.PolyCollection method)</td>
<td></td>
<td>1087</td>
</tr>
<tr>
<td>get_cursor_data() (matplotlib.collections.QuadMesh method)</td>
<td></td>
<td>1099</td>
</tr>
<tr>
<td>get_cursor_data() (matplotlib.collections.RegularPolyCollection method)</td>
<td></td>
<td>1111</td>
</tr>
<tr>
<td>get_cursor_data() (matplotlib.collections.StarPolygonCollection method)</td>
<td></td>
<td>1123</td>
</tr>
<tr>
<td>get_cursor_data() (matplotlib.collections.TriMesh method)</td>
<td></td>
<td>1135</td>
</tr>
<tr>
<td>get_converter() (matplotlib.units.Registry method)</td>
<td></td>
<td>1686</td>
</tr>
<tr>
<td>get_current_fig_manager() (matplotlib.pyplot)</td>
<td></td>
<td>1486</td>
</tr>
<tr>
<td>get_cursor_data() (matplotlib.artist.Artist method)</td>
<td></td>
<td>538</td>
</tr>
<tr>
<td>get_cursor_data() (matplotlib.axes.Axes method)</td>
<td></td>
<td>775</td>
</tr>
<tr>
<td>get_cursor_data() (matplotlib.axis.Axis method)</td>
<td></td>
<td>881</td>
</tr>
<tr>
<td>get_cursor_data() (matplotlib.axis.Tick method)</td>
<td></td>
<td>841</td>
</tr>
<tr>
<td>get_cursor_data() (matplotlib.axis.XAxis method)</td>
<td></td>
<td>892</td>
</tr>
<tr>
<td>get_cursor_data() (matplotlib.axis.XTick method)</td>
<td></td>
<td>852</td>
</tr>
<tr>
<td>get_cursor_data() (matplotlib.axis.YAxis method)</td>
<td></td>
<td>903</td>
</tr>
<tr>
<td>get_cursor_data() (matplotlib.axis.YTick method)</td>
<td></td>
<td>863</td>
</tr>
<tr>
<td>get_cursor_data() (matplotlib.collections.AsteriskPolygonCollection method)</td>
<td></td>
<td>978</td>
</tr>
<tr>
<td>get_cursor_data() (matplotlib.collections.BrokenBarHCollection method)</td>
<td></td>
<td>990</td>
</tr>
<tr>
<td>get_cursor_data() (matplotlib.collections.CircleCollection method)</td>
<td></td>
<td>1002</td>
</tr>
<tr>
<td>get_cursor_data() (matplotlib.collections.Collection method)</td>
<td></td>
<td>1014</td>
</tr>
<tr>
<td>get_cursor_data() (matplotlib.collections.EllipseCollection method)</td>
<td></td>
<td>1026</td>
</tr>
<tr>
<td>get_cursor_data() (matplotlib.collections.EventCollection method)</td>
<td></td>
<td>1038</td>
</tr>
<tr>
<td>get_cursor_data() (matplotlib.collections.LineCollection method)</td>
<td></td>
<td>1051</td>
</tr>
</tbody>
</table>

Index 3051
get_cursor_data() (matplotlib.collections.PatchCollection method), 1063
get_cursor_data() (matplotlib.collections.PathCollection method), 1075
get_cursor_data() (matplotlib.collections.PolyCollection method), 1087
get_cursor_data() (matplotlib.collections.QuadMesh method), 1099
get_cursor_data() (matplotlib.collections.RegularPolyCollection method), 1111
get_cursor_data() (matplotlib.collections.StarPolygonCollection method), 1123
get_cursor_data() (matplotlib.collections.TriMesh method), 1135
get_data() (matplotlib.image.AxesImage method), 1231
get_data() (matplotlib.image.PcolorImage method), 1233
get_data_interval() (matplotlib.axis.Axis method), 801
get_data_interval() (matplotlib.axis.XAxis method), 815
get_data_interval() (matplotlib.axis.YAxis method), 825
get_data_interval() (mpl_toolkits.mplot3d.axis3d.XAxis method), 1870
get_data_interval() (mpl_toolkits.mplot3d.axis3d.YAxis method), 1870
get_data_interval() (mpl_toolkits.mplot3d.axis3d.ZAxis method), 1871
get_data_ratio() (matplotlib.axes.Axes method), 787
get_data_ratio() (matplotlib.projections.polar.PolarAxes method), 1382
get_dashes() (matplotlib.collections.EventCollection method), 1026
get_dashes() (matplotlib.collections.LineCollection method), 1051
get_dashes() (matplotlib.collections.PatchCollection method), 1063
get_dashes() (matplotlib.collections.PathCollection method), 1075
get_dashes() (matplotlib.collections.PolyCollection method), 1087
get_dashes() (matplotlib.collections.QuadMesh method), 1099
get_dashes() (matplotlib.collections.RegularPolyCollection method), 1111
get_dashes() (matplotlib.collections.StarPolygonCollection method), 1123
get_dashes() (matplotlib.collections.TriMesh method), 1135
get_dashcapstyle() (matplotlib.lines.Line2D method), 1247
get_dashjoinstyle() (matplotlib.lines.Line2D method), 1247
get_dashdirection() (matplotlib.text.TextWithDash method), 1632
get_dashlength() (matplotlib.text.TextWithDash method), 1632
get_dashpad() (matplotlib.text.TextWithDash method), 1632
get_dashpush() (matplotlib.text.TextWithDash method), 1632
get_dataratio() (matplotlib.axes.Axes method), 815
get_data_interval() (mpl_toolkits.mplot3d.axis3d.XAxis method), 1870
get_data_interval() (mpl_toolkits.mplot3d.axis3d.YAxis method), 1870
get_data_interval() (mpl_toolkits.mplot3d.axis3d.ZAxis method), 1871
get_datalowering() (matplotlib.collections.EllipseCollection method), 1026
Index 3053

get_data_ratio_log() (matplotlib.axes.Axes method), 787
get_data_transform() (matplotlib.patches.Patch method), 1352
get_datalim() (matplotlib.collections.AsteriskPolygonCollection get_default_size() (matplotlib.legend.Legend class method), 1240
get_datalim() (matplotlib.collections.BrokenBarHCollection get_default_weight() (matplotlib.font_manager.FontManager method), 1220
get_datalim() (matplotlib.collections.CircleCollection method), 990
get_datalim() (matplotlib.collections.EllipseCollection method), 1026
get_datalim() (matplotlib.collections.EventCollection method), 1038
get_datalim() (matplotlib.collections.LineCollection method), 1051
get_datalim() (matplotlib.collections.PatchCollection method), 1063
get_datalim() (matplotlib.collections.PathCollection method), 1075
get_datalim() (matplotlib.collections.PolyCollection method), 1087
get_datalim() (matplotlib.collections.QuadMesh method), 1099
get_datalim() (matplotlib.collections.RegularPolyCollection method), 1111
get_datalim() (matplotlib.collections.StarPolygonCollection method), 1123
get_datalim() (matplotlib.collections.TriMesh method), 1135
get_default_bbox_extra_artists() (matplotlib.axes.Axes method), 789
get_default_bbox_extra_artists() (matplotlib.figure.Figure method), 1196
get_default_filename() (matplotlib.backend_bases.FigureCanvasBase get_default_size() (matplotlib.font_manager.FontManager static method), 1220
get_default_filetype() (matplotlib.backend_bases.FigureCanvasBase class method), 915
get_default_handler_map() (matplotlib.legend.Legend class method), 1240
get_default_size() (matplotlib.font_manager.FontManager static method), 1220
get_depth() (matplotlib.mathtext.MathTextParser method), 1267
get_dir_vector() (in module mpl_toolkits.mplot3d.art3d), 1874
get_dpi() (matplotlib.figure.Figure method), 1197
get_dpi_cor() (matplotlib.patches.FancyArrowPatch method), 1346
get_drawstyle() (matplotlib.lines.Line2D method), 1247
get_ec() (matplotlib.patches.Patch method), 1352
get_edgecolor() (matplotlib.collections.AsteriskPolygonCollection method), 978
get_edgecolor() (matplotlib.collections.BrokenBarHCollection method), 990
get_edgecolor() (matplotlib.collections.CircleCollection method), 1002
get_edgecolor() (matplotlib.collections.EllipseCollection method), 1026
get_edgecolor() (matplotlib.collections.EventCollection method), 1039
get_edgecolor() (matplotlib.collections.LineCollection method), 1051
get_edgecolor() (matplotlib.collections.PatchCollection method), 1063
get_edgecolor() (matplotlib.collections.PolyCollection method), 1075
get_edgecolor() (matplotlib.collections.QuadMesh method), 1099
get_edgecolor() (matplotlib.collections.StarPolygonCollection method), 1123
get_edgecolor() (matplotlib.collections.TriMesh method), 1135
get_edgecolor() (matplotlib.collections.LineCollection method), 1051
get_edgecolor() (matplotlib.collections.PatchCollection method), 1063
get_edgecolor() (matplotlib.collections.PolyCollection method), 1075
get_edgecolor() (matplotlib.collections.QuadMesh method), 1099
get_edgecolor() (matplotlib.collections.StarPolygonCollection method), 1123
get_edgecolor() (matplotlib.collections.TriMesh method), 1135
get_edgecolor() (matplotlib.collections.LineCollection method), 1051
get_edgecolor() (matplotlib.collections.PatchCollection method), 1063
get_edgecolor() (matplotlib.collections.PolyCollection method), 1075
get_edgecolor() (matplotlib.collections.QuadMesh method), 1099
get_edgecolor() (matplotlib.collections.StarPolygonCollection method), 1123
get_edgecolor() (matplotlib.collections.TriMesh method), 1135
get_edgecolor() (matplotlib.collections.PolyCollection method), 1087
get_edgecolor() (matplotlib.collections.QuadMesh method), 1099
get_edgecolor() (matplotlib.collections.RegularPolyCollection method), 1111
get_edgecolor() (matplotlib.collections.StarPolygonCollection method), 1123
get_edgecolor() (matplotlib.collections.TriMesh method), 1135
get_edgecolor() (matplotlib.figure.Figure method), 1197
get_edgecolor() (matplotlib.patches.Patch method), 1352
get_edgecolor() (mpl_toolkits.mplot3d.art3d.Poly3DCollection method), 1873
get_edgecolors() (matplotlib.collections.AsteriskPolygonCollection method), 978
get_edgecolors() (matplotlib.collections.BrokenBarHCollection method), 990
get_edgecolors() (matplotlib.collections.CircleCollection method), 1002
get_edgecolors() (matplotlib.collections.Collection method), 1014
get_edgecolors() (matplotlib.collections.EllipseCollection method), 1026
get_edgecolors() (matplotlib.collections.EventCollection method), 1039
get_edgecolors() (matplotlib.collections.LineCollection method), 1052
get_edgecolors() (matplotlib.collections.PatchCollection method), 1063
get_edgecolors() (matplotlib.collections.PathCollection method), 1075
get_edgecolors() (matplotlib.collections.PolyCollection method), 1087
get_edgecolors() (matplotlib.collections.QuadMesh method), 1099
get_edgecolors() (matplotlib.collections.RegularPolyCollection method), 1111
get_edgecolors() (matplotlib.collections.StarPolygonCollection method), 1123
get_edgecolors() (matplotlib.collections.TriMesh method), 1135
get_edgecolors() (mpl_toolkits.mplot3d.art3d.Poly3DCollection method), 1873
get_err_size() (matplotlib.legend_handler.HandlerErrorbar method), 1242
g get_extent() (matplotlib.image.AxesImage method), 1231
get_extent() (matplotlib.image.FigureImage method), 1232
get_extent() (matplotlib.image.NonUniformImage method), 1232
get_extent() (matplotlib.offsetbox.AncoredOffsetbox method), 1314
get_extent() (matplotlib.offsetbox.AuxTransformBox method), 1316
get_extent() (matplotlib.offsetbox.DrawingArea method), 1318
get_extent() (matplotlib.offsetbox.OffsetBox method), 1319
get_extent() (matplotlib.offsetbox.OffsetImage method), 1320
get_extent() (matplotlib.offsetbox.TextArea method), 1322
get_extent() (mpl_toolkits.axes_grid1.inset_locator.AnchoredSizeLocator method), 1811
get_extent() (mpl_toolkits.axes_grid1.inset_locator.AnchoredZoomLocator method), 1811
get_extent_offsets() (matplotlib.offsetbox.HPacker method), 1319
get_extent_offsets() (matplotlib.offsetbox.OffsetBox method), 1319
get_extent_offsets() (matplotlib.offsetbox.PaddedBox method), 1321
get_extent_offsets() (matplotlib.offsetbox.VPacker method), 1323
get_extents() (matplotlib.patches.Patch method), 1352
get_extents() (matplotlib.path.Path method), 1370
get_facecolor() (matplotlib.axes.Axes method), 735
get_facecolor() (matplotlib.collections.AsteriskPolygonCollection method), 978
get_facecolor() (matplotlib.collections.BrokenBarHCollection method), 990
get_facecolor() (matplotlib.collections.CircleCollection method), 1002
get_facecolor() (matplotlib.collections.Collection method), 1014
get_facecolor() (matplotlib.collections.EllipseCollection method), 1026
get_facecolor() (matplotlib.collections.EventCollection method), 1039
get_facecolor() (matplotlib.collections.LineCollection method), 1052
get_facecolor() (matplotlib.collections.PatchCollection method), 1063
get_facecolor() (matplotlib.collections.PathCollection method), 1075
get_facecolor() (matplotlib.collections.PolyCollection method), 1087
get_facecolor() (matplotlib.collections.QuadMesh method), 1099
get_facecolor() (matplotlib.collections.RegularPolyCollection method), 1111
get_facecolor() (matplotlib.collections.StarPolygonCollection method), 1123
get_facecolor() (matplotlib.collections.TriMesh method), 1135
get_facecolor() (matplotlib.figure.Figure method), 1197
get_facecolor() (matplotlib.patches.Patch method), 1352
get_facecolor() (mpl_toolkits.mplot3d.art3d.Patch3D method), 1872
get_facecolor() (mpl_toolkits.mplot3d.art3d.Poly3DCollection method), 1873
get_facecolors() (matplotlib.collections.AsteriskPolygonCollection method), 978
get_facecolors() (matplotlib.collections.BrokenBarHCollection method), 990
get_facecolors() (matplotlib.collections.CircleCollection method), 1002
get_facecolors() (matplotlib.collections.Collection method), 1014
get_facecolors() (matplotlib.collections.EllipseCollection method), 1026
get_facecolors() (matplotlib.collections.EventCollection method), 1039
get_facecolors() (matplotlib.collections.LineCollection method), 1052
get_facecolors() (matplotlib.collections.PatchCollection method), 1063
get_facecolors() (matplotlib.collections.PathCollection method), 1075
get_facecolors() (matplotlib.collections.PolyCollection method), 1087
get_facecolors() (matplotlib.collections.QuadMesh method), 1099
get_facecolors() (matplotlib.collections.RegularPolyCollection method), 1111
get_facecolors() (matplotlib.collections.StarPolygonCollection method), 1123
get_facecolors() (matplotlib.collections.TriMesh method), 1135
get_facecolors() (mpl_toolkits.mplot3d.art3d.Poly3DCollection method), 1873
get_family() (matplotlib.patches.Patch method), 1352
get_family() (matplotlib.text.Text method), 1626
get_familyname() (matplotlib.afm.AFM method), 488
get_fc() (matplotlib.axes.Axes method), 736
get_fc() (matplotlib.patches.Patch method), 1352
get_fighheight() (matplotlib.figure.Figure method), 1197
get_figlabels() (in module matplotlib.pyplot), 1486
get_fignums() (in module matplotlib.pyplot), 1486
get_figure() (matplotlib.artist.Artist method), 547
get_figure() (matplotlib.axes.Axes method), 785
get_figure() (matplotlib.axis.Axis method), 881
get_figure() (matplotlib.axis.Tick method), 841
get_figure() (matplotlib.axis.XAxis method), 892
get_figure() (matplotlib.axis.XTick method), 852
get_figure() (matplotlib.axis.YAxis method), 903
get_figure() (matplotlib.axis.YTick method), 864
get_figure() (matplotlib.collections.AsteriskPolygonCollection method), 978
get_figure() (matplotlib.collections.BrokenBarHCollection method), 990
get_figure() (matplotlib.collections.CircleCollection method), 1002
get_figure() (matplotlib.collections.Collection method), 1014
get_figure() (matplotlib.collections.EllipseCollection method), 1026
get_figure() (matplotlib.collections.EventCollection method), 1039
get_figure() (matplotlib.collections.LineCollection method), 1052
get_figure() (matplotlib.collections.PatchCollection method), 1063
get_figure() (matplotlib.collections.PathCollection method), 1075
get_figure() (matplotlib.collections.QuadMesh method), 1087
get_figure() (matplotlib.collections.RegularPolyCollection method), 1111
get_figure() (matplotlib.collections.StarPolygonCollection method), 1124
get_figure() (matplotlib.collections.TriMesh method), 1135
get_fill() (matplotlib.collections.BrokenBarHCollection method), 990
get_fill() (matplotlib.collections.CircleCollection method), 1002
get_fill() (matplotlib.collections.Collection method), 1015
get_fill() (matplotlib.collections.EllipseCollection method), 1026
get_fill() (matplotlib.collections.EventCollection method), 1039
get_fill() (matplotlib.collections.LineCollection method), 1052
get_fill() (matplotlib.collections.PatchCollection method), 1063
get_fill() (matplotlib.collections.PathCollection method), 1075
get_fill() (matplotlib.collections.QuadMesh method), 1087
get_fill() (matplotlib.collections.RegularPolyCollection method), 1111
get_fill() (matplotlib.collections.StarPolygonCollection method), 1124
get_fill() (matplotlib.collections.TriMesh method), 1135
get_fill() (matplotlib.patches.Patch method), 1352
get_fillstyle() (matplotlib.lines.Line2D method), 1247
get_fillstyle() (matplotlib.markers.MarkerStyle method), 1257
get_flat_tri_mask() (matplotlib.tri.TriAnalyzer method), 1679
get_flip_min_max() (in module mpl_toolkits.mplot3d.axis3d), 1871
get_font_properties() (matplotlib.text.Text method), 1626
get_fontconfig_fonts() (in module matplotlib.font_manager), 1224
get_fontconfig_pattern() (matplotlib.font_manager.FontProperties method), 1222
get_fonttext_synonyms() (in module matplotlib.font_manager), 1224
get_fontfamily() (matplotlib.text.Text method), 1626
get_fontname() (matplotlib.afm.AFm method), 488
get_fontname() (matplotlib.text.Text method), 1626
get_fontproperties() (matplotlib.text.Text method), 1626
get_fontsize() (matplotlib.offsetbox.AnnotationBbox method), 1315
get_fontsize() (matplotlib.text.Text method), 1626
get_fontstretch() (matplotlib.text.Text method), 1626
get_fontstyle() (matplotlib.text.Text method), 1626
get_fontvariant() (matplotlib.text.Text method), 1626
get_fontweight() (matplotlib.text.Text method), 1626
get_forced_alpha() (matplotlib.backend_bases.GraphicsContextBase method), 920
get_formatd() (in module matplotlib.mlab), 1296
get_frame() (matplotlib.legend.Legend method), 1240
get_frame_on() (matplotlib.axes.Axes method), 733
get_frame_on() (matplotlib.legend.Legend method), 1240
get_frame_on() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 1852
get_frameon() (matplotlib.figure.Figure method), 1197
get_from_args_and_kwargs() (matplotlib.tri.Triangulation static method), 1672
get_fullname() (matplotlib.afm.AFM method), 488
get_fully_transformed_path() (matplotlib.transforms.TransformedPath method), 1669
get_geometry() (matplotlib.gridspec.GridSpecBase method), 1228
get_geometry() (matplotlib.gridspec.SubplotSpec method), 1229
get_geometry() (mpl_toolkits.axes_grid.axis_divider.SubplotDivider method), 1799
get_gid() (matplotlib.collections.AsteriskPolygonCollection method), 978
get_gid() (matplotlib.collections.BrokenBarHCollection method), 990
get_gid() (matplotlib.collections.CircleCollection method), 1002
get_gid() (matplotlib.collections.Collection method), 1015
get_gid() (matplotlib.collections.EllipseCollection method), 1026
get_gid() (matplotlib.collections.EventCollection method), 1039
get_gid() (matplotlib.collections.LineCollection method), 1052
get_gid() (matplotlib.collections.PathCollection method), 1063
get_gid() (matplotlib.collections.PathCollection method), 1075
get_gid() (matplotlib.collections.PolyCollection method), 1087
get_gid() (matplotlib.collections.QuadMesh method), 1099
get_gid() (matplotlib.collections.RegularPolyCollection method), 1111
get_gid() (matplotlib.collections.StarPolygonCollection method), 1124
get_gid() (matplotlib.collections.TriMesh method), 1135
get_grid_positions() (matplotlib.gridspec.GridSpecBase method), 1228
get_gridlines() (matplotlib.axis.Axis method), 800
get_gridlines() (matplotlib.axis.XAxis method), 825
get_gridlines() (matplotlib.axis.YAxis method), 815
get_gridspec() (matplotlib.gridspec.SubplotSpec method), 1229
get_ha() (matplotlib.text.Text method), 1626
get_hatch() (matplotlib.backend_bases.GraphicsContextBase method), 920
get_hatch() (matplotlib.collections.AsteriskPolygonCollection method), 978
get_hatch() (matplotlib.collections.BrokenBarHCollection method), 990
get_hatch() (matplotlib.collections.CircleCollection method), 1002
get_hatch() (matplotlib.collections.Collection method), 1015
get_hatch() (matplotlib.collections.EllipseCollection method), 1015
get_hatch() (matplotlib.collections.EventCollection method), 1039
get_hatch() (matplotlib.collections.LineCollection method), 1052
get_hatch() (matplotlib.collections.PatchCollection method), 1063
get_hatch() (matplotlib.collections.PathCollection method), 1075
get_hatch() (matplotlib.collections.PolyCollection method), 1087
get_hatch() (matplotlib.collections.QuadMesh method), 1099
get_hatch() (matplotlib.collections.RegularPolyCollection method), 1111
get_hatch() (matplotlib.collections.StarPolygonCollection method), 1124
get_hatch() (matplotlib.collections.TriMesh method), 1135
get_height() (matplotlib.patches.Patch method), 1352
get_height_char() (matplotlib.afm.AFM method), 488
get_height_ratios() (matplotlib.gridspec.GridSpecBase method), 1228
get_hinter() (mpl_toolkits.axes_grid.axis_art.Tick method), 1015
get_horizontal() (mpl_toolkits.axes_grid.axes_divider.Divider method), 1796
get_horizontal_sizes() (mpl_toolkits.axes_grid.axes_divider.Divider method), 1796
get_horizontal_stem_width() (matplotlib.afm.AFM method), 488
get_horizontalalignment() (matplotlib.text.Text method), 1627
get_image_magnification() (matplotlib.backend_bases.RendererBase method), 929
get_images() (matplotlib.axes.Axes method), 776
get_joinstyle() (matplotlib.backend_bases.GraphicsContextBase method), 920
get_joinstyle() (matplotlib.markers.MarkerStyle method), 1257
get_joinstyle() (matplotlib.patches.Patch method), 1352
get_kern() (matplotlib.mathtext.Fonts method), 1264
get_kern() (matplotlib.mathtext.StandardPsFonts method), 1275
get_kern() (matplotlib.mathtext.TruetypeFonts method), 1276
get_kern_dist() (matplotlib.afm.AFM method), 488
get_kern_dist_from_name() (matplotlib.afm.AFM method), 488
get_kerning() (matplotlib.mathtext.Char method), 1262
get_kerning() (matplotlib.mathtext.Node method), 1271
get_label() (matplotlib.artist.Artist method), 550
get_label() (matplotlib.axes.Axes method), 783
get_label() (matplotlib.axes.Axis method), 881
get_label() (matplotlib.axes.Tick method), 841
get_label() (matplotlib.axes.XAxis method), 903
get_label() (matplotlib.axes.XTick method), 853
get_label() (matplotlib.axes.YAxis method), 903
get_label() (matplotlib.axes.YTick method), 864
get_label() (matplotlib.collections.AsteriskPolygonCollection method), 978
get_label() (matplotlib.collections.BrokenBarHCollection method), 990
get_label() (matplotlib.collections.CircleCollection method), 1002
get_label() (matplotlib.collections.Collection method), 1015
get_label() (matplotlib.collections.EllipseCollection method), 1026
get_label() (matplotlib.collections.EventCollection method), 1039
get_label() (matplotlib.collections.LineCollection method), 1052
get_label() (matplotlib.collections.PatchCollection method), 1063
get_label() (matplotlib.collections.PathCollection method), 1075
get_label() (matplotlib.collections.PolyCollection method), 1087
get_label() (matplotlib.collections.QuadMesh method), 1099
get_label() (matplotlib.collections.RegularPolyCollection method), 1111
get_label() (matplotlib.collections.StarPolygonCollection method), 1124
get_label() (matplotlib.collections.TriMesh method), 1135
get_label() (matplotlib.patches.Patch method), 1352
get_label() (matplotlib.patches.FancyBboxPatch method), 1349
get_label() (matplotlib.patches.Rectangle method), 1359
get_label() (matplotlib.patches.FancyBboxPatch method), 1349
get_label() (matplotlib.patches.Rectangle method), 1359
get_label() (matplotlib.patches.FancyBboxPatch method), 1349
get_label() (matplotlib.patches.Rectangle method), 1359
plotlib.collections.BrokenBarHCollection method), 990
get_linestyles() (mat- plotlib.collections.CircleCollection method), 1002
get_linestyles() (matplotlib.collections.Collection method), 1015
get_linestyles() (matplotlib.collections.EllipseCollection method), 1026
get_linestyles() (matplotlib.collections.EventCollection method), 1039
get_linestyles() (matplotlib.collections.LineCollection method), 1052
get_linestyles() (matplotlib.collections.PatchCollection method), 1064
get_linestyles() (matplotlib.collections.PathCollection method), 1075
get_linestyles() (matplotlib.collections.PolyCollection method), 1087
get_linestyles() (matplotlib.collections.QuadMesh method), 1100
get_linestyles() (matplotlib.collections.RegularPolyCollection method), 1112
get_linestyles() (matplotlib.collections.StarPolygonCollection method), 1124
get_linestyles() (matplotlib.collections.TriMesh method), 1136
get_linewidth() (matplotlib.backend_bases.GraphicsContextBase method), 921
get_linewidth() (matplotlib.collections.AsteriskPolygonCollection method), 979
get_linewidth() (matplotlib.collections.BrokenBarHCollection method), 991
get_linewidth() (matplotlib.collections.CircleCollection method), 1003
get_linewidth() (matplotlib.collections.EllipseCollection method), 1015
get_linewidth() (matplotlib.collections.Collection method), 1027
get_linewidth() (matplotlib.collections.Collection method), 1039
get_linewidth() (matplotlib.collections.EventCollection method), 1052
get_linewidth() (matplotlib.collections.PatchCollection method), 1064
get_linewidth() (matplotlib.collections.PathCollection method), 1075
get_linewidth() (matplotlib.collections.PolyCollection method), 1087
get_linewidth() (matplotlib.collections.QuadMesh method), 1100
get_linewidth() (matplotlib.collections.RegularPolyCollection method), 1112
get_linewidth() (matplotlib.collections.StarPolygonCollection method), 1124
get_linewidth() (matplotlib.collections.TriMesh method), 1136
get_linewidth() (matplotlib.lines.Line2D method), 1247
get_linewidth() (matplotlib.patches.Patch method), 1352
get_linewidths() (matplotlib.collections.AsteriskPolygonCollection method), 979
get_linewidths() (matplotlib.collections.BrokenBarHCollection method), 991
get_linewidths() (matplotlib.collections.CircleCollection method), 1003
get_linewidths() (matplotlib.collections.Collection method), 1015
get_linewidths() (matplotlib.collections.EllipseCollection method), 1027
get_linewidths() (matplotlib.collections.Collection method), 1352
plotlib.collections.EventCollection

get_linewidths() (matplotlib.collections.LineCollection method), 1075

get_linewidths() (matplotlib.collections.PatchCollection method), 1075

get_linewidths() (matplotlib.collections.PathCollection method), 1075

get_linewidths() (matplotlib.collections.PolyCollection method), 1075

get_linewidths() (matplotlib.collections.QuadMesh method), 1075

get_loc() (matplotlib.axis.Tick method), 809

get_loc() (matplotlib.axis.XTick method), 810

get_loc() (matplotlib.axis.YTick method), 812

get_loc_in_canvas() (matplotlib.offsetbox.DraggableOffsetBox method), 1317

get_major_locator() (matplotlib.axis.YAxis method), 816

get_major_locator() (matplotlib.axis.XAxis method), 798

get_major_ticks() (matplotlib.axis.XAxis method), 826

get_major_ticks() (matplotlib.axis.YAxis method), 816

get_major_ticks() (mpl_toolkits.mplot3d.axis3d.Axis method), 1870

get_major_ticklabels() (matplotlib.axis.YAxis method), 798

get_major_ticklabels() (matplotlib.axis.XAxis method), 826

get_major_ticklabels() (matplotlib.axis.YAxis method), 816

get_major_ticklines() (matplotlib.axis.XAxis method), 798

get_major_ticklines() (matplotlib.axis.YAxis method), 826

get_major_ticklines() (matplotlib.axis.YAxis method), 816

get_major_ticklocs() (matplotlib.axis.XAxis method), 798

get_major_ticklocs() (matplotlib.axis.XAxis method), 826

get_major_ticklocs() (matplotlib.axis.YAxis method), 816

get_major_ticklocs() (mpl_toolkits.mplot3d.axis3d.Axis method), 1870

get_major_ticklabels() (matplotlib.axis.YAxis method), 798

get_major_ticklabels() (matplotlib.axis.XAxis method), 826

get_major_ticklabels() (matplotlib.axis.YAxis method), 816

get_major_ticklines() (matplotlib.axis.XAxis method), 798

get_major_ticklines() (matplotlib.axis.YAxis method), 826

get_major_ticklines() (matplotlib.axis.YAxis method), 816

get_major_ticklocs() (matplotlib.axis.XAxis method), 798

get_major_ticklocs() (matplotlib.axis.XAxis method), 826

get_major_ticklocs() (matplotlib.axis.YAxis method), 816

get_marker() (matplotlib.lines.Line2D method), 1247

get_marker() (matplotlib.markers.MarkerStyle method), 1257

get_markeredgecolor() (matplotlib.lines.Line2D method), 1247

get_markeredgewidth() (matplotlib.lines.Line2D method), 1248

get_markerfacecolor() (matplotlib.lines.Line2D method), 1248

get_markerfacecoloralt() (matplotlib.lines.Line2D method), 1248

get_markersize() (matplotlib.lines.Line2D method), 1248

get_markevery() (matplotlib.lines.Line2D method), 1248

get_masked_triangles() (matplotlib.tri.Triangulation method), 1672
method), 1380
get_matrix() (matplotlib.projections.polar.PolarAxes.PolarAffine method), 1381
get_matrix() (matplotlib.transforms.Affine2D method), 1663
get_matrix() (matplotlib.transforms.BboxTransform method), 1669
get_matrix() (matplotlib.transforms.BboxTransformFrom method), 1669
get_matrix() (matplotlib.transforms.BboxTransformTo method), 1669
get_matrix() (matplotlib.transforms.BlendedAffine2D method), 1667
get_matrix() (matplotlib.transforms.CompositeAffine2D method), 1668
get_matrix() (matplotlib.transforms.IdentityTransform method), 1664
get_matrix() (matplotlib.transforms.ScaledTranslation method), 1669
get_matrix() (matplotlib.transforms.Transform method), 1658
get_mec() (matplotlib.lines.Line2D method), 1248
get_metrics() (matplotlib.mathtext.Fonts method), 1264
get_mew() (matplotlib.lines.Line2D method), 1248
get_mfc() (matplotlib.lines.Line2D method), 1248
get_mfcalt() (matplotlib.lines.Line2D method), 1248
get_minimumdescent() (matplotlib.offsetbox.TextArea method), 1322
get_minor_formatter() (matplotlib.axis.Axis method), 795
get_minor_formatter() (matplotlib.axis.XAxis method), 826
get_minor_formatter() (matplotlib.axis.YAxis method), 817
get_minorticklabels() (matplotlib.axis.Axis method), 798
get_minorticklabels() (matplotlib.axis.XAxis method), 826
get_minorticklabels() (matplotlib.axis.YAxis method), 817
get_minorticklines() (matplotlib.axis.Axis method), 798
get_minorticklines() (matplotlib.axis.XAxis method), 826
get_minorticklines() (matplotlib.axis.YAxis method), 817
get_minorticklocs() (matplotlib.axis.Axis method), 799
get_minorticklocs() (matplotlib.axis.XAxis method), 827
get_minorticklocs() (matplotlib.axis.YAxis method), 817
get_minpos() (matplotlib.axis.Axis method), 801
get_minpos() (matplotlib.axis.XAxis method), 827
get_minpos() (matplotlib.axis.YAxis method), 817
get_ms() (matplotlib.lines.Line2D method), 1248
get_multilinebaseline() (matplotlib.offsetbox.TextArea method), 1322
get_mutation_aspect() (matplotlib.patches.FancyArrowPatch method), 1346
get_mutation_aspect() (matplotlib.patches.FancyBboxPatch method), 1349
get_mutation_scale() (matplotlib.patches.FancyArrowPatch method), 1346
get_mutation_scale() (matplotlib.patches.FancyBboxPatch method), 1349
get_name() (matplotlib.font_manager.FontProperties method), 1222
get_name() (matplotlib.text.Text method), 1627
get_name_char() (matplotlib.afm.AFM method),
get_named_colors_mapping() (in module matplotlib.colors), 1162
get_navigate() (matplotlib.axes.Axes method), 771
get_navigate_mode() (matplotlib.axes.Axes method), 771
get_numpoints() (matplotlib.legend_handler.HandlerLineCollection method), 1243
get_numpoints() (matplotlib.legend_handler.HandlerNpoints method), 1243
get_numpoints() (matplotlib.legend_handler.HandlerRegularPolyCollection method), 1244
get_numsides() (matplotlib.collections.AsteriskPolygonCollection method), 979
get_numsides() (matplotlib.collections.RegularPolyCollection method), 1112
get_numsides() (matplotlib.collections.StarPolygonCollection method), 1124
get_offset() (matplotlib.offsetbox.AuxTransformBox method), 1316
get_offset() (matplotlib.offsetbox.DrawingArea method), 1318
get_offset() (matplotlib.offsetbox.OffsetBox method), 1319
get_offset() (matplotlib.offsetbox.OffsetImage method), 1320
get_offset() (matplotlib.offsetbox.TextArea method), 1322
get_offset() (matplotlib.ticker.FixedFormatter method), 1638
get_offset() (matplotlib.ticker.Formatter method), 1638
get_offset() (matplotlib.ticker.ScalarFormatter method), 1639
get_offset_position() (matplotlib.collections.AsteriskPolygonCollection method), 979
get_offset_position() (matplotlib.collections.BrokenBarHCollection method), 991
get_offset_position() (matplotlib.collections.CircleCollection method), 1003
get_offset_position() (matplotlib.collections.Collection method), 1015
get_offset_position() (matplotlib.collections.EllipseCollection method), 1027
get_offset_position() (matplotlib.collections.EventCollection method), 1039
get_offset_position() (matplotlib.collections.LineCollection method), 1052
get_offset_position() (matplotlib.collections.PatchCollection method), 1064
get_offset_position() (matplotlib.collections.PathCollection method), 1075
get_offset_position() (matplotlib.collections.PolyCollection method), 1087
get_offset_position() (matplotlib.collections.QuadMesh method), 1100
get_offset_position() (matplotlib.collections.RegularPolyCollection method), 1112
get_offset_position() (matplotlib.collections.StarPolygonCollection method), 1124
get_offset_position() (matplotlib.collections.TriMesh method), 1136
get_offset_text() (matplotlib.axis.Axis method), 799
get_offset_text() (matplotlib.axis.XAxis method), 827
get_offset_text() (matplotlib.axis.YAxis method), 817
get_offset_transform() (matplotlib.collections.AsteriskPolygonCollection method), 979
get_offset_transform() (matplotlib.collections.BrokenBarHCollection method), 991
get_offset_transform() (matplotlib.collections.CircleCollection method), 1003
get_patch_transform() (matplotlib.patches.YAArrow method), 1365
get_patch_transform() (matplotlib.spines.Spine method), 1616
get_patch_verts() (in module mpl_toolkits.mplot3d.art3d), 1874
get_patches() (matplotlib.legend.Legend method), 1240
get_path() (matplotlib.lines.Line2D method), 1248
get_path() (matplotlib.markers.MarkerStyle method), 1257
get_path() (matplotlib.patches.Arrow method), 1328
get_path() (matplotlib.patches.Ellipse method), 1343
get_path() (matplotlib.patches.FancyArrowPatch method), 1346
get_path() (matplotlib.patches.FancyBboxPatch method), 1349
get_path() (matplotlib.patches.Patch method), 1353
get_path() (matplotlib.patches.PathPatch method), 1356
get_path() (matplotlib.patches.Polygon method), 1357
get_path() (matplotlib.patches.Rectangle method), 1359
get_path() (matplotlib.patches.RegularPolygon method), 1361
get_path() (matplotlib.patches.Shadow method), 1362
get_path() (matplotlib.patches.Wedge method), 1363
get_path() (matplotlib.patches.YAArrow method), 1365
get_path() (mpl_toolkits.axes_grid1.inset_locator.BboxConnector method), 1814
get_path() (mpl_toolkits.axes_grid1.inset_locator.BboxConnectorPatch method), 1815
get_path() (mpl_toolkits.axes_grid1.inset_locator.BboxPath method), 1816
get_path() (mpl_toolkits.mplot3d.art3d.Patch3D method), 1872
get_path_collection_extents() (in module matplotlib.path), 1373
get_path_effects() (matplotlib.artist.Artist method), 546
get_path_effects() (matplotlib.axes.Axes method), 784
get_path_effects() (matplotlib.axis.Axis method), 881
get_path_effects() (matplotlib.axis.Tick method), 841
get_path_effects() (matplotlib.axis.XAxis method), 892
get_path_effects() (matplotlib.axis.XTick method), 853
get_path_effects() (matplotlib.axis.YAxis method), 903
get_path_effects() (matplotlib.axis.YTick method), 864
get_path_effects() (matplotlib.collections.AsteriskPolygonCollection method), 979
get_path_effects() (matplotlib.collections.BrokenBarHCollection method), 991
get_path_effects() (matplotlib.collections.CircleCollection method), 1003
get_path_effects() (matplotlib.collections.Collection method), 1015
get_path_effects() (matplotlib.collections.EllipseCollection method), 1027
get_path_effects() (matplotlib.collections.EventCollection method), 1040
get_path_effects() (matplotlib.collections.LineCollection method), 1052
get_path_effects() (matplotlib.collections.PatchCollection method), 1064
get_path_effects() (matplotlib.collections.PathCollection method), 1076
get_path_effects() (matplotlib.collections.PolyCollection method), 1088
get_path_effects() (matplotlib.collections.QuadMesh method), 1100
get_path_effects() (matplotlib.collections.RegularPolyCollection method), 1112
get_path_effects() (matplotlib.collections.StarPolygonCollection method), 1124

Index 3065
get_pickradius() (matplotlib.collections.LineCollection method), 1053
get_pickradius() (matplotlib.collections.PatchCollection method), 1064
get_pickradius() (matplotlib.collections.PathCollection method), 1076
get_pickradius() (matplotlib.collections.PolyCollection method), 1088
get_pickradius() (matplotlib.collections.QuadMesh method), 1100
get_pickradius() (matplotlib.collections.RegularPolyCollection method), 1112
get_pickradius() (matplotlib.collections.StarPolygonCollection method), 1124
get_pickradius() (matplotlib.collections.TriMesh method), 1136
get_pickradius() (matplotlib.collections.AsteriskPolygonCollection method), 1124
get_pickradius() (matplotlib.collections.BrokenBarHCollection method), 991
get_pickradius() (matplotlib.collections.CircleCollection method), 1003
get_pickradius() (matplotlib.collections.EventCollection method), 1040
get_pickradius() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 1852
get_projection_class() (in module matplotlib.projections), 1379
get_projection_class() (matplotlib.projections.ProjectionRegistry method), 1379
get_projection_names() (in module matplotlib.projections), 1379
get_projection_names() (matplotlib.projections.ProjectionRegistry method), 1379
get_prop_tuple() (matplotlib.text.Text method), 1627
get_prop_tuple() (matplotlib.text.TextWithDash method), 1633
get_radius() (matplotlib.patches.Circle method), 1337
get_rasterization_zorder() (matplotlib.axes.Axes method), 777
get_rasterized() (matplotlib.artist.Artist method), 545
get_rasterized() (matplotlib.axes.Axes method), 784
get_rasterized() (matplotlib.axis.Axis method), 881
get_rasterized() (matplotlib.axis.Tick method), 841
get_rasterized() (matplotlib.axis.XAxis method), 893
get_rasterized() (matplotlib.axis.YAxis method), 904
get_rasterized() (matplotlib.axis.XTick method), 853
get_rasterized() (matplotlib.axis.YTick method), 864
get_rasterized() (matplotlib.collections.AsteriskPolygonCollection method), 979
get_rasterized() (matplotlib.collections.BrokenBarHCollection method), 991
get_rasterized() (matplotlib.collections.CircleCollection method), 1003
get_rasterized() (matplotlib.collections.Collection method), 1015
get_rasterized() (matplotlib.collections.EllipseCollection method), 1027
get_rasterized() (matplotlib.collections.EventCollection method), 1040
get_rasterized() (matplotlib.collections.LineCollection method), 1040
get_rasterized() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 1852
get_pickradius() (matplotlib.collections.LineCollection method), 1053
get_pickradius() (matplotlib.collections.PatchCollection method), 1064
get_pickradius() (matplotlib.collections.PathCollection method), 1076
get_pickradius() (matplotlib.collections.PolyCollection method), 1088
get_pickradius() (matplotlib.collections.QuadMesh method), 1100
get_pickradius() (matplotlib.collections.RegularPolyCollection method), 1112
get_pickradius() (matplotlib.collections.StarPolygonCollection method), 1124
get_pickradius() (matplotlib.collections.TriMesh method), 1136
get_pickradius() (matplotlib.collections.AsteriskPolygonCollection method), 1124
get_pickradius() (matplotlib.collections.BrokenBarHCollection method), 991
get_pickradius() (matplotlib.collections.CircleCollection method), 1003
get_pickradius() (matplotlib.collections.EventCollection method), 1040
get_pickradius() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 1852
get_projection_class() (in module matplotlib.projections), 1379
get_projection_class() (matplotlib.projections.ProjectionRegistry method), 1379
get_projection_names() (in module matplotlib.projections), 1379
get_projection_names() (matplotlib.projections.ProjectionRegistry method), 1379
get_prop_tuple() (matplotlib.text.Text method), 1627
get_prop_tuple() (matplotlib.text.TextWithDash method), 1633
get_radius() (matplotlib.patches.Circle method), 1337
get_rasterization_zorder() (matplotlib.axes.Axes method), 777
get_rasterized() (matplotlib.artist.Artist method), 545
get_rasterized() (matplotlib.axes.Axes method), 784
get_rasterized() (matplotlib.axis.Axis method), 881
get_rasterized() (matplotlib.axis.Tick method), 841
get_rasterized() (matplotlib.axis.XAxis method), 893
get_rasterized() (matplotlib.axis.YAxis method), 904
get_rasterized() (matplotlib.axis.XTick method), 853
get_rasterized() (matplotlib.axis.YTick method), 864
get_rasterized() (matplotlib.collections.AsteriskPolygonCollection method), 979
get_rasterized() (matplotlib.collections.BrokenBarHCollection method), 991
get_rasterized() (matplotlib.collections.CircleCollection method), 1003
get_rasterized() (matplotlib.collections.Collection method), 1015
get_rasterized() (matplotlib.collections.EllipseCollection method), 1027
get_rasterized() (matplotlib.collections.EventCollection method), 1040
get_rasterized() (matplotlib.collections.LineCollection method),
<table>
<thead>
<tr>
<th>Method</th>
<th>Class</th>
<th>Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_rasterized()</td>
<td>matplotlib.collections.PatchCollection</td>
<td>1053</td>
</tr>
<tr>
<td>get_rasterized()</td>
<td>matplotlib.collections.PathCollection</td>
<td>1076</td>
</tr>
<tr>
<td>get_rasterized()</td>
<td>matplotlib.collections.PolyCollection</td>
<td>1088</td>
</tr>
<tr>
<td>get_rasterized()</td>
<td>matplotlib.collections.QuadMesh</td>
<td>1100</td>
</tr>
<tr>
<td>get_rasterized()</td>
<td>matplotlib.collections.RegularPolyCollection</td>
<td>1112</td>
</tr>
<tr>
<td>get_rasterized()</td>
<td>matplotlib.collections.StarPolygonCollection</td>
<td>1124</td>
</tr>
<tr>
<td>get_rasterized()</td>
<td>matplotlib.collections.TriMesh</td>
<td>1136</td>
</tr>
<tr>
<td>get_recursive_filelist()</td>
<td></td>
<td>961</td>
</tr>
<tr>
<td>get_registered_canvas_class()</td>
<td></td>
<td>933</td>
</tr>
<tr>
<td>get_renderer()</td>
<td>matplotlib.axes.Axes</td>
<td>777</td>
</tr>
<tr>
<td>get_results()</td>
<td>matplotlib.mathtext.Fonts</td>
<td>1265</td>
</tr>
<tr>
<td>get_results()</td>
<td>matplotlib.mathtext.MathtextBackendPs</td>
<td>1270</td>
</tr>
<tr>
<td>get_results()</td>
<td>matplotlib.mathtext.MathtextBackendSvg</td>
<td>1271</td>
</tr>
<tr>
<td>get_rgb()</td>
<td>matplotlib.backend_bases.GraphicsContextBase</td>
<td>921</td>
</tr>
<tr>
<td>get_rlabel_position()</td>
<td>matplotlib.projections.polar.PolarAxes</td>
<td>1382</td>
</tr>
<tr>
<td>get_rotate_label()</td>
<td>mpl_toolkits.mplot3d.axis3d.Axis</td>
<td>1870</td>
</tr>
<tr>
<td>get_rotation()</td>
<td>matplotlib.text</td>
<td>1627</td>
</tr>
<tr>
<td>get_rotation()</td>
<td>matplotlib.collections.AsteriskPolygonCollection</td>
<td>979</td>
</tr>
<tr>
<td>get_rotation()</td>
<td>matplotlib.collections.RegularPolyCollection</td>
<td>1112</td>
</tr>
<tr>
<td>get_rotation()</td>
<td>matplotlib.collections.StarPolygonCollection</td>
<td>1125</td>
</tr>
<tr>
<td>get_rotation()</td>
<td>matplotlib.collections.TriMesh</td>
<td>1136</td>
</tr>
<tr>
<td>get_rotation()</td>
<td>matplotlib.collections.EventCollection</td>
<td>1040</td>
</tr>
<tr>
<td>get_rotation()</td>
<td>matplotlib.collections.LineCollection</td>
<td>1053</td>
</tr>
<tr>
<td>get_samples()</td>
<td>matplotlib.font_manager.FontProperties</td>
<td>1222</td>
</tr>
<tr>
<td>get_shared_x_axes()</td>
<td>matplotlib.axes.Axes</td>
<td>767</td>
</tr>
<tr>
<td>get_shared_y_axes()</td>
<td>matplotlib.axes.Axes</td>
<td>767</td>
</tr>
<tr>
<td>get_siblings()</td>
<td>matplotlib.cbook.Grouper</td>
<td>955</td>
</tr>
<tr>
<td>get_size()</td>
<td>matplotlib.font_manager.FontProperties</td>
<td>1222</td>
</tr>
<tr>
<td>get_size()</td>
<td>matplotlib.text.Text</td>
<td>1627</td>
</tr>
</tbody>
</table>
get_size_in_points() (matplotlib.font_manager.FontProperties method), 1222
get_size_inches() (matplotlib.figure.Figure method), 1197
get_sized_alternatives_for_symbol() (matplotlib.mathtext.BakomaFonts method), 1261
get_sized_alternatives_for_symbol() (matplotlib.mathtext.Fonts method), 1265
get_sized_alternatives_for_symbol() (matplotlib.mathtext.StixFonts method), 1275
get_sized_alternatives_for_symbol() (matplotlib.mathtext.UnicodeFonts method), 1276
get_sizes() (matplotlib.collections.AsteriskPolygonCollection method), 979
get_sizes() (matplotlib.collections.BrokenBarHCollection method), 991
get_sizes() (matplotlib.collections.CircleCollection method), 1003
get_sizes() (matplotlib.collections.PathCollection method), 1076
get_sizes() (matplotlib.collections.PolyCollection method), 1088
get_sizes() (matplotlib.collections.RegularPolyCollection method), 1112
get_sizes() (matplotlib.collections.StarPolygonCollection method), 1125
get_sizes() (matplotlib.collections.TriMesh method), 1136
get_sketch_params() (matplotlib.artist.Artist method), 545
get_sketch_params() (matplotlib.axes.Axes method), 782
get_sketch_params() (matplotlib.axis.Axis method), 881
get_sketch_params() (matplotlib.axis.Tick method), 842
get_sketch_params() (matplotlib.axis.XAxis method), 893
get_sketch_params() (matplotlib.axis.XTick method), 853
get_sketch_params() (matplotlib.axis.YAxis method), 904
get_sketch_params() (matplotlib.axis.YTick method), 864
get_sketch_params() (matplotlib.backend_bases.GraphicsContextBase method), 921
get_sketch_params() (matplotlib.collections.AsteriskPolygonCollection method), 979
get_sketch_params() (matplotlib.collections.BrokenBarHCollection method), 991
get_sketch_params() (matplotlib.collections.CircleCollection method), 1003
get_sketch_params() (matplotlib.collections.LineCollection method), 1053
get_sketch_params() (matplotlib.collections.PatchCollection method), 1064
get_sketch_params() (matplotlib.collections.PathCollection method), 1076
get_sketch_params() (matplotlib.collections.QuadMesh method), 1100
get_sketch_params() (matplotlib.collections.RegularPolyCollection method), 1112
get_sketch_params() (matplotlib.collections.StarPolygonCollection method), 1125
get_sketch_params() (matplotlib.collections.TriMesh method), 1136
get_slant() (matplotlib.font_manager.FontProperties method), 1222
get_smart_bounds() (matplotlib.axis.Axis method), 904
get_smart_bounds() (matplotlib.axis.XAxis method), 827
get_smart_bounds() (matplotlib.axis.YAxis method), 818
get_smart_bounds() (matplotlib.spines.Spines method), 1616
get_snap() (matplotlib.artist.Artist method), 543
get_snap() (matplotlib.axes.Axes method), 784
get_snap() (matplotlib.axis.Axis method), 882
get_snap() (matplotlib.axis.Tick method), 842
get_snap() (matplotlib.axis.XAxis method), 893
get_snap() (matplotlib.axis.YAxis method), 904
get_snap() (matplotlib.axis.YTick method), 865
get_snap() (matplotlib.backend_bases.GraphicsContextBase method), 921
get_snap() (matplotlib.collections.AsteriskPolygonCollection method), 980
get_snap() (matplotlib.collections.BrokenBarHCollection method), 991
get_snap() (matplotlib.collections.CircleCollection method), 1003
get_snap() (matplotlib.collections.Collection method), 1016
get_snap() (matplotlib.collections.EllipseCollection method), 1027
get_snap() (matplotlib.collections.EventCollection method), 1040
get_snap() (matplotlib.collections.LineCollection method), 1053
get_snap() (matplotlib.collections.PatchCollection method), 1065
get_snap() (matplotlib.collections.PathCollection method), 1076
get_snap() (matplotlib.collections.PolyCollection method), 1088
get_snap() (matplotlib.collections.QuadMesh method), 1100
get_snap() (matplotlib.collections.RegularPolyCollection method), 1113
get_snap() (matplotlib.collections.StarPolygonCollection method), 1125
get_snap() (matplotlib.collections.TriMesh method), 1137
get_snap_threshold() (matplotlib.markers.MarkerStyle method), 1257
get_solid_capstyle() (matplotlib.lines.Line2D method), 1248
get_solid_joinstyle() (matplotlib.lines.Line2D method), 1248
get_sparse_matrix() (in module matplotlib.mlab), 1296
get_spine_transform() (matplotlib.spines.Spines method), 1616
get_split_ind() (in module matplotlib.cbook), 961
get_state() (matplotlib.mathtext.Parser method), 1272
get_str_bbox() (matplotlib.afm.AFM method), 1285
get_str_bbox_and_descent() (matplotlib.afm.AFM method), 1288
get_style() (matplotlib.font_manager.FontProperties method), 1222
get_style() (matplotlib.text.Text method), 1627
get_subplot_params() (matplotlib.gridspec.GridSpec method), 1227
get_subplot_params() (matplotlib.gridspec.GridSpecBase method), 1228
get_subplotspec() (matplotlib.gridspec.GridSpecFromSubplotSpec method), 1228
get_subplotspec() (mpl_toolkits.axes_grid.axes_divider.AxesDivider method), 1798
get_subplotspec() (mpl_toolkits.axes_grid.axes_divider.SubplotDivider method), 1799
get_subplotspec_list() (in module matplotlib.tight_layout), 1647
get_supported_filetypes() (matplotlib.backend_bases.FigureCanvasBase class method), 915
get_supported_filetypes_grouped() (matplotlib.backend_bases.FigureCanvasBase class method), 915
get_test_data() (in module mpl_toolkits.mplot3d.axes3d), 1869
get_text() (matplotlib.offsetbox.TextArea method), 1322
get_texmanager() (matplotlib.backend_bases.RendererBase method), 929
Index 3073

1016
get_transformed_clip_path_and_affine() (matplotlib.collections.EllipseCollection method), 1028
get_transformed_clip_path_and_affine() (matplotlib.collections.EllipseCollection method), 1028
get_transformed_clip_path_and_affine() (matplotlib.collections.EventCollection method), 1041
get_transformed_clip_path_and_affine() (matplotlib.collections.EventCollection method), 1040
get_transformed_clip_path_and_affine() (matplotlib.collections.LineCollection method), 1053
get_transformed_clip_path_and_affine() (matplotlib.collections.PatchCollection method), 1065
get_transformed_clip_path_and_affine() (matplotlib.collections.PatchCollection method), 1053
get_transformed_clip_path_and_affine() (matplotlib.collections.PatchCollection method), 1065
get_transformed_clip_path_and_affine() (matplotlib.collections.PatchCollection method), 1077
get_transformed_clip_path_and_affine() (matplotlib.collections.PatchCollection method), 1089
get_transformed_clip_path_and_affine() (matplotlib.collections.QuadMesh method), 1101
get_transformed_clip_path_and_affine() (matplotlib.collections.QuadMesh method), 1101
get_transformed_clip_path_and_affine() (matplotlib.collections.QuadMesh method), 1113
get_transformed_clip_path_and_affine() (matplotlib.collections.QuadMesh method), 1125
get_transformed_clip_path_and_affine() (matplotlib.collections.StarPolygonCollection method), 1137
get_transformed_clip_path_and_affine() (matplotlib.collections.StarPolygonCollection method), 1125
get_transformed_path_and_affine() (matplotlib.transforms.TransformedPath method), 1669
get_transformed_path_and_affine() (matplotlib.transforms.TransformedPath method), 1669
get_transformed_path_and_affine() (matplotlib.transforms.TransformedPath method), 1669
get_transformed_path_and_affine() (matplotlib.transforms.TransformedPath method), 1669
get_transformed_points_and_affine() (matplotlib.transforms.TransformedPath method), 1669
get_unit() (matplotlib.text.OffsetFrom method), 1264
get_unit() (matplotlib.text.Text method), 1267
get_unit() (matplotlib.text.Text method), 1267
get_unit() (matplotlib.text.Text method), 1267
get_underline_thickness() (matplotlib.text.AFM method), 488
get_underline_thickness() (matplotlib.text.Fonts method), 1265
get_underline_thickness() (matplotlib.text.StandardPsFonts method), 1275
get_underline_thickness() (matplotlib.text.TruetypeFonts method), 1276
get_transforms() (matplotlib.collections.AsteriskPolygonCollection method), 980
get_transforms() (matplotlib.collections.BrokenBarHCollection method), 992
get_transforms() (matplotlib.collections.CircleCollection method), 1004
get_transforms() (matplotlib.collections.Collection method), 1016
get_transforms() (matplotlib.collections.EllipseCollection method), 1028
get_transforms() (matplotlib.collections.EventCollection method), 1041
get_transforms() (matplotlib.collections.EventCollection method), 1040
get_transforms() (matplotlib.collections.LineCollection method), 1053
get_transforms() (matplotlib.collections.PatchCollection method), 1065
get_transforms() (matplotlib.collections.PatchCollection method), 1053
get_transforms() (matplotlib.collections.PatchCollection method), 1065
get_transforms() (matplotlib.collections.PatchCollection method), 1077
get_transforms() (matplotlib.collections.PolyCollection method), 1089
get_transforms() (matplotlib.collections.QuadMesh method), 1101
get_transforms() (matplotlib.collections.QuadMesh method), 1101
get_transforms() (matplotlib.collections.QuadMesh method), 1113
get_transforms() (matplotlib.collections.StarPolygonCollection method), 1125
get_transforms() (matplotlib.collections.StarPolygonCollection method), 1125
get_trifinder() (matplotlib.tri.Triangulation method), 1672
get_unitless_position() (matplotlib.text.Text method), 1627
get_unitless_position() (matplotlib.text.Text method), 1627
get_unitless_position() (matplotlib.text.Text method), 1627
get_unitless_position() (matplotlib.text.Text method), 1627
get_view_interval() (matplotlib.axis.XAxis method), 809
get_view_interval() (matplotlib.axis.XTick method), 829
get_view_interval() (matplotlib.axis.YAxis method), 811
get_view_interval() (matplotlib.axis.YTick method), 819
get_view_interval() (mpl_toolkits.mplot3d.axis3d.Axis method), 1870
get_visible() (matplotlib.artist.Artist method), 543
get_visible() (matplotlib.axes.Axes method), 784
get_visible() (matplotlib.axis.Axis method), 882
get_visible() (matplotlib.axis.XAxis method), 894
get_visible() (matplotlib.axis.XTick method), 854
get_visible() (matplotlib.axis.YAxis method), 905
get_visible() (matplotlib.axis.YTick method), 865
get_visible() (matplotlib.collections.AsteriskPolygonCollection method), 980
get_visible() (matplotlib.collections.BrokenBarHCollection method), 992
get_visible() (matplotlib.collections.CircleCollection method), 1004
get_visible() (matplotlib.collections.Collection method), 1016
get_visible() (matplotlib.collections.EllipseCollection method), 1028
get_visible() (matplotlib.collections.EventCollection method), 1041
get_visible() (matplotlib.collections.LineCollection method), 1053
get_visible() (matplotlib.collections.PatchCollection method), 1065
get_visible() (matplotlib.collections.PathCollection method), 1077
get_visible() (matplotlib.collections.PolyCollection method), 1089
get_visible() (matplotlib.collections.QuadMesh method), 1101
get_visible() (matplotlib.collections.RegularPolyCollection method), 1113
get_visible() (matplotlib.collections.StarPolygonCollection method), 1125
get_visible() (matplotlib.collections.TriMesh method), 1137
get_visible_children() (matplotlib.offsetbox.OffsetBox method), 1319
get_vsize_hsize() (mpl_toolkits.axes_grid.axes_divider.Divider method), 1797
get_w_lims() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 1852
get_weight() (matplotlib.afm.AFM method), 488
get_weight() (matplotlib.font_manager.FontProperties method), 1222
get_weight() (matplotlib.text.Text method), 1627
get_width() (matplotlib.patches.FancyBboxPatch method), 1349
get_width() (matplotlib.patches.Rectangle method), 1359
get_width_char() (matplotlib.afm.AFM method), 488
get_width_from_char_name() (matplotlib.afm.AFM method), 488
get_width_height() (matplotlib.backend_bases.FigureCanvasBase method), 915
get_width_ratios() (matplotlib.gridspec.GridSpecBase method), 1228
get_window_extent() (matplotlib.artist.Artist method), 546
get_window_extent() (matplotlib.axes.Axes method), 777
get_window_extent() (matplotlib.axis.Axis method), 883
get_window_extent() (matplotlib.axis.Tick method), 843
get_window_extent() (matplotlib.axis.XAxis method), 894
get_window_extent() (matplotlib.axis.XTick method), 854
get_window_extent() (matplotlib.axis.YAxis method), 905
get_window_extent() (matplotlib.axis.YTick method), 865
get_window_extent() (matplotlib.zoom.PlotZoom method), 1117
is_figure_set() (matplotlib.collections.EllipseCollection method), 1028
is_figure_set() (matplotlib.collections.EventCollection method), 1041
is_figure_set() (matplotlib.collections.LineCollection method), 1054
is_figure_set() (matplotlib.collections.PatchCollection method), 1065
is_figure_set() (matplotlib.collections.PolyCollection method), 1089
is_figure_set() (matplotlib.collections.QuadMesh method), 1101
is_figure_set() (matplotlib.collections.RegularPolyCollection method), 1113
is_figure_set() (matplotlib.collections.StartPolygonCollection method), 1126
is_figure_set() (matplotlib.collections.TriMesh method), 1137
is_filled() (matplotlib.markers.MarkerStyle method), 1257
is_frame_like() (matplotlib.spines.Spine method), 1616
is_gray() (matplotlib.colors.Colormap method), 1153
is_hashable() (in module matplotlib.cbook), 962
is_horizontal() (matplotlib.collections.EventCollection method), 1041
is_math_text() (in module matplotlib.cbook), 962
is_math_text() (matplotlib.text.Text static method), 1628
is_missing() (matplotlib.cbook.converter method), 959
is_numlike() (in module matplotlib.cbook), 962
is_numlike() (matplotlib.units.ConversionInterface static method), 1686
is_opentype_cff_font() (in module matplotlib.font_manager), 1224
is_overunder() (matplotlib.mathtext.Parser method), 1272
is_saving() (matplotlib.backend_bases.FigureCanvasBase method), 915
is_scalar() (in module matplotlib.cbook), 962
is_scalar_or_string() (in module matplotlib.cbook), 962
is_separable (matplotlib.scale.InvertedLogTransformBase attribute), 1606
is_separable (matplotlib.scale.InvertedSymmetricalLogTransform attribute), 1606
is_separable (matplotlib.scale.LogisticTransform attribute), 1610
is_separable (matplotlib.scale.LogScale.LogTransformBase attribute), 1609
is_separable (matplotlib.scale.SymmetricalLogScale.InvertedSymmetricalLogTransform attribute), 1612
is_separable (matplotlib.scale.SymmetricalLogScale.SymmetricalLogTransform attribute), 1613
is_separable (matplotlib.transforms.Transform attribute), 1613
is_sequence_of_strings() (in module matplotlib.cbook), 962
is_slanted() (matplotlib.mathtext.Char method), 1262
is_slanted() (matplotlib.mathtext.Parser method), 1272
is_string_like() (in module matplotlib.cbook), 962
is_transform_set() (matplotlib.artist.Artist method), 549
is_transform_set() (matplotlib.axes.Axes method), 786
is_transform_set() (matplotlib.axis.Axis method), 883
is_transform_set() (matplotlib.axis.Tick method), 843
is_transform_set() (matplotlib.axis.XAxis method), 895
is_transform_set() (matplotlib.axis.XTick method), 855
is_transform_set() (matplotlib.axis.YAxis method), 906

Index 3083
is_transform_set() (matplotlib.axis.YTick method), 866
is_transform_set() (matplotlib.collections.AsteriskPolygonCollection method), 980
is_transform_set() (matplotlib.collections.BrokenBarHCollection method), 992
is_transform_set() (matplotlib.collections.CircleCollection method), 1004
is_transform_set() (matplotlib.collections.Collection method), 1016
is_transform_set() (matplotlib.collections.EllipseCollection method), 1028
is_transform_set() (matplotlib.collections.EventCollection method), 1041
is_transform_set() (matplotlib.collections.LineCollection method), 1054
is_transform_set() (matplotlib.collections.PatchCollection method), 1065
is_transform_set() (matplotlib.collections.PathCollection method), 1077
is_transform_set() (matplotlib.collections.PolyCollection method), 1089
is_transform_set() (matplotlib.collections.QuadMesh method), 1101
is_transform_set() (matplotlib.collections.RegularPolyCollection method), 1113
is_transform_set() (matplotlib.collections.StarPolygonCollection method), 1126
is_transform_set() (matplotlib.collections.TriMesh method), 1137
is_unit() (matplotlib.transforms.BboxBase method), 1653
is_writable_file_like() (in module matplotlib.cbook), 962
iscolor() (in module mpl_toolkits.mplot3d.art3d), 1875
ishold() (in module matplotlib.pylab), 1503
ishold() (matplotlib.axes.Axes method), 790
isinteractive() (in module matplotlib.pylab), 1503
ispower2() (in module matplotlib.mlab), 1298
issubclass_safe() (in module matplotlib.mlab), 962
isvector() (in module matplotlib.mlab), 1298
iter_segments() (matplotlib.path.Path method), 1371
iter_ticks() (matplotlib.axis.Axes method), 800
iter_ticks() (matplotlib.axis.XAxis method), 829
iter_ticks() (matplotlib.axis.YAxis method), 819
iterable() (in module matplotlib.mlab), 962
join() (in module matplotlib.pyplot), 1503
join() (matplotlib.cbook.Grouper method), 955
joined() (matplotlib.cbook.Grouper method), 955
JPG, 3017
juggle_axes() (in module mpl_toolkits.mplot3d.art3d), 1875
K
Kern (class in matplotlib.mathtext), 1267
kern() (matplotlib.mathtext.HList method), 1267
key_press() (matplotlib.backend_bases.FigureManagerBase method), 919
key_press_event() (matplotlib.backend_bases.FigureCanvasBase method), 915
key_press_handler() (in module matplotlib.backend_bases), 933
key_release_event() (matplotlib.backend_bases.FigureCanvasBase method), 915
KeyEvent (class in matplotlib.backend_bases), 923
kwdoc() (in module matplotlib.artist), 553
L
l1norm() (in module matplotlib.mlab), 1298
l2norm() (in module matplotlib.mlab), 1298
label_minor() (matplotlib.ticker.LogFormatter method, 1641
LABELPAD (mpl_toolkits.axes_grid.axis_artist.Artist attribute), 1802
Lasso (class in matplotlib.widgets), 1692
LassoSelector (class in matplotlib.widgets), 1692

lastevent (matplotlib.backend_bases.LocationEvent attribute), 924
leave_notify_event() (matplotlib.backend_bases.FigureCanvasBase method), 915
Legend (class in matplotlib.legend), 1237
legend entry, 68
legend handle, 68
legend key, 68
legend label, 68
legend() (in module matplotlib.pyplot), 1503
legend() (matplotlib.axes.Axes method), 744
legend() (matplotlib.figure.Figure method), 1198
legend_artist() (matplotlib.legend_handler.HandlerBase method), 1242
less_simple_linear_interpolation() (in module matplotlib.mlab), 1298
LightSource (class in matplotlib.colors), 1153
limit_range_for_scale() (matplotlib.axis.Axis method), 806
limit_range_for_scale() (matplotlib.axis.XAxis method), 829
limit_range_for_scale() (matplotlib.axis.YAxis method), 820
limit_range_for_scale() (matplotlib.scale.LogitScale method), 1610
limit_range_for_scale() (matplotlib.scale.LogScale method), 1609
limit_range_for_scale() (matplotlib.scale.ScaleBase method), 1611
Line2D (class in matplotlib.lines), 1245
line2d() (in module mpl_toolkits.mplot3d.projection), 1876
line2d_dist() (in module mpl_toolkits.mplot3d.projection), 1876
line2d_segmented_dist() (in module mpl_toolkits.mplot3d.projection), 1876
Line3D (class in mpl_toolkits.mplot3d.art3d), 1871
Line3DCollection (class in mpl_toolkits.mplot3d.art3d), 1871
line_2d_to_3d() (in module mpl_toolkits.mplot3d.art3d), 1875
line_collection_2d_to_3d() (in module mpl_toolkits.mplot3d.art3d), 1875
linear_spine() (matplotlib.spines.Spine class method), 1617
LinearLocator (class in matplotlib.transforms), 1643
LinearScale (class in matplotlib.scale), 1606
LinearSegmentedColormap (class in matplotlib.colors), 1158
LinearTriInterpolator (class in matplotlib.tri), 1673
LineCollection (class in matplotlib.collections), 1048
lineStyles (matplotlib.lines.Line2D attribute), 1249
LINETO (matplotlib.path.Path attribute), 1368
List (class in matplotlib.mathtext), 1267
list_fonts() (in module matplotlib.font_manager), 1224
ListedColormap (class in matplotlib.colors), 1159
listFiles() (in module matplotlib.cbook), 962
locally_modified_subplot_params() (matplotlib.gridspec.GridSpec method), 1227
locate() (mpl_toolkits.axes_grid1 axes_grid1 axes_divider Divider method), 1797
LocationEvent (class in matplotlib.backend_bases), 923
Locator (class in matplotlib.ticker), 1642
locator_params() (in module matplotlib.pyplot), 1507
locator_params() (matplotlib.axes.Axes method), 744
locator_params() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 1854
Locked (class in matplotlib.widgets), 1693
Locked (class in matplotlib.mlab), 1693
Locked (class in matplotlib.mlab), 1693
Locked.TimeoutError, 955
LOCF (matplotlib.mlab.Locked attribute), 955
locs (matplotlib.ticker.Formatter attribute), 1238
Log10Transform (class in matplotlib.scale), 1607
log2() (in module matplotlib.mlab), 1298
Log2Transform (class in matplotlib.scale), 1607
LogFormatter (class in matplotlib.ticker), 1640
LogFormatterExponent (class in matplotlib.ticker), 1641
LogFormatterMathText (class in matplotlib.ticker), 1641
LogisticTransform (class in matplotlib.scale), 1610
LogitScale (class in matplotlib.scale), 1610
LogitTransform (class in matplotlib.scale), 1611
LogLocator (class in matplotlib.ticker), 1644
loglog() (in module matplotlib.pyplot), 1507
loglog() (matplotlib.axes.Axes method), 569
LogNorm (class in matplotlib.colors), 1159
LogScale (class in matplotlib.scale), 1607
LogScale.InvertedLog10Transform (class in matplotlib.scale), 1607
LogScale.InvertedLog2Transform (class in matplotlib.scale), 1608
LogScale.InvertedLogTransform (class in matplotlib.scale), 1608
LogScale.InvertedNaturalLogTransform (class in matplotlib.scale), 1608
LogScale.Log10Transform (class in matplotlib.scale), 1608
LogScale.Log2Transform (class in matplotlib.scale), 1608
LogScale.LogTransform (class in matplotlib.scale), 1608
LogScale.LogTransformBase (class in matplotlib.scale), 1609
LogScale.NaturalLogTransform (class in matplotlib.scale), 1609
logspace() (in module matplotlib.mlab), 1298
LogTransform (class in matplotlib.scale), 1609
LogTransformBase (class in matplotlib.scale), 1609
longest_contiguous_ones() (in module matplotlib.mlab), 1298
longest_ones() (in module matplotlib.mlab), 1298
m
magma() (in module matplotlib.pyplot), 1509
magnitude_spectrum() (in module matplotlib.mlab), 1298
magnitude_spectrum() (in module matplotlib.mlab), 1298
magnitude_spectrum() (matplotlib.axes.Axes method), 621
main() (matplotlib.mathtext.Parser method), 1272
mainloop() (matplotlib.backend_bases.ShowBase method), 931
make_axes() (in module matplotlib.colorbar), 1147
make_axes_gridspec() (in module matplotlib.colorbar), 1148
make_compound_path() (matplotlib.path.Path class method), 1371
make_compound_path_from_polys() (matplotlib.path.Path class method), 1372
make_image() (matplotlib.image.BboxImage method), 1232
make_image() (matplotlib.image.FigureImage method), 1232
make_image() (matplotlib.image.NonUniformImage method), 1232
make_image() (matplotlib.image.PcolorImage method), 1233
make_rcparams_key() (matplotlib.font_manager.TempCache method), 1233
makeMappingArray() (in module matplotlib.colors), 1162
margins() (in module matplotlib.pyplot), 1513
margins() (matplotlib.axes.Axes method), 751
margins() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 1855
mark_inset() (in module mpl_toolkits.axes_grid1.inset_locator), 1819
markerObject() (matplotlib.backends.backend_pdf.PdfFile method), 949
markers (matplotlib.lines.Line2D attribute), 1249
markers (matplotlib.markers.MarkerStyle attribute), 1257
MarkerStyle (class in matplotlib.markers), 1256
math() (matplotlib.mathtext.Parser method), 1272
math_string() (matplotlib.mathtext.Parser method), 1273
math_to_image() (in module matplotlib.mathtext), 1277
MathtextBackend (class in matplotlib.mathtext), 1268
MathtextBackendAgg (class in matplotlib.mathtext), 1269
MathtextBackendBitmap (class in matplotlib.mathtext), 1269
MathtextBackendCairo (class in matplotlib.mathtext), 1270
MathtextBackendPath (class in matplotlib.mathtext), 1270
MathtextBackendPdf (class in matplotlib.mathtext), 1270
MathtextBackendPs (class in matplotlib.mathtext), 1270
MathtextBackendSvg (class in matplotlib.mathtext), 1270
<table>
<thead>
<tr>
<th>Function/Method</th>
<th>Module</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>mkdirs()</td>
<td>matplotlib.cbook</td>
<td>963</td>
</tr>
<tr>
<td>mod()</td>
<td>mpl_toolkits.mplot3d.proj3d</td>
<td>1876</td>
</tr>
<tr>
<td>MonthLocator</td>
<td>matplotlib.dates</td>
<td>1173</td>
</tr>
<tr>
<td>motion_notify_event()</td>
<td>matplotlib.backend_bases.FigureCanvasBase</td>
<td>916</td>
</tr>
<tr>
<td>mouse_init()</td>
<td>mpl_toolkits.mplot3d.axes3d.Axes3D</td>
<td>1855</td>
</tr>
<tr>
<td>mouse_move()</td>
<td>matplotlib.backend_bases.NavigationToolbar2</td>
<td>926</td>
</tr>
<tr>
<td>MouseEvent</td>
<td>matplotlib.backend_bases</td>
<td>924</td>
</tr>
<tr>
<td>mouseover</td>
<td>matplotlib.artist.Artist</td>
<td>538</td>
</tr>
<tr>
<td>mouseover</td>
<td>matplotlib.axes.Axes</td>
<td>773</td>
</tr>
<tr>
<td>mouseover</td>
<td>matplotlib.axis.Axis</td>
<td>883</td>
</tr>
<tr>
<td>mouseover</td>
<td>matplotlib.axis.Tick</td>
<td>844</td>
</tr>
<tr>
<td>mouseover</td>
<td>matplotlib.axis.XAxis</td>
<td>895</td>
</tr>
<tr>
<td>mouseover</td>
<td>matplotlib.axis.XTick</td>
<td>855</td>
</tr>
<tr>
<td>mouseover</td>
<td>matplotlib.axis.YAxis</td>
<td>906</td>
</tr>
<tr>
<td>mouseover</td>
<td>matplotlib.axis.YTick</td>
<td>866</td>
</tr>
<tr>
<td>mouseover</td>
<td>matplotlib.collections.AsteriskPolygonCollection</td>
<td>980</td>
</tr>
<tr>
<td>mouseover</td>
<td>matplotlib.collections.BrokenBarHCollection</td>
<td>992</td>
</tr>
<tr>
<td>mouseover</td>
<td>matplotlib.collections.CircleCollection</td>
<td>1004</td>
</tr>
<tr>
<td>mouseover</td>
<td>matplotlib.collections.Collection</td>
<td>1017</td>
</tr>
<tr>
<td>mouseover</td>
<td>matplotlib.collections.EllipseCollection</td>
<td>1028</td>
</tr>
<tr>
<td>mouseover</td>
<td>matplotlib.collections.EventCollection</td>
<td>1041</td>
</tr>
<tr>
<td>mouseover</td>
<td>matplotlib.collections.LineCollection</td>
<td>1054</td>
</tr>
<tr>
<td>mouseover</td>
<td>matplotlib.collections.PatchCollection</td>
<td>1065</td>
</tr>
<tr>
<td>mouseover</td>
<td>matplotlib.collections.PathCollection</td>
<td>1077</td>
</tr>
<tr>
<td>mouseover</td>
<td>matplotlib.collections.PolyCollection</td>
<td>1089</td>
</tr>
<tr>
<td>mouseover</td>
<td>matplotlib.collections.QuadMesh</td>
<td>1101</td>
</tr>
<tr>
<td>mouseover</td>
<td>matplotlib.collections.RegularPolyCollection</td>
<td>1113</td>
</tr>
<tr>
<td>mouseover</td>
<td>matplotlib.collections.StarPolygonCollection</td>
<td>1126</td>
</tr>
<tr>
<td>movav()</td>
<td>matplotlib.mlab</td>
<td>1299</td>
</tr>
<tr>
<td>MOVE</td>
<td>matplotlib.backend_tools.Cursors</td>
<td>937</td>
</tr>
<tr>
<td>move_from_center()</td>
<td>mpl_toolkits.mplot3d.axis3d</td>
<td>1871</td>
</tr>
<tr>
<td>MOVETO</td>
<td>matplotlib.path.Path</td>
<td>1368</td>
</tr>
<tr>
<td>MovieWriter</td>
<td>matplotlib.animation</td>
<td>527</td>
</tr>
<tr>
<td>MovieWriterRegistry</td>
<td>matplotlib.animation</td>
<td>526</td>
</tr>
<tr>
<td>mpl_connect()</td>
<td>matplotlib.backend_bases.FigureCanvasBase</td>
<td>916</td>
</tr>
<tr>
<td>mpl_disconnect()</td>
<td>matplotlib.backend_bases.FigureCanvasBase</td>
<td>917</td>
</tr>
<tr>
<td>mpl_toolkits.axes_grid.axes_size()</td>
<td>mpl_toolkits.axes_grid1.anchored_artists</td>
<td>1795</td>
</tr>
<tr>
<td>mpl_toolkits.axes_grid1.inset_locator()</td>
<td>mpl_toolkits.mplot3d.art3d</td>
<td>1871</td>
</tr>
<tr>
<td>mpl_toolkits.mplot3d.axes3d</td>
<td>mpl_toolkits.mplot3d.axis3d</td>
<td>1848</td>
</tr>
<tr>
<td>mplToolkits</td>
<td>mpl_toolkits.mplot3d.axes3d</td>
<td>1870</td>
</tr>
<tr>
<td>mpl_toolkits.mplot3d.proj3d</td>
<td>mpl_toolkits.mplot3d</td>
<td>1876</td>
</tr>
<tr>
<td>MPLBACKEND</td>
<td>mpl_toolkits.mplot3d</td>
<td>325</td>
</tr>
<tr>
<td>MPLCONFIGDIR</td>
<td>mpl_toolkits.mplot3d</td>
<td>1719</td>
</tr>
<tr>
<td>MPLCONFIGDIR</td>
<td>mpl_toolkits.mplot3d</td>
<td>1720</td>
</tr>
<tr>
<td>MPLCONFIGDIR</td>
<td>mpl_toolkits.mplot3d</td>
<td>1909</td>
</tr>
<tr>
<td>mplDeprecation</td>
<td>matplotlib.backends.backend_pdf</td>
<td>963</td>
</tr>
<tr>
<td>MultiCursor</td>
<td>class in matplotlib.widgets</td>
<td>1693</td>
</tr>
<tr>
<td>MultipleLocator</td>
<td>class in matplotlib.tick</td>
<td>1644</td>
</tr>
<tr>
<td>mutated()</td>
<td>matplotlib.transforms.Bbox method</td>
<td>1655</td>
</tr>
<tr>
<td>mutatedx()</td>
<td>matplotlib.transforms.Bbox method</td>
<td>1655</td>
</tr>
<tr>
<td>mutatedy()</td>
<td>matplotlib.transforms.Bbox method</td>
<td>1655</td>
</tr>
<tr>
<td>mx2num()</td>
<td>matplotlib.dates</td>
<td>1169</td>
</tr>
<tr>
<td>N</td>
<td>n_rasterize</td>
<td>1146</td>
</tr>
<tr>
<td>Name</td>
<td>class in matplotlib.backends.backend_pdf</td>
<td>949</td>
</tr>
<tr>
<td>name</td>
<td>matplotlib.axes.Axes</td>
<td>786</td>
</tr>
<tr>
<td>name</td>
<td>matplotlib.backend_tools.ToolBase</td>
<td>939</td>
</tr>
<tr>
<td>name</td>
<td>matplotlib.scale.LinearScale</td>
<td>1607</td>
</tr>
</tbody>
</table>
OffsetFrom (class in matplotlib.text), 1624
OffsetImage (class in matplotlib.offsetbox), 1320
OFFSETTEXTPAD (matplotlib.axis.Axis attribute), 806
OFFSETTEXTPAD (matplotlib.axis.XAxis attribute), 824
OFFSETTEXTPAD (matplotlib.axis.YAxis attribute), 815
on_changed() (matplotlib.widgets.Slider method), 1698
on_clicked() (matplotlib.widgets.Button method), 1688
on_clicked() (matplotlib.widgets.CheckButtons method), 1689
on_clicked() (matplotlib.widgets.RadioButton method), 1695
on_mappable_changed() (matplotlib.colorbar.Colorbar method), 1145
on_motion() (matplotlib.offsetbox.DraggableBase method), 1317
on_motion_blit() (matplotlib.offsetbox.DraggableBase method), 1317
on_pick() (matplotlib.offsetbox.DraggableBase method), 1317
on_release() (matplotlib.offsetbox.DraggableBase method), 1317
ontrue() (in module matplotlib.cbook), 964
onHilite() (matplotlib.backend_bases.FigureCanvasBase method), 917
onmove() (matplotlib.widgets.Cursor method), 1690
onmove() (matplotlib.widgets.Lasso method), 1692
onmove() (matplotlib.widgets.MultiCursor method), 1694
onpick() (matplotlib.lines.VertexSelector method), 1253
onpress() (matplotlib.widgets.LassoSelector method), 1693
onrelease() (matplotlib.widgets.Lasso method), 1692
onrelease() (matplotlib.widgets.LassoSelector method), 1693
onRemove() (matplotlib.backend_bases.FigureCanvasBase method), 917
open_group() (matplotlib.backend_bases.RendererBase method), 930
Operator (class in matplotlib.backends.backend_pdf), 949
operatorname() (matplotlib.mathtext.Parser method), 1273
option_image_nocomposite() (matplotlib.backend_bases.RendererBase method), 930
option_scale_image() (matplotlib.backend_bases.RendererBase method), 930
ord() (in module matplotlib.dviread), 1184
ord() (in module matplotlib.type1font), 1684
orientation (matplotlib.patches.RegularPolygon attribute), 1361
OSXInstalledFonts() (in module matplotlib.font_manager), 1223
output_dims (matplotlib.scale.InvertedLogTransformBase attribute), 1606
output_dims (matplotlib.scale.InvertedSymmetricalLogTransform attribute), 1606
output_dims (matplotlib.scale.LogisticTransform attribute), 1610
output_dims (matplotlib.scale.LogitTransform attribute), 1611
output_dims (matplotlib.scale.LogScale.LogTransformBase attribute), 1609
output_dims (matplotlib.scale.LogScale.LogTransformBase attribute), 1610
output_dims (matplotlib.scale.SymmetricalLogScale.InvertedSymmetricalLogTransform attribute), 1612
output_dims (matplotlib.scale.SymmetricalLogScale.SymmetricalLogTransform attribute), 1613
output_dims (matplotlib.scale.SymmetricalLogTransform attribute), 1613
over() (in module matplotlib.pyplot), 1514
overlaps() (matplotlib.transforms.BboxBase method), 1653
overline() (matplotlib.mathtext.Parser method), 1273
p0 (matplotlib.transforms.BboxBase attribute), 1653
method), 1089
pchanged() (matplotlib.collections.QuadMesh method), 1101
pchanged() (matplotlib.collections.RegularPolyCollection method), 1113
pchanged() (matplotlib.collections.StarPolygonCollection method), 1126
pchanged() (matplotlib.collections.TriMesh method), 1137
pchanged() (matplotlib.container.Container method), 1165
pcolor() (in module matplotlib.pyplot), 1515
pcolor() (matplotlib.axes.Axes method), 693
pcolorfast() (matplotlib.axes.Axes method), 696
PcolorImage (class in matplotlib.image), 1233
pcolormesh() (in module matplotlib.pyplot), 1518
pcolormesh() (matplotlib.axes.Axes method), 698
PDF, 3017
PdfFile (class in matplotlib.backends.backend_pdf), 949
PdfPages (class in matplotlib.backends.backend_pdf), 949
dfRepr() (in module matplotlib.backends.backend_pdf), 951
persp_transformation() (in module mpl_toolkits.mplot3d, 1876
phase_spectrum() (in module matplotlib.mlab), 1300
phase_spectrum() (in module matplotlib.pyplot), 1520
phase_spectrum() (matplotlib.axes.Axes method), 624
pick() (matplotlib.artist.Artist method), 538
pick() (matplotlib.axes.Axes method), 773
pick() (matplotlib.axis.Axis method), 884
pick() (matplotlib.axis.Tick method), 844
pick() (matplotlib.axis.XAxis method), 895
pick() (matplotlib.axis.XTick method), 855
pick() (matplotlib.axis.YAxis method), 906
pick() (matplotlib.axis.YTick method), 867
pick() (matplotlib.backend_bases.FigureCanvasBase method), 917
pick() (matplotlib.collections.AsteriskPolygonCollection method), 981
pick() (matplotlib.collections.BrokenBarHCollection method), 1005
pick() (matplotlib.collections.CircleCollection method), 1028
pick() (matplotlib.collections.EllipseCollection method), 1041
pick() (matplotlib.collections.EventCollection method), 1054
pick() (matplotlib.collections.LineCollection method), 1065
pick() (matplotlib.collections.PatchCollection method), 1077
pick() (matplotlib.collections.PolyCollection method), 1101
pick() (matplotlib.collections.QuadMesh method), 1114
pick() (matplotlib.collections.RegularPolyCollection method), 1126
pick() (matplotlib.collections.StarPolygonCollection method), 1137
pick() (matplotlib.collections.TriMesh method), 1157
pick_event() (matplotlib.backend_bases.FigureCanvasBase method), 917
pickable() (matplotlib.artist.Artist method), 538
pickable() (matplotlib.axes.Axes method), 773
pickable() (matplotlib.axis.Axis method), 884
pickable() (matplotlib.axis.Tick method), 844
pickable() (matplotlib.axis.XAxis method), 895
pickable() (matplotlib.axis.XTick method), 855
pickable() (matplotlib.axis.YAxis method), 906
pickable() (matplotlib.axis.YTick method), 867
pickable() (matplotlib.collections.AsteriskPolygonCollection method), 981
pickable() (matplotlib.collections.BrokenBarHCollection method), 993
pickable() (matplotlib.collections.CircleCollection method), 1005
pickable() (matplotlib.collections.Collection method), 1017
pickable() (matplotlib.collections.EllipseCollection method), 1028
pickable() (matplotlib.collections.EventCollection method), 1041
pickable() (matplotlib.collections.LineCollection method), 1054
pickable() (matplotlib.collections.PatchCollection method), 1066
pickable() (matplotlib.collections.PathCollection method), 1077
pickable() (matplotlib.collections.PolyCollection method), 1089
pickable() (matplotlib.collections.QuadMesh method), 1101
pickable() (matplotlib.collections.RegularPolyCollection method), 1114
pickable() (matplotlib.collections.StarPolygonCollection method), 1126
pickable() (matplotlib.collections.TriMesh method), 1113
PickEvent (class in matplotlib.backend_bases), 1138
pickle_dump() (in module matplotlib.font_manager), 1224
pickle_load() (in module matplotlib.font_manager), 1224
pie() (in module matplotlib.pyplot), 1523
pie() (matplotlib.axes.Axes method), 592
pieces() (in module matplotlib.cbook), 964
pil_to_array() (in module matplotlib.image), 1235
pink() (in module matplotlib.pyplot), 1525
plasma() (in module matplotlib.pyplot), 1525
plot() (in module matplotlib.pyplot), 1526
plot() (matplotlib.axes.Axes method), 1526
plot() (matplotlib.cbook.MemoryMonitor method), 956
plot() (mpl_toolkits.mplot3d.Axes3D method), 1827
plot() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 1855
plot3D() (mpl_toolkits.mplot3d.Axes3D method), 1856
plot_date() (in module matplotlib.pyplot), 1529
plot_date() (matplotlib.axes.Axes method), 566
plot_day_summary2_ochl() (in module matplotlib.pyplot), 1212
plot_day_summary2_ohlc() (in module matplotlib.pyplot), 1213
plot_day_summary_ochl() (in module matplotlib.pyplot), 1213
plot_day_summary_ohlc() (in module matplotlib.pyplot), 1213
plot_surface() (mpl_toolkits.mplot3d.Axes3D method), 1830
plot_surface() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 1856
plot_trisurf() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 1856
plot_wireframe() (mpl_toolkits.mplot3d.Axes3D method), 1829
plot_wireframe() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 1858
plotfile() (in module matplotlib.pyplot), 1531
plotting() (in module matplotlib.pyplot), 425
PNG, 3017
POINTER (matplotlib.backend_tools.Cursors attribute), 937
points_to_pixels() (matplotlib.backend_bases.RendererBase method), 930
points_to_pixels() (matplotlib.pathffects.PathEffectRenderer method), 1376
polar() (in module matplotlib.pyplot), 1532
PolarAffine (class in matplotlib.projections.polar), 1380
PolarAxes (class in matplotlib.projections.polar), 1380
PolarAxes.InvertedPolarTransform (class in matplotlib.projections.polar), 1381
PolarAxes.PolarAffine (class in matplotlib.projections.polar), 1381
PolarAxes.PolarTransform (class in matplotlib.projections.polar), 1381
PolarAxes.RadialLocator (class in matplotlib.projections.polar), 1382
PolarAxes.ThetaFormatter (class in matplotlib.projections.polar), 1382
PolarAxes.Transform (class in matplotlib.projections.polar), 1382
Poly3DCollection (class in mpl_toolkits.mplot3d.abc3d), 1873
poly_below() (in module matplotlib.mlab), 1301
poly_between() (in module matplotlib.mlab), 1301
poly_collection_2d_to_3d() (in module mpl_toolkits.mplot3d.art3d), 1875
PolyCollection (class in matplotlib.collections), 1084
Polygone (class in matplotlib.patches), 1356
pop_state() (matplotlib.mathtext.Parser method), 1273
popall() (in module matplotlib.cbook), 964
PowerNorm (class in matplotlib.colors), 1161
pprint_val() (matplotlib.tick.LogFormatter

Index 3093
Index 3095
remove_callback() (matplotlib.collections.BrokenBarHCollection method), 993
remove_callback() (matplotlib.collections.CircleCollection method), 1005
remove_callback() (matplotlib.collections.Collection method), 1017
remove_callback() (matplotlib.collections.EllipseCollection method), 1029
remove_callback() (matplotlib.collections.EventCollection method), 1042
remove_callback() (matplotlib.collections.LineCollection method), 1054
remove_callback() (matplotlib.collections.PatchCollection method), 1066
remove_callback() (matplotlib.collections.PathCollection method), 1078
remove_callback() (matplotlib.collections.PolyCollection method), 1090
remove_callback() (matplotlib.collections.QuadMesh method), 1102
remove_callback() (matplotlib.collections.RegularPolygonCollection method), 1114
remove_callback() (matplotlib.collections.StarPolygonCollection method), 1126
remove_callback() (matplotlib.collections.TriMesh method), 1138
remove_callback() (matplotlib.container.Container method), 1165
remove_rubberband() (matplotlib.backend_bases.NavigationToolbar2 method), 926
remove_rubberband() (matplotlib.backend_tools.RubberbandBase method), 938
remove_tool() (matplotlib.backend_managers.ToolManager method), 935
remove_toolitem() (matplotlib.backend_bases.ToolContainerBase method), 933
render() (matplotlib.mathtext.Accent method), 1261
render() (matplotlib.mathtext.Box method), 1262
render() (matplotlib.mathtext.Char method), 1262
render() (matplotlib.mathtext.Node method), 1271
render() (matplotlib.mathtext.Rule method), 1274
render_glyph() (matplotlib.mathtext.Fonts method), 1265
render_glyph() (matplotlib.mathtext.MathtextBackend method), 1269
render_glyph() (matplotlib.mathtext.MathtextBackendAgg method), 1269
render_glyph() (matplotlib.mathtext.MathtextBackendCairo method), 1270
render_glyph() (matplotlib.mathtext.MathtextBackendPath method), 1270
render_glyph() (matplotlib.mathtext.MathtextBackendPdf method), 1270
render_glyph() (matplotlib.mathtext.MathtextBackendPs method), 1270
render_glyph() (matplotlib.mathtext.MathtextBackendSvg method), 1271
render_rect_filled() (matplotlib.mathtext.Fonts method), 1266
render_rect_filled() (matplotlib.mathtext.MathtextBackend method), 1269
render_rect_filled() (matplotlib.mathtext.MathtextBackendAgg method), 1269
render_rect_filled() (matplotlib.mathtext.MathtextBackendCairo method), 1270
render_rect_filled() (matplotlib.mathtext.MathtextBackendPath method), 1270
render_rect_filled() (matplotlib.mathtext.MathtextBackendPdf method), 1270
<table>
<thead>
<tr>
<th>function or method</th>
<th>module or class</th>
<th>line number</th>
</tr>
</thead>
<tbody>
<tr>
<td>rotate_axes()</td>
<td>(in module</td>
<td>1270</td>
</tr>
<tr>
<td></td>
<td>mpl_toolkits.mpl_toolkits.mplot3d.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>art3d)</td>
<td></td>
</tr>
<tr>
<td>rotate_deg()</td>
<td>(matplotlib.transforms.Affine2D method)</td>
<td>1663</td>
</tr>
<tr>
<td>rotate_deg_around()</td>
<td>(matplotlib.transforms.Affine2D method)</td>
<td>1663</td>
</tr>
<tr>
<td>rotated()</td>
<td>(matplotlib.transforms.BboxBase method)</td>
<td>1653</td>
</tr>
<tr>
<td>rrule</td>
<td>(class in matplotlib.dates)</td>
<td>1175</td>
</tr>
<tr>
<td>RRuleLocator</td>
<td>(class in matplotlib.dates)</td>
<td>1171</td>
</tr>
<tr>
<td>RubberbandBase</td>
<td>(class in matplotlib.backend_tools)</td>
<td>937</td>
</tr>
<tr>
<td>save()</td>
<td>(matplotlib.animation.Animation method)</td>
<td>494</td>
</tr>
<tr>
<td>save()</td>
<td>(matplotlib.animation.MovieWriter method)</td>
<td>524</td>
</tr>
<tr>
<td>savefig()</td>
<td>(in module matplotlib.pyplot)</td>
<td>1543</td>
</tr>
<tr>
<td>savefig()</td>
<td>(matplotlib.backends.backend_pdf.PdfPages method)</td>
<td>950</td>
</tr>
<tr>
<td>SaveFigureBase</td>
<td>(class in matplotlib.backend_tools)</td>
<td>938</td>
</tr>
<tr>
<td>restrict_dict()</td>
<td>(in module matplotlib.cbook)</td>
<td>921</td>
</tr>
<tr>
<td>revcmap()</td>
<td>(in module matplotlib.cm)</td>
<td>973</td>
</tr>
<tr>
<td>reverse_dict()</td>
<td>(in module matplotlib.cbook)</td>
<td>966</td>
</tr>
<tr>
<td>rgb_to_hsv()</td>
<td>(in module matplotlib.colors)</td>
<td>1162</td>
</tr>
<tr>
<td>rgrids()</td>
<td>(in module matplotlib.pyplot)</td>
<td>1542</td>
</tr>
<tr>
<td>RingBuffer</td>
<td>(class in matplotlib.cbook)</td>
<td>956</td>
</tr>
<tr>
<td>rk4()</td>
<td>(in module matplotlib.mlab)</td>
<td>1306</td>
</tr>
<tr>
<td>rms_flat()</td>
<td>(in module matplotlib.mlab)</td>
<td>1307</td>
</tr>
<tr>
<td>rot_x()</td>
<td>(in module mpl_toolkits.mplot3d.proj3d)</td>
<td>1876</td>
</tr>
<tr>
<td>rotate()</td>
<td>(matplotlib.transforms.Affine2D method)</td>
<td>1663</td>
</tr>
<tr>
<td>rotate_around()</td>
<td>(matplotlib.transforms.Affine2D method)</td>
<td>1663</td>
</tr>
<tr>
<td>render_rect_filled()</td>
<td>(matplotlib.mathtext.MathtextBackendPs method)</td>
<td>1270</td>
</tr>
<tr>
<td>render_rect_filled()</td>
<td>(matplotlib.mathtext.MathtextBackendSvg method)</td>
<td>1271</td>
</tr>
<tr>
<td>RendererBase</td>
<td>(class in matplotlib.backend_bases)</td>
<td>927</td>
</tr>
<tr>
<td>replace()</td>
<td>(matplotlib.dates.rrule method)</td>
<td>1177</td>
</tr>
<tr>
<td>report()</td>
<td>(matplotlib.cbook.MemoryMonitor method)</td>
<td>956</td>
</tr>
<tr>
<td>report_memory()</td>
<td>(in module matplotlib.cbook)</td>
<td>965</td>
</tr>
<tr>
<td>required_group()</td>
<td>(matplotlib.mathtext.Parser method)</td>
<td>1273</td>
</tr>
<tr>
<td>reserveObject()</td>
<td>(matplotlib.backends.backend_pdf.PdfFile method)</td>
<td>949</td>
</tr>
<tr>
<td>reset()</td>
<td>(matplotlib.widgets.Slider method)</td>
<td>1698</td>
</tr>
<tr>
<td>reset_position()</td>
<td>(matplotlib.axes.Axes method)</td>
<td>1698</td>
</tr>
<tr>
<td>reset_ticks()</td>
<td>(matplotlib.axis.Axis method)</td>
<td>806</td>
</tr>
<tr>
<td>reset_ticks()</td>
<td>(matplotlib.axis.XAxis method)</td>
<td>829</td>
</tr>
<tr>
<td>reset_ticks()</td>
<td>(matplotlib.axis.YAxis method)</td>
<td>820</td>
</tr>
<tr>
<td>resize()</td>
<td>(matplotlib.backend_bases.FigureCanvasBase method)</td>
<td>918</td>
</tr>
<tr>
<td>resize()</td>
<td>(matplotlib.backend_bases.FigureManagerBase method)</td>
<td>919</td>
</tr>
<tr>
<td>resize_event()</td>
<td>(matplotlibbackend_bases.FigureCanvasBase method)</td>
<td>918</td>
</tr>
<tr>
<td>ResizeEvent</td>
<td>(class in matplotlib.backend_bases)</td>
<td>930</td>
</tr>
<tr>
<td>restore()</td>
<td>(matplotlib.backend_bases.GraphicsContextBase method)</td>
<td>921</td>
</tr>
<tr>
<td>save_offset()</td>
<td>(matplotlib.offsetbox.DraggableAnnotation method)</td>
<td>1316</td>
</tr>
<tr>
<td>save_offset()</td>
<td>(matplotlib.offsetbox.DraggableBase method)</td>
<td>1317</td>
</tr>
<tr>
<td>save_offset()</td>
<td>(matplotlib.offsetbox.DraggableOffsetBox method)</td>
<td>1318</td>
</tr>
<tr>
<td>sca()</td>
<td>(in module matplotlib.pyplot)</td>
<td>1544</td>
</tr>
<tr>
<td>sca()</td>
<td>(matplotlib.figure.Figure method)</td>
<td>1201</td>
</tr>
<tr>
<td>ScalarFormatter</td>
<td>(class in matplotlib.ticker)</td>
<td>1639</td>
</tr>
</tbody>
</table>
ScalarMappable (class in matplotlib.cm), 971
scale() (matplotlib.transforms.Affine2D method), 1663
scale_factors (matplotlib.tri.TriAnalyzer attribute), 1681
scale_factory() (in module matplotlib.scale), 1614
ScaleBase (class in matplotlib.scale), 1611
Scaled (class in mpl_toolkits.axes_grid.axes_size), 1795
scaled() (matplotlib.colors.Normalize method), 1161
ScaledTranslation (class in matplotlib.transforms), 1669
scatter() (in module matplotlib.pyplot), 1544
scatter() (matplotlib.axes.Axes method), 564
scatter() (mpl_toolkits.mplot3d.Axes3D method), 1827
scatter() (mpl_toolkits.mplot3d.axes3d.Axes3D.method), 1860
scatter3D() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 1861
sci() (in module matplotlib.pyplot), 1546
score_family() (matplotlib.font_manager.FontManager method), 1220
score_size() (matplotlib.font_manager.FontManager method), 1220
score_stretch() (matplotlib.font_manager.FontManager method), 1220
score_style() (matplotlib.font_manager.FontManager method), 1220
score_variant() (matplotlib.font_manager.FontManager method), 1220
score_weight() (matplotlib.font_manager.FontManager method), 1220
scotts_factor() (matplotlib.mlab.GaussianKDE method), 1283
script_space (matplotlib.mathtext.ComputerModernFontConstants attribute), 1262
script_space (matplotlib.mathtext.FontConstantsBase attribute), 1264
script_space (matplotlib.mathtext.STIXFontConstants attribute), 1274
script_space (matplotlib.mathtext.STIXSansFontConstants attribute), 1274
scroll_event() (matplotlib.backend_bases.FigureCanvasBase method), 918
scroll_zoom() (matplotlib.backend_tools.ZoomPanBase method), 944
SecondLocator (class in matplotlib.dates), 1174
seconds() (in module matplotlib.dates), 1178
segment_hits() (in module matplotlib.lines), 1254
segments_intersect() (in module matplotlib.mlab), 1307
SELECTREGION (matplotlib.backend_tools.Cursors attribute), 937
semilogx() (in module matplotlib.pyplot), 1546
semilogx() (matplotlib.axes.Axes method), 571
semilogy() (in module matplotlib.pyplot), 1548
semilogy() (matplotlib.axes.Axes method), 573
send_message() (matplotlib.backend_tools.ToolCursorPosition method), 940
set() (matplotlib.artist.Artist method), 542
set() (matplotlib.axes.Axes method), 778
set() (matplotlib.axis.Axis method), 885
set() (matplotlib.axis.Tick method), 845
set() (matplotlib.axis.XAxis method), 896
set() (matplotlib.axis.XTick method), 856
set() (matplotlib.axis.YAxis method), 907
set() (matplotlib.axis.YTick method), 867
set() (matplotlib.collections.AsteriskPolygonCollection method), 981
set() (matplotlib.collections.BrokenBarHCollection method), 993
set() (matplotlib.collections.CircleCollection method), 1005
set() (matplotlib.collections.Collection method), 1017
set() (matplotlib.collections.EllipseCollection method), 1029
set() (matplotlib.collections.EventCollection method), 1042
set() (matplotlib.collections.LineCollection method), 1054
set() (matplotlib.collections.PatchCollection method), 1066
set() (matplotlib.collections.PolyCollection method), 1078
set() (matplotlib.collections.QuadMesh method), 1102
set() (matplotlib.collections.RegularPolyCollection method), 1114
set() (matplotlib.collections.StarPolygonCollection method), 1126
set() (matplotlib.collections.TriMesh method), 1138
set() (matplotlib.font_manager.TempCache method), 1224
set() (matplotlib.transforms.Affine2D method), 1663
set() (matplotlib.transforms.Bbox method), 1655
set() (matplotlib.transforms.TransformWrapper method), 1660
set_3d_properties() (mpl_toolkits.mplot3d.art3d.Line3D method), 1871
set_3d_properties() (mpl_toolkits.mplot3d.art3d.Patch3D method), 1872
set_3d_properties() (mpl_toolkits.mplot3d.art3d.Path3D method), 1872
set_3d_properties() (mpl_toolkits.mplot3d.art3d.Path3DCollection method), 1873
set_3d_properties() (mpl_toolkits.mplot3d.art3d.PathPatch3D method), 1873
set_3d_properties() (mpl_toolkits.mplot3d.art3d.Poly3DCollection method), 1873
set_3d_properties() (mpl_toolkits.mplot3d.art3d.Text3D method), 1874
set_aa() (matplotlib.lines.Line2D method), 1249
set_aa() (matplotlib.patches.Patch method), 1353
set_active() (matplotlib.widgets.CheckButtons method), 1689
set_active() (matplotlib.widgets.RadioButton method), 1695
set_active() (matplotlib.widgets.Widget method), 1701
set_adjustable() (matplotlib.axes.Axes method), 755
set_agg_filter() (matplotlib.artist.Artist method), 544
set_agg_filter() (matplotlib.artist.Axes method), 782
set_agg_filter() (matplotlib.axis.Axis method), 885
set_agg_filter() (matplotlib.axis.Tick method), 845
set_agg_filter() (matplotlib.axis.XAxis method), 896
set_agg_filter() (matplotlib.axis.XTick method), 856
set_agg_filter() (matplotlib.axis.YAxis method), 907
set_agg_filter() (matplotlib.axis.YTick method), 868
set_agg_filter() (matplotlib.collections.AsteriskPolygonCollection method), 981
set_agg_filter() (matplotlib.collections.BrokenBarHCollection method), 993
set_agg_filter() (matplotlib.collections.CircleCollection method), 1005
set_agg_filter() (matplotlib.collections.Collection method), 1017
set_agg_filter() (matplotlib.collections.EllipseCollection method), 1029
set_agg_filter() (matplotlib.collections.EventCollection method), 1042
set_agg_filter() (matplotlib.collections.LineCollection method), 1054
set_agg_filter() (matplotlib.collections.PatchCollection method), 1066
set_agg_filter() (matplotlib.collections.PathCollection method), 1078
set_agg_filter() (matplotlib.collections.PolyCollection method), 1090
set_agg_filter() (matplotlib.collections.QuadMesh method), 1102
set_agg_filter() (matplotlib.collections.RegularPolyCollection method), 1114
set_agg_filter() (matplotlib.collections.StarPolygonCollection method), 1126
set_agg_filter() (matplotlib.collections.TriMesh method), 1138
set_alpha() (matplotlib.artist.Artist method), 544
set_alpha() (matplotlib.artist.Axes method), 779
set_alpha() (matplotlib.axis.Axis method), 885
set_alpha() (matplotlib.axis.Tick method), 845
set_alpha() (matplotlib.axis.XAxis method), 896
set_alpha() (matplotlib.axis.XTick method), 856
set_alpha() (matplotlib.axis.YAxis method), 907
set_alpha() (matplotlib.axis.YTick method), 868
set_alpha() (matplotlib.backend_bases.GraphicsContextBase method), 921
set_alpha() (matplotlib.collections.AsteriskPolygonCollection method), 981
set_alpha() (matplotlib.collections.AsteriskPolygonCollection method), 981
set_alpha() (matplotlib.collections.BrokenBarHCollection method), 993
set_alpha() (matplotlib.collections.CircleCollection method), 1005
set_alpha() (matplotlib.collections.Collection method), 1017
set_alpha() (matplotlib.collections.EllipseCollection method), 1029
set_alpha() (matplotlib.collections.EventCollection method), 1042
set_alpha() (matplotlib.collections.LineCollection method), 1055
set_alpha() (matplotlib.collections.PatchCollection method), 1066
set_alpha() (matplotlib.collections.PathCollection method), 1078
set_alpha() (matplotlib.collections.PolyCollection method), 1090
set_alpha() (matplotlib.collections.QuadMesh method), 1102
set_alpha() (matplotlib.collections.RegularPolyCollection method), 1114
set_alpha() (matplotlib.collections.StarPolygonCollection method), 1127
set_alpha() (matplotlib.collections.TriMesh method), 1138
set_alpha() (matplotlib.colorbar.ColorbarBase method), 1147
set_alpha() (mpl_toolkits.mplot3d.art3d.Poly3DCollection method), 1151
set_anchor() (matplotlib.axes.Axes method), 767
set_anchor() (mpl_toolkits.axes_grid.axes_divider.Divider method), 1797
set_animated() (matplotlib.artist.Artist method), 543
set_animated() (matplotlib.axis.Axis method), 780
set_animated() (matplotlib.axis.Tick method), 845
set_animated() (matplotlib.axis.XAxis method), 896
set_animated() (matplotlib.axis.XTick method), 857
set_animated() (matplotlib.axis.YAxis method), 907
set_animated() (matplotlib.axis.YTick method), 868
set_animated() (matplotlib.collections.AsteriskPolygonCollection method), 981
set_animated() (matplotlib.collections.BrokenBarHCollection method), 993
set_antialiased() (matplotlib.backend_bases.GraphicsContextBase method), 1339
set_annotation_clip() (matplotlib.widgets.ToolHandles method), 1700
set_anti aliased() (matplotlib.backends.backend_bases.GraphicsContextBase method), 921
set_anti aliased() (matplotlib.collections.AsteriskPolygonCollection method), 981
set_anti aliased() (matplotlib.collections.BrokenBarHCollection method), 993
set_anti aliased() (mat-
<table>
<thead>
<tr>
<th>Function</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>set_antialiased()</td>
<td>matplotlib.collections.Collection method</td>
</tr>
<tr>
<td>set_antialiased()</td>
<td>matplotlib.collections.CircleCollection method, 1005</td>
</tr>
<tr>
<td>set_antialiased()</td>
<td>matplotlib.collections.EllipseCollection method, 1029</td>
</tr>
<tr>
<td>set_antialiased()</td>
<td>matplotlib.collections.EventCollection method, 1042</td>
</tr>
<tr>
<td>set_antialiased()</td>
<td>matplotlib.collections.LineCollection method, 1055</td>
</tr>
<tr>
<td>set_antialiased()</td>
<td>matplotlib.collections.PatchCollection method, 1066</td>
</tr>
<tr>
<td>set_antialiased()</td>
<td>matplotlib.collections.PathCollection method, 1078</td>
</tr>
<tr>
<td>set_antialiased()</td>
<td>matplotlib.collections.PolyCollection method, 1090</td>
</tr>
<tr>
<td>set_antialiased()</td>
<td>matplotlib.collections.QuadMesh method, 1102</td>
</tr>
<tr>
<td>set_antialiased()</td>
<td>matplotlib.collections.RegularPolyCollection method, 1114</td>
</tr>
<tr>
<td>set_antialiased()</td>
<td>matplotlib.collections.StarPolygonCollection method, 1127</td>
</tr>
<tr>
<td>set_antialiased()</td>
<td>matplotlib.collections.TriMesh method, 1138</td>
</tr>
<tr>
<td>set_array()</td>
<td>matplotlib.cm.ScalarMappable method, 972</td>
</tr>
<tr>
<td>set_array()</td>
<td>matplotlib.collections.AsteriskPolygonCollection method, 981</td>
</tr>
<tr>
<td>set_array()</td>
<td>matplotlib.collections.BrokenBarHCollection method, 993</td>
</tr>
<tr>
<td>set_array()</td>
<td>matplotlib.collections.CircleCollection method, 1005</td>
</tr>
<tr>
<td>set_array()</td>
<td>matplotlib.collections.Collection method, 1018</td>
</tr>
<tr>
<td>set_array()</td>
<td>matplotlib.collections.EventCollection method, 1029</td>
</tr>
<tr>
<td>set_array()</td>
<td>matplotlib.collections.LineCollection method, 1055</td>
</tr>
<tr>
<td>set_array()</td>
<td>matplotlib.collections.PatchCollection method, 1066</td>
</tr>
<tr>
<td>set_array()</td>
<td>matplotlib.collections.PathCollection method, 1078</td>
</tr>
<tr>
<td>set_array()</td>
<td>matplotlib.collections.PolyCollection method, 1102</td>
</tr>
<tr>
<td>set_array()</td>
<td>matplotlib.collections.RegularPolyCollection method, 1114</td>
</tr>
<tr>
<td>set_array()</td>
<td>matplotlib.collections.StarPolygonCollection method, 1127</td>
</tr>
<tr>
<td>set_array()</td>
<td>matplotlib.collections.TriMesh method, 1138</td>
</tr>
<tr>
<td>set_array()</td>
<td>matplotlib.cm.ScalarMappable method, 972</td>
</tr>
<tr>
<td>set_array()</td>
<td>matplotlib.collections.AsteriskPolygonCollection method, 981</td>
</tr>
<tr>
<td>set_array()</td>
<td>matplotlib.collections.BrokenBarHCollection method, 993</td>
</tr>
<tr>
<td>set_array()</td>
<td>matplotlib.collections.CircleCollection method, 1005</td>
</tr>
<tr>
<td>set_array()</td>
<td>matplotlib.collections.Collection method, 1018</td>
</tr>
<tr>
<td>set_array()</td>
<td>matplotlib.collections.EventCollection method, 1029</td>
</tr>
<tr>
<td>set_array()</td>
<td>matplotlib.collections.LineCollection method, 1055</td>
</tr>
<tr>
<td>set_array()</td>
<td>matplotlib.collections.PatchCollection method, 1066</td>
</tr>
<tr>
<td>set_array()</td>
<td>matplotlib.collections.PathCollection method, 1078</td>
</tr>
<tr>
<td>set_array()</td>
<td>matplotlib.collections.PolyCollection method, 1102</td>
</tr>
</tbody>
</table>
set_bad() (matplotlib.colors.Colormap method), 1153
set_bbox() (matplotlib.text.Text method), 1628
set_bbox_to_anchor() (matplotlib.legend.Legend method), 1240
set_bbox_to_anchor() (matplotlib.offsetbox.AnchoredOffsetbox method), 1314
set_bounds() (matplotlib.patches.FancyBboxPatch method), 1349
set_bounds() (matplotlib.patches.Rectangle method), 1359
set_bounds() (matplotlib.patches.Spine method), 1617
set_bounds() (matplotlib.ticker.TickHelper method), 1637
set_boxstyle() (matplotlib.patches.FancyBboxPatch method), 1349
set_c() (matplotlib.lines.Line2D method), 1249
set_canvas() (matplotlib.figure.Figure method), 1201
set_canvas_size() (matplotlib.mathtext.Fonts method), 1266
set_canvas_size() (matplotlib.mathtext.MathtextBackend method), 1269
set_canvas_size() (matplotlib.mathtext.MathtextBackendAgg method), 1269
set_capstyle() (matplotlib.backend_bases.GraphicsContextBase method), 921
set_capstyle() (matplotlib.patches.Patch method), 1353
set_center() (matplotlib.patches.Wedge method), 1363
set_child() (matplotlib.offsetbox.AnchoredOffsetbox method), 1314
set_children() (matplotlib.transforms.TransformNode method), 1651
set_clim() (matplotlib.cm.ScalarMappable method), 972
set_clim() (matplotlib.collections.Collection method), 1006
set_clim() (matplotlib.collections.EllipseCollection method), 1018
set_clim() (matplotlib.collections.EventCollection method), 1029
set_clim() (matplotlib.collections.LineCollection method), 1042
set_clim() (matplotlib.collections.PatchCollection method), 1055
set_clim() (matplotlib.collections.PathCollection method), 1067
set_clim() (matplotlib.collections.PolyCollection method), 1078
set_clim() (matplotlib.collections.QuadMesh method), 1102
set_clim() (matplotlib.collections.RegularPolyCollection method), 1115
set_clim() (matplotlib.collections.StarPolygonCollection method), 1127
set_clim() (matplotlib.collections.TriMesh method), 1139
set_clip_box() (matplotlib.artist.Artist method), 541
set_clip_box() (matplotlib.axes.Axes method), 780
set_clip_box() (matplotlib.axis.Axis method), 885
set_clip_box() (matplotlib.axis.Tick method), 845
set_clip_box() (matplotlib.axis.YAxis method), 897
set_clip_box() (matplotlib.axis.XTick method), 857
set_clip_box() (matplotlib.axis.YTick method), 868
set_clip_box() (matplotlib.collections.AsteriskPolygonCollection method), 982
set_clip_box() (matplotlib.collections.BrokenBarHCollection method), 994
set_clip_box() (matplotlib.collections.CircleCollection method), 1006
set_clip_box() (matplotlib.collections.Collection method), 1018
set_clip_box() (matplotlib.collections.EllipseCollection method), 1030
set_clip_box() (matplotlib.collections.EventCollection method), 1042
set_clip_box() (matplotlib.collections.LineCollection method), 1055
set_clip_box() (matplotlib.collections.PatchCollection method), 1067
set_clip_box() (matplotlib.collections.PathCollection method), 1078
set_clip_box() (matplotlib.collections.PolyCollection method), 1090
set_clip_box() (matplotlib.collections.QuadMesh method), 1103
set_clip_box() (matplotlib.collections.RegularPolyCollection method), 1115
set_clip_box() (matplotlib.collections.StarPolygonCollection method), 1127
set_clip_box() (matplotlib.collections.TriMesh method), 1139
set_clip_box() (matplotlib.collections.EventCollection method), 1043
set_clip_on() (matplotlib.collections.LineCollection method), 1055
set_clip_on() (matplotlib.collections.PatchCollection method), 1067
set_clip_on() (matplotlib.collections.PathCollection method), 1079
set_clip_on() (matplotlib.collections.PolyCollection method), 1091
set_clip_on() (matplotlib.collections.QuadMesh method), 1103
set_clip_on() (matplotlib.collections.RegularPolyCollection method), 1115
set_clip_on() (matplotlib.collections.StarPolygonCollection method), 1127
set_clip_on() (matplotlib.collections.TriMesh method), 1139
set_clip_on() (matplotlib.collections.EventCollection method), 1043

set_clip_path() (matplotlib.artist.Artist method), 541
set_clip_path() (matplotlib.axes.Axes method), 780
set_clip_path() (matplotlib.axis.Axis method), 886
set_clip_path() (matplotlib.axis.Tick method), 846
set_clip_path() (matplotlib.axis.XAxis method), 897
set_clip_path() (matplotlib.axis.XTick method), 857
set_clip_path() (matplotlib.axis.YAxis method), 908
set_clip_path() (matplotlib.axis.YTick method), 869
set_clip_path() (matplotlib.backend_bases.GraphicsContextBase method), 921
set_clip_path() (matplotlib.collections.AsteriskPolygonCollection method), 982
set_clip_path() (matplotlib.collections.BrokenBarHCollection method), 994
set_clip_path() (matplotlib.collections.CircleCollection method), 1006
set_clip_path() (matplotlib.collections.Collection method), 1018
set_clip_path() (matplotlib.collections.EllipseCollection method), 1030
set_clip_path() (matplotlib.collections.EventCollection method), 1043
set_dashrotation() (matplotlib.text.TextWithDash method), 1633
set_data() (matplotlib.image.FigureImage method), 1232
set_data() (matplotlib.image.NonUniformImage method), 1233
set_data() (matplotlib.image.PcolorImage method), 1233
set_data() (matplotlib.lines.Line2D method), 1249
set_data() (matplotlib.offsetbox.OffsetImage method), 1320
set_data() (matplotlib.widgets.ToolHandles method), 1701
set_data_interval() (matplotlib.axis.Axis method), 801
set_data_interval() (matplotlib.axis.XAxis method), 829
set_data_interval() (matplotlib.axis.YAxis method), 820
set_data_interval() (matplotlib.dates.MicrosecondLocator method), 1174
set_data_interval() (matplotlib.ticker.TickHelper method), 1637
set_default_handler_map() (matplotlib.legend.Legend class method), 1241
set_default_intervals() (matplotlib.axis.Axis method), 806
set_default_intervals() (matplotlib.axis.XAxis method), 830
set_default_intervals() (matplotlib.axis.YAxis method), 820
set_default_locators_and_formatters() (matplotlib.scale.LinearScale method), 1607
set_default_locators_and_formatters() (matplotlib.scale.LogitScale method), 1611
set_default_locators_and_formatters() (matplotlib.scale.LogScale method), 1609
set_default_locators_and_formatters() (matplotlib.scale.ScaleBase method), 1612
set_default_locators_and_formatters() (matplotlib.scale.SymmetricalLogScale method), 1613
set_default_weight() (matplotlib.font_manager.FontManager method), 1221
**Index**

```
set_edgecolor() (mpl_toolkits.mplot3d.art3d.Poly3DCollection method), 1874
set_edgecolors() (matplotlib.collections.AsteriskPolygonCollection method), 983
set_edgecolors() (matplotlib.collections.BrokenBarHCollection method), 995
set_edgecolors() (matplotlib.collections.CircleCollection method), 1007
set_edgecolors() (matplotlib.collections.EventCollection method), 1044
set_edgecolors() (matplotlib.collections.LineCollection method), 1056
set_edgecolors() (matplotlib.collections.PatchCollection method), 1068
set_edgecolors() (matplotlib.collections.RegularPolyCollection method), 1116
set_edgecolors() (matplotlib.collections.StarPolygonCollection method), 1128
set_edgecolors() (matplotlib.collections.TriMesh method), 1140
set_edgecolors() (mpl_toolkits.mplot3d.art3d.Poly3DCollection method), 1874
set_extent() (matplotlib.image.AxesImage method), 1231
set_facecolor() (matplotlib.axes.Axes method), 736
set_facecolor() (matplotlib.collections.AsteriskPolygonCollection method), 983
set_facecolor() (matplotlib.collections.BrokenBarHCollection method), 995
set_facecolor() (matplotlib.collections.CircleCollection method), 1007
set_facecolor() (matplotlib.collections.EventCollection method), 1044
set_facecolor() (matplotlib.collections.LineCollection method), 1056
set_facecolor() (matplotlib.collections.PatchCollection method), 1068
set_facecolor() (matplotlib.collections.PathCollection method), 1080
set_facecolor() (matplotlib.collections.PolyCollection method), 1092
set_facecolor() (matplotlib.collections.QuadMesh method), 1104
set_facecolor() (matplotlib.collections.RegularPolyCollection method), 1116
set_facecolor() (matplotlib.collections.StarPolygonCollection method), 1128
set_facecolor() (matplotlib.collections.TriMesh method), 1140
set_facecolor() (matplotlib.figure.Figure method), 1201
set_facecolor() (matplotlib.patches.Patch method), 1354
set_facecolors() (matplotlib.collections.AsteriskPolygonCollection method), 983
```
set_filternorm() (matplotlib.image.NonUniformImage method), 1233
set_filterrad() (matplotlib.image.NonUniformImage method), 1233
set_font_properties() (matplotlib.text.Text method), 1629
set_fontconfig_pattern() (matplotlib.font_manager.FontProperties method), 1223
set_fontname() (matplotlib.text.Text method), 1629
set_fontproperties() (matplotlib.text.Text method), 1629
set_fontsize() (matplotlib.offsetbox.AnnotationBbox method), 1315
set_fontsize() (matplotlib.text.Text method), 1629
set_fontstretch() (matplotlib.text.Text method), 1629
set_fontstyle() (matplotlib.text.Text method), 1629
set_fontweight() (matplotlib.text.Text method), 1629
set_foreground() (matplotlib.backend_bases.GraphicsContextBase method), 922
set_frame_on() (matplotlib.axes.Axes method), 733
set_frame_on() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 1862
set_frame_on() (matplotlib.figure.Figure method), 1202
set_frameon() (matplotlib.legend.Legend method), 1241
set_frameon() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 1862
set_frameon() (matplotlib.figure.Figure method), 1202
set_gamma() (matplotlib.colors.LinearSegmentedColormap method), 1159
set_gid() (matplotlib.artist.Artist method), 550
set_gid() (matplotlib.axes.Axes method), 780
set_gid() (matplotlib.axis.Axis method), 886
set_gid() (matplotlib.axis.Tick method), 847
set_gid() (matplotlib.axis.XAxis method), 897
set_gid() (matplotlib.axis.XTick method), 858
set_gid() (matplotlib.axis.YAxis method), 909
set_gid() (matplotlib.axis.YTick method), 869
set_gid() (matplotlib.backend_bases.GraphicsContextBase method), 922
set_gid() (matplotlib.collections.AsteriskPolygonCollection method), 983
set_gid() (matplotlib.collections.BrokenBarHCollection method), 995
set_gid() (matplotlib.collections.CircleCollection method), 1007
set_gid() (matplotlib.collections.Collection method), 1019
set_gid() (matplotlib.collections.EllipseCollection method), 1019
set_gid() (matplotlib.collections.EventCollection method), 1031
set_gid() (matplotlib.collections.LineCollection method), 1044
set_gid() (matplotlib.collections.LineCollection method), 1057
set_gid() (matplotlib.collections.LineCollection method), 1057
set_gid() (matplotlib.collections.PatchCollection method), 1068
set_gid() (matplotlib.collections.PathCollection method), 1080
set_gid() (matplotlib.collections.PolyCollection method), 1092
set_gid() (matplotlib.collections.QuadMesh method), 1104
set_gid() (matplotlib.collections.RegularPolyCollection method), 1116
set_gid() (matplotlib.collections.StarPolygonCollection method), 1129
set_gid() (matplotlib.collections.TriMesh method), 1140
set_graylevel() (matplotlib.backend_bases.GraphicsContextBase method), 922
set_ha() (matplotlib.text.Text method), 1629
set_hatch() (matplotlib.backend_bases.GraphicsContextBase method), 922
set_hatch() (matplotlib.collections.AsteriskPolygonCollection method), 983
set_hatch() (matplotlib.collections.BrokenBarHCollection method), 995
set_hatch() (matplotlib.collections.CircleCollection method), 1007
set_hatch() (matplotlib.collections.Collection method), 1019
set_hatch() (matplotlib.collections.EllipseCollection method), 1031
set_hatch() (matplotlib.collections.EventCollection method), 1044
set_hatch() (matplotlib.collections.LineCollection method), 1057
set_hatch() (matplotlib.collections.PatchCollection method), 1068
set_hatch() (matplotlib.collections.PathCollection method), 1080
set_hatch() (matplotlib.collections.PolyCollection method), 1092
set_hatch() (matplotlib.collections.QuadMesh method), 1104
set_hatch() (matplotlib.collections.RegularPolygon method), 1116
set_hatch() (matplotlib.collections.StarPolygonCollection method), 1129
set_hatch() (matplotlib.collections.TriMesh method), 1140
set_hatch() (matplotlib.patches.Patch method), 1354
set_hatch_color() (matplotlib.backend_bases.GraphicsContextBase method), 922
set_height() (matplotlib.offsetbox.OffsetsBox method), 1320
set_height() (matplotlib.patches.FancyBboxPatch method), 1350
set_height() (matplotlib.patches.Rectangle method), 1359
set_height_ratios() (matplotlib.gridspec.GridSpecBase method), 1228
set_history_buttons() (matplotlib.backend_bases.NavigationToolbar2 method), 926
set_horizontal() (mpl_toolkits.axes_grid.axes_divider.Divider method), 1798
set_interpolation() (matplotlib.image.NonUniformImage method), 1233
set_joinstyle() (matplotlib.backend_bases.GraphicsContextBase method), 922
set_joinstyle() (matplotlib.patches.Patch method), 1354
set_label() (matplotlib.artist.Artist method), 550
set_label() (matplotlib.axes.Axes method), 781
set_label() (matplotlib.axis.Axes method), 886
set_label() (matplotlib.axis.Tick method), 847
set_label() (matplotlib.axis.XAxis method), 898
set_label() (matplotlib.axis.XTick method), 858
set_label() (matplotlib.axis.YAxis method), 909
set_label() (matplotlib.axis.YTick method), 869
set_label() (matplotlib.collections.AsteriskPolygonCollection method), 984
set_label() (matplotlib.collections.BrokenBarHCollection method), 996
set_label() (matplotlib.collections.CircleCollection method), 1008
set_label() (matplotlib.collections.Collection method), 1020
set_label() (matplotlib.collections.EllipseCollection method), 1032
set_label() (matplotlib.collections.EventCollection method), 1044
set_label() (matplotlib.collections.LineCollection method), 1057
set_label() (matplotlib.collections.PatchCollection method), 1069
set_label() (matplotlib.collections.PolyCollection method), 1092
set_label() (matplotlib.collections.QuadMesh method), 1105
set_label() (matplotlib.collections.RegularPolygonCollection method), 1117
set_label() (matplotlib.collections.StarPolygonCollection method), 1129
set_label() (matplotlib.collections.TriMesh method), 1141
set_label() (matplotlib.colorbar.ColorbarBase method), 1147
set_label() (matplotlib.container.Container method), 1165
set_label() (mpl_toolkits.axes_grid.axis_artist.AxisArtist method), 1803
set_label() (matplotlib.axis.Artist method), 1809
set_label() (matplotlib.axis.Tick method), 809
set_label1() (matplotlib.axis.XAxisArtist method), 811
set_label1() (matplotlib.axis.YAxisArtist method), 813
set_label2() (matplotlib.axis.XAxisArtist method), 809
set_label2() (matplotlib.axis.YAxisArtist method), 811
set_label1() (matplotlib.axis.XAxis method), 796
set_label1() (matplotlib.axis.YAxis method), 830
set_label2() (matplotlib.axis.XAxis method), 820
set_label2() (matplotlib.axis.YAxis method), 830
set_label_coords() (matplotlib.axis.XAxis method), 797
set_label_position() (matplotlib.axis.YAxis method), 820
set_label_text() (matplotlib.axis.Axis method), 797
set_label_text() (matplotlib.axis.XAxis method), 830
set_label_text() (matplotlib.axis.YAxis method), 821
set_linelength() (matplotlib.collections.EventCollection method), 1044
set_lineoffset() (matplotlib.collections.EventCollection method), 1045
set_linespacing() (matplotlib.text.Text method), 1629
set_linestyle() (matplotlib.backend_bases.GraphicsContextBase method), 922
set_linestyle() (matplotlib.collections.AsteriskPolygonCollection method), 984
set_linestyle() (matplotlib.collections.BrokenBarHCollection method), 996
set_linestyle() (matplotlib.collections.CircleCollection method), 1008
set_linestyle() (matplotlib.collections.Collection method), 1020
set_linestyle() (matplotlib.collections.EllipseCollection method), 1032
set_linestyle() (matplotlib.collections.EventCollection method), 1045
set_linestyle() (matplotlib.collections.LineCollection method), 1058
set_linestyle() (matplotlib.collections.PatchCollection method), 1069
set_linestyle() (matplotlib.collections.PathCollection method), 1081
set_linestyle() (matplotlib.collections.PatchCollection method), 1081
set_linestyle() (matplotlib.collections.PathCollection method), 1093
set_linestyle() (matplotlib.collections.PolyCollection method), 1093
set_linestyle() (matplotlib.collections.QuadMesh method), 1105
set_linestyle() (matplotlib.collections.QuadMesh method), 1105
set_linestyle() (matplotlib.collections.RegularPolyCollection method), 1117
set_linestyle() (matplotlib.collections.StarPolygonCollection method), 1129
set_linestyle() (matplotlib.collections.TriMesh method), 1141
set_linestyle() (matplotlib.lines.Line2D method), 1250
set_linestyle() (matplotlib.patches.Patch method), 1354
set_linestyles() (matplotlib.collections.AsteriskPolygonCollection method), 984
set_linestyles() (matplotlib.collections.BrokenBarHCollection method), 996
set_linestyles() (matplotlib.collections.CircleCollection method), 1008
set_linestyles() (matplotlib.collections.Collection method), 1020
set_linestyles() (matplotlib.collections.EllipseCollection method), 1032
set_linestyles() (matplotlib.collections.EventCollection method), 1045
set_linestyles() (matplotlib.collections.LineCollection method), 1058
set_linestyles() (matplotlib.collections.PatchCollection method), 1069
set_linestyles() (matplotlib.collections.PathCollection method), 1081
set_linestyles() (matplotlib.collections.PolyCollection method), 1081
set_linestyles() (matplotlib.collections.QuadMesh method), 1093
set_linestyles() (matplotlib.collections.RegularPolyCollection method), 1105
set_linestyles() (matplotlib.collections.StarPolygonCollection method), 1117
Method)

set_linestyles() (matplotlib.collections.TriMesh method), 1130
set_linewidths() (matplotlib.collections.AsteriskPolygonCollection method), 984
set_linestyles() (matplotlib.collections.BrokenBarHCollection method), 996
set_linestyles() (matplotlib.collections.CircleCollection method), 1008
set_linestyles() (matplotlib.collections.EventCollection method), 1020
set_linestyles() (matplotlib.collections.LineCollection method), 1032
set_linestyles() (matplotlib.collections.PatchCollection method), 1045
set_linestyles() (matplotlib.collections.PathCollection method), 1058
set_linestyles() (matplotlib.collections.PolyCollection method), 1069
set_linestyles() (matplotlib.collections.PatchCollection method), 1081
set_linestyles() (matplotlib.collections.PatchCollection method), 1093
set_linestyles() (matplotlib.collections.QuadMesh method), 1105
set_linestyles() (matplotlib.collections.RegularPolygonCollection method), 1117
set_linestyles() (matplotlib.collections.TriMesh method), 1130
set_locator() (mpl_toolkits.axes_grid.axes_divider.Divider method), 1798
set_locators() (matplotlib.ticker.FormatMethod method), 1638
set_locators() (matplotlib.ticker.LogFormatter method), 1641
set_locators() (matplotlib.ticker.ScalarFormatter method), 1639
set_ls() (matplotlib.lines.Line2D method), 1250
set_ls() (matplotlib.patches.Patch method), 1355
set_lw() (matplotlib.collections.AsteriskPolygonCollection method), 984
set_lw() (matplotlib.collections.BrokenBarHCollection method), 996
set_lw() (matplotlib.collections.CircleCollection method), 1008
set_lw() (matplotlib.collections.Collection method), 1021
set_lw() (matplotlib.collections.EllipseCollection method), 1032
set_lw() (matplotlib.collections.EventCollection method), 1045
set_lw() (matplotlib.collections.LineCollection method), 1058
set_lw() (matplotlib.collections.PatchCollection method), 1069
set_lw() (matplotlib.collections.PathCollection method), 1081
set_lw() (matplotlib.collections.PolyCollection method), 1093
set_lw() (matplotlib.collections.QuadMesh method), 1105
set_lw() (matplotlib.collections.RegularPolyCollection method), 1117
set_lw() (matplotlib.collections.StarPolygonCollection method), 1130
set_lw() (matplotlib.collections.TriMesh method), 1141
set_lw() (matplotlib.lines.Line2D method), 1250
set_lw() (matplotlib.patches.Patch method), 1355
set_ma() (matplotlib.text.Text method), 1630
set_major_formatter() (matplotlib.axis.Axis method), 796
set_major_formatter() (matplotlib.axis.XAxis method), 830
set_major_formatter() (matplotlib.axis.YAxis method), 821
set_major_locator() (matplotlib.axis.Axis method), 796
set_major_locator() (matplotlib.axis.XAxis method), 830
set_major_locator() (matplotlib.axis.YAxis method), 821
set_marker() (matplotlib.lines.Line2D method), 1250
set_marker() (matplotlib.markers.MarkerStyle method), 1257
set_markeredgewidth() (matplotlib.lines.Line2D method), 1251
set_markerfacecolor() (matplotlib.lines.Line2D method), 1251
set_markerfacecoloralt() (matplotlib.lines.Line2D method), 1251
set_markersize() (matplotlib.lines.Line2D method), 1251
set_markevery() (matplotlib.lines.Line2D method), 1251
set_mask() (matplotlib.tri.Triangulation method), 1672
set_matrix() (matplotlib.transforms.Affine2D method), 1663
set_mec() (matplotlib.lines.Line2D method), 1252
set_mfc() (matplotlib.lines.Line2D method), 1252
set_mfcalt() (matplotlib.lines.Line2D method), 1252
set_minimumdescent() (matplotlib.offsetbox.TextArea method), 1322
set_minor_formatter() (matplotlib.axis.Axis method), 796
set_minor_formatter() (matplotlib.axis.XAxis method), 831
set_minor_formatter() (matplotlib.axis.YAxis method), 821
set_minor_locator() (matplotlib.axis.Axis method), 796
set_minor_locator() (matplotlib.axis.XAxis method), 831
set_minor_locator() (matplotlib.axis.YAxis method), 821
set_ms() (matplotlib.lines.Line2D method), 1252
set_multialignment() (matplotlib.text.Text method), 1630
set_multilinebaseline() (matplotlib.offsetbox.TextArea method), 1322
set_mutation_aspect() (matplotlib.patches.FancyArrowPatch method), 1347
set_mutation_aspect() (matplotlib.patches.FancyBboxPatch method), 1350
set_mutation_scale() (matplotlib.patches.FancyArrowPatch method), 1347
set_mutation_scale() (matplotlib.patches.FancyBboxPatch method), 1350
set_name() (matplotlib.font_manager.FontProperties method), 1223
set_name() (matplotlib.text.Text method), 1630
set.navigate() (matplotlib.axes.Axes method), 771
set.navigate_mode() (matplotlib.axes.Axes method), 771
set_norm() (matplotlib.cm.ScalarMappable method), 972
set_norm() (matplotlib.cm.ScalarMappable method), 1233
set_offset() (matplotlib.offsetbox.AuxTransformBox method), 1316
set_offset() (matplotlib.offsetbox.DrawingArea method), 1318
set_offset() (matplotlib.offsetbox.OffsetBox method), 1320
set_offset() (matplotlib.offsetbox.TextArea method), 1322
set_offset_position() (matplotlib.axis.YAxis method), 804
set_offset_position() (matplotlib.collections.AsteriskPolygonCollection method), 985
set_offset_position() (matplotlib.collections.BrokenBarHCollection method), 996
set_offset_position() (matplotlib.collections.CircleCollection method), 1008
set_offset_position() (matplotlib.collections.Collection method), 1021
set_offset_position() (matplotlib.collections.Collection method), 1021
set_offset_position() (matplotlib.collections.EllipseCollection method), 1032
set_offset_position() (matplotlib.collections.EventCollection method), 1045
set_offset_position() (matplotlib.collections.LineCollection method), 1058
set_offset_position() (matplotlib.collections.PatchCollection method), 1069
set_offset_position() (matplotlib.collections.PatchCollection method), 1070
set_offset_position() (matplotlib.collections.PathCollection method), 1081
set_offset_position() (matplotlib.collections.PathCollection method), 1081
set_offset_position() (matplotlib.collections.PolyCollection method), 1093
set_offset_position() (matplotlib.collections.PolyCollection method), 1093
set_offset_position() (matplotlib.collections.QuadMesh method), 1105
set_offset_position() (matplotlib.collections.QuadMesh method), 1105
set_offset_position() (matplotlib.collections.RegularPolygonCollection method), 1117
set_offset_position() (matplotlib.collections.RegularPolygonCollection method), 1130
set_offset_position() (matplotlib.collections.TriMesh method), 1141
set_offset_position() (matplotlib.collections.TriMesh method), 1141
set_offset_position() (matplotlib.collections.TriMesh method), 1141
set_offset_position() (matplotlib.collections.StarPolygonCollection method), 1130
set_offset_position() (matplotlib.collections.TriMesh method), 1142
set_offset_string() (matplotlib.ticker.FixedFormatter method), 1638
set_offsets() (matplotlib.collections.AsteriskPolygonCollection method), 985
set_offsets() (matplotlib.collections.BrokenBarHCollection method), 997
set_offsets() (matplotlib.collections.CircleCollection method), 1009
set_offsets() (matplotlib.collections.Collection method), 1021
set_offsets() (matplotlib.collections.EllipseCollection method), 1032
set_offsets() (matplotlib.collections.EventCollection method), 1045
set_offsets() (matplotlib.collections.LineCollection method), 1058
set_offsets() (matplotlib.collections.PathCollection method), 1081
set_offsets() (matplotlib.collections.PatchCollection method), 1093
set_offsets() (matplotlib.collections.QuadMesh method), 1105
set_offsets() (matplotlib.collections.RegularPolyCollection method), 1118
set_offsets() (matplotlib.collections.StarPolygonCollection method), 1130
set_offsets() (matplotlib.collections.TriMesh method), 1142
set_orientation() (matplotlib.collections.EventCollection method), 1046
set_over() (matplotlib.colors.Colormap method), 1153
set_pad() (matplotlib.axis.Tick method), 810
set_pad() (matplotlib.axis.XTick method), 811
set_pad() (matplotlib.axis.YTick method), 813
set_pad() (mpl_toolkits.axes_grid.axis_artist.AxesLabel method), 1804
set_pane_color() (mpl_toolkits.mplot3d.axis3d.Axes3D method), 1870
set_pane_pos() (mpl_toolkits.mplot3d.axis3d.Axes3D method), 1870
set_params() (matplotlib.ticker.FixedLocator method), 1643
set_params() (matplotlib.ticker.IndexLocator method), 1643
set_params() (matplotlib.ticker.LinearLocator method), 1643
set_params() (matplotlib.ticker.LogLocator method), 1644
set_params() (matplotlib.ticker.MaxNLocator method), 1645
set_params() (matplotlib.ticker.MultipleLocator method), 1644
set_params() (matplotlib.ticker.SymmetricalLogLocator method), 1646
set_patch_circle() (matplotlib.spines.Spines method), 1617
set_patch_line() (matplotlib.spines.Spines method), 1617
set_patchA() (matplotlib.patches.FancyArrowPatch method), 1347
set_patchB() (matplotlib.patches.FancyArrowPatch method), 1347
set_path_effects() (matplotlib.artist.Artist method), 545
set_path_effects() (matplotlib.axes.Axes method), 782
set_path_effects() (matplotlib.axis.Axes method), 887
set_path_effects() (matplotlib.axis.Tick method), 847
set_path_effects() (matplotlib.axis.XAxis method), 898
set_path_effects() (matplotlib.axis.XTick method), 858
set_path_effects() (matplotlib.axis.YAxis method), 909
set_path_effects() (matplotlib.axis.YTick method), 870
set_path_effects() (matplotlib.collections.AsteriskPolygonCollection method), 985
set_path_effects() (matplotlib.collections.BrokenBarHCollection method), 1870
set_path_effects() (matplotlib.collections.AsteriskPolygonCollection method), 985
set_path_effects() (matplotlib.collections.BrokenBarHCollection method), 997
set_path_effects() (matplotlib.collections.CircleCollection method), 1009
set_path_effects() (matplotlib.collections.Collection method), 1021
set_path_effects() (matplotlib.collections.EllipseCollection method), 1033
set_path_effects() (matplotlib.collections.EventCollection method), 1046
set_path_effects() (matplotlib.collections.LineCollection method), 1058
set_path_effects() (matplotlib.collections.PatchCollection method), 1070
set_path_effects() (matplotlib.collections.PathCollection method), 1082
set_path_effects() (matplotlib.collections.PolyCollection method), 1094
set_path_effects() (matplotlib.collections.QuadMesh method), 1106
set_path_effects() (matplotlib.collections.RegularPolyCollection method), 1118
set_path_effects() (matplotlib.collections.StarPolygonCollection method), 1130
set_path_effects() (matplotlib.collections.TriMesh method), 1142
set_paths() (matplotlib.collections.AsteriskPolygonCollection method), 985
set_paths() (matplotlib.collections.BrokenBarHCollection method), 997
set_paths() (matplotlib.collections.CircleCollection method), 1009
set_paths() (matplotlib.collections.Collection method), 1021
set_paths() (matplotlib.collections.EllipseCollection method), 1033
set_paths() (matplotlib.collections.EventCollection method), 1046
set_paths() (matplotlib.collections.LineCollection method), 1058
set_paths() (matplotlib.collections.PatchCollection method), 1070
set_paths() (matplotlib.collections.PathCollection method), 1082
set_paths() (matplotlib.collections.PolyCollection method), 1094
set_paths() (matplotlib.collections.QuadMesh method), 1106
set_paths() (matplotlib.collections.RegularPolyCollection method), 1118
set_paths() (matplotlib.collections.StarPolygonCollection method), 1130
set_pick() (matplotlib.artist.Artist method), 539
set_pick() (matplotlib.axes.Axes method), 774
set_pick() (matplotlib.axis.Axis method), 887
set_pick() (matplotlib.axis.Tick method), 847
set_pick() (matplotlib.axis.XAxis method), 898
set_pick() (matplotlib.axis.XTick method), 858
set_pick() (matplotlib.axis.YAxis method), 909
set_pick() (matplotlib.axis.YTick method), 870
set_pick() (matplotlib.collections.AsteriskPolygonCollection method), 985
set_pick() (matplotlib.collections.BrokenBarHCollection method), 997
set_pick() (matplotlib.collections.CircleCollection method), 1009
set_pick() (matplotlib.collections.Collection method), 1021
set_pick() (matplotlib.collections.EllipseCollection method), 1033
set_pick() (matplotlib.collections.EventCollection method), 1046
set_pick() (matplotlib.collections.LineCollection method), 1058
set_pick() (matplotlib.collections.PatchCollection method), 1070
set_pick() (matplotlib.collections.PathCollection method), 1082
set_pick() (matplotlib.collections.PolyCollection method), 1094
set_pick() (matplotlib.collections.QuadMesh method), 1106
set_pick() (matplotlib.collections.RegularPolyCollection method), 1118
set_pick() (matplotlib.collections.StarPolygonCollection method), 1130
set_pick() (matplotlib.collections.TriMesh method), 1142
set_picker() (matplotlib.collections.TriMesh method), 1142
set_picker() (matplotlib.lines.Line2D method), 1252
set_pickradius() (matplotlib.axis.Axis method), 802
set_pickradius() (matplotlib.axis.XAxis method), 831
set_pickradius() (matplotlib.axis.YAxis method), 821
set_pickradius() (matplotlib.collections.AsteriskPolygonCollection method), 985
set_pickradius() (matplotlib.collections.BrokenBarHCollection method), 997
set_pickradius() (matplotlib.collections.CircleCollection method), 1009
set_pickradius() (matplotlib.collections.Collection method), 1021
set_pickradius() (matplotlib.collections.EllipseCollection method), 1033
set_pickradius() (matplotlib.collections.EventCollection method), 1046
set_pickradius() (matplotlib.collections.LineCollection method), 1059
set_pickradius() (matplotlib.collections.PatchCollection method), 1070
set_pickradius() (matplotlib.collections.PathCollection method), 1082
set_pickradius() (matplotlib.collections.PolyCollection method), 1094
set_pickradius() (matplotlib.collections.QuadMesh method), 1106
set_pickradius() (matplotlib.collections.RegularPolyCollection method), 1118
set_pickradius() (matplotlib.collections.StarPolygonCollection method), 1131
set_pickradius() (matplotlib.collections.TriMesh method), 1142
set_pickradius() (matplotlib.lines.Line2D method), 1252
set_points() (matplotlib.transforms.Bbox method), 1655
set_position() (matplotlib.axes.Axes method), 768
set_position() (matplotlib.spines.Spine method), 1617
set_position() (matplotlib.text.Text method), 1630
set_position() (matplotlib.text.TextWithDash method), 1634
set_positions() (matplotlib.collections.EventCollection method), 1046
set_positions() (matplotlib.patches.FancyArrowPatch method), 1347
set_powerlimits() (matplotlib.ticker.ScalarFormatter method), 1639
set_prop_cycle() (matplotlib.axes.Axes method), 736
set_radius() (matplotlib.patches.Circle method), 1337
set_radius() (matplotlib.patches.Wedge method), 1363
set_rasterization_zorder() (matplotlib.axes.Axes method), 777
set_rasterized() (matplotlib.artist.Artist method), 545
set_rasterized() (matplotlib.axes.Axes method), 781
set_rasterized() (matplotlib.axes.Axes method), 887
set_rasterized() (matplotlib.axis.Tick method), 848
set_rasterized() (matplotlib.axis.XAxis method), 898
set_rasterized() (matplotlib.axis.XTick method), 859
set_rasterized() (matplotlib.axis.YAxis method), 910
set_rasterized() (matplotlib.axis.YTick method), 870
set_rasterized() (matplotlib.collections.AsteriskPolygonCollection method), 985
set_rasterized() (matplotlibcollections.BrokenBarHCollection method), 997
set_rasterized() (matplotlibcollections.CircleCollection method), 1009
set_rasterized() (matplotlibcollections.Collection method), 1021
set_rasterized() (mat-
set_solid_capstyle() (matplotlib.lines.Line2D method), 1252
set_solid_joinstyle() (matplotlib.lines.Line2D method), 1252
set_sort_zpos() (mpl_toolkits.mplot3d.art3d.Line3DCollection method), 1871
set_sort_zpos() (mpl_toolkits.mplot3d.art3d.Patch3DCollection method), 1872
set_sort_zpos() (mpl_toolkits.mplot3d.art3d.Path3DCollection method), 1873
set_solid_capstyle() (matplotlib.artist.Artist method), 544
set_solid_joinstyle() (matplotlib.artist.Artist method), 544
set_snap() (matplotlib.axes.Axes method), 782
set_snap() (matplotlib.artist.Artist method), 782
set_snap() (matplotlib.axis.Axes method), 888
set_snap() (matplotlib.axis.Tick method), 848
set_snap() (matplotlib.axis.XAxis method), 899
set_snap() (matplotlib.axis.XTick method), 859
set_snap() (matplotlib.axis.YAxis method), 910
set_snap() (matplotlib.axis.YTick method), 871
set_snap() (matplotlib.backend_bases.GraphicsContextBase method), 922
set_snap() (matplotlib.collections.AsteriskPolygonCollection method), 986
set_snap() (matplotlib.collections.BrokenBarHCollection method), 998
set_snap() (matplotlib.collections.CircleCollection method), 1010
set_snap() (matplotlib.collections.Collection method), 1022
set_snap() (matplotlib.collections.EllipseCollection method), 1034
set_snap() (matplotlib.collections.EventCollection method), 1047
set_snap() (matplotlib.collections.LineCollection method), 1059
set_snap() (matplotlib.collections.PatchCollection method), 1071
set_snap() (matplotlib.collections.PathCollection method), 1083
set_snap() (matplotlib.collections.PolyCollection method), 1095
set_snap() (matplotlib.collections.QuadMesh method), 1107
set_snap() (matplotlib.collections.RegularPolyCollection method), 1119
set_snap() (matplotlib.collections.TriMesh method), 1131
set_snap() (matplotlib.collections.TriMesh method), 1131
set_snap() (matplotlib.collections.TriMesh method), 1143
set_snap() (matplotlib.collections.TriMesh method), 1143
set_snap() (matplotlib.collections.TriMesh method), 1143
set_sort_zpos() (mpl_toolkits.mplot3d.art3d.Line3DCollection method), 1871
set_sort_zpos() (mpl_toolkits.mplot3d.art3d.Patch3DCollection method), 1872
set_sort_zpos() (mpl_toolkits.mplot3d.art3d.Path3DCollection method), 1873
set_ske
set_sort_zpos() (mpl_toolkits.mplot3d.art3d.Poly3DCollection method), 1874
set_stretch() (matplotlib.font_manager.FontProperties method), 1223
set_stretch() (matplotlib.text.Text method), 1630
set_style() (matplotlib.font_manager.FontProperties method), 1223
set_style() (matplotlib.text.Text method), 1630
set_subplotspec() (mpl_toolkits.axes_grid.axes_divider.SubplotDivider method), 1799
set_text() (matplotlib.offsetbox.TextArea method), 1322
set_text() (matplotlib.text.Text method), 1630
set_theta1() (matplotlib.patches.Wedge method), 1363
set_theta2() (matplotlib.patches.Wedge method), 1363
set_theta_direction() (matplotlib.projections.polar.PolarAxes method), 1384
set_theta_offset() (matplotlib.projections.polar.PolarAxes method), 1384
set_theta_zero_location() (matplotlib.projections.polar.PolarAxes method), 1384
set_thetagrids() (matplotlib.projections.polar.PolarAxes method), 1384
set_tick_out() (mpl_toolkits.axes_grid.axis_artist.Ticks method), 1803
set_tick_params() (matplotlib.axis.Axis method), 807
set_tick_params() (matplotlib.axis.XAxis method), 831
set_tick_params() (matplotlib.axis.YAxis method), 822
set_ticklabel_direction() (mpl_toolkits.axes_grid.axis_artist.AxesArtist method), 1803
set_ticklabel_direction() (mpl_toolkits.axes_grid.axis_artist.AxesArtist method), 1803
set_ticklabels() (matplotlib.axis.Axis method), 807
set_ticklabels() (matplotlib.axis.XAxis method), 831
set_ticklabels() (matplotlib.axis.YAxis method), 822
set_ticklabels() (matplotlib.colorbar.ColorbarBase method), 1147
set_ticks() (matplotlib.axes.Axes method), 743
set_ticks() (matplotlib.axis.XAxis method), 805
set_ticks() (matplotlib.axis.YAxis method), 804
set_ticks() (matplotlib.axes_grid.axis_artist.Ticks method), 1804
set_title() (matplotlib.figure.Figure method), 1202
set_title() (matplotlib.axes.Axes method), 743
set_title() (matplotlib.legend.Legend method), 1241
set_title() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 1862
set_top_view() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 1862
set_transform() (matplotlib.artist.Artist method), 548
set_transform() (matplotlib.axis.XAxis method), 888
set_transform() (matplotlib.axis.Tick method), 848
set_transform() (matplotlib.axis.XAxis method), 899
set_transform() (matplotlib.axis.XTick method), 860
set_transform() (matplotlib.axis.YAxis method), 910
set_transform() (matplotlib.axis.YTick method), 871
set_transform() (matplotlib.collections.AsteriskPolygonCollection method), 986
set_transform() (matplotlib.collections.BrokenBarHCollection method), 998
set_transform() (matplotlib.collections.CircleCollection method), 1010
set_transform() (matplotlib.collections.Collection method), 1022
set_transform() (matplotlib.collections.EllipseCollection method), 1034
set_transform() (matplotlib.collections.EventCollection method), 1047
set_transform() (matplotlib.collections.LineCollection method), 1059
set_transform() (matplotlib.collections.PatchCollection method), 1071
set_transform() (matplotlib.collections.PathCollection method), 1071
set_transform() (matplotlib.collections.PathCollection method), 1083
set_transform() (matplotlib.collections.PolyCollection method), 1095
set_transform() (matplotlib.collections.QuadMesh method), 1107
set_transform() (matplotlib.collections.RegularPolyCollection method), 1119
set_transform() (matplotlib.collections.StarPolygonCollection method), 1132
set_transform() (matplotlib.collections.TriMesh method), 1143
set_transform() (matplotlib.offsetbox.AuxTransformBox method), 1316
set_transform() (matplotlib.offsetbox.DrawingArea method), 1318
set_transform() (matplotlib.offsetbox.TextArea method), 1323
set_transform() (matplotlib.text.TextWithDash method), 1634
set_tzinfo() (matplotlib.dates DateFormatter method), 1170
set_tzinfo() (matplotlib.dates.DateLocator method), 1171
set_under() (matplotlib.colors.Colormap method), 1153
set_unit() (matplotlib.text.OffsetFrom method), 1624
set_units() (matplotlib.axis.Axis method), 802
set_units() (matplotlib.axis.XAxis method), 832
set_units() (matplotlib.axis.YAxis method), 822
set_url() (matplotlib.artist.Artist method), 550
set_url() (matplotlib.axes.Axes method), 781
set_url() (matplotlib.axis.Axis method), 888
set_url() (matplotlib.axis.Tick method), 849
set_url() (matplotlib.axis.XAxis method), 899
set_url() (matplotlib.axis.XTick method), 860
set_url() (matplotlib.axis.YAxis method), 911
set_url() (matplotlib.axis.YTick method), 871
set_url() (matplotlib.backend_bases.GraphicsContextBase method), 923
set_url() (matplotlib.collections.AsteriskPolygonCollection method), 986
set_url() (matplotlib.collections.BrokenBarHCollection method), 998
set_url() (matplotlib.collections.CircleCollection method), 1010
set_url() (matplotlib.collections.Collection method), 1022
set_url() (matplotlib.collections.EllipseCollection method), 1034
set_url() (matplotlib.collections.EventCollection method), 1047
set_url() (matplotlib.collections.LineCollection method), 1059
set_url() (matplotlib.collections.PathCollection method), 1071
set_url() (matplotlib.collections.PatchCollection method), 1083
set_url() (matplotlib.collections.PolyCollection method), 1095
set_url() (matplotlib.collections.QuadMesh method), 1107
set_url() (matplotlib.collections.RegularPolyCollection method), 1119
set_url() (matplotlib.collections.StarPolygonCollection method), 1132
set_url() (matplotlib.collections.TriMesh method), 1143
set_urls() (matplotlib.collections.AsteriskPolygonCollection method), 986
set_urls() (matplotlib.collections.BrokenBarHCollection method), 998
set_urls() (matplotlib.collections.CircleCollection method), 1010
set_urls() (matplotlib.collections.Collection method), 1022
set_urls() (matplotlib.collections.EllipseCollection method), 1034
set_urls() (matplotlib.collections.EventCollection method), 1047
set_urls() (matplotlib.collections.LineCollection method), 1059
set_urls() (matplotlib.collections.PatchCollection method), 1071
set_urls() (matplotlib.collections.PathCollection method), 1083
set_urls() (matplotlib.collections.PolyCollection method), 1095
set_urls() (matplotlib.collections.QuadMesh method), 1107
set_urls() (matplotlib.collections.StarPolygonCollection method), 1132
set_urls() (matplotlib.collections.TriMesh method), 1143
set_url() (matplotlib.collections.AsteriskPolygonCollection method), 986
set_url() (matplotlib.collections.BrokenBarHCollection method), 998
set_url() (matplotlib.collections.CircleCollection method), 1010
set_url() (matplotlib.collections.EllipseCollection method), 1034
set_url() (matplotlib.collections.EventCollection method), 1047
set_url() (matplotlib.collections.LineCollection method), 1059
set_url() (matplotlib.collections.PatchCollection method), 1071
set_url() (matplotlib.collections.PathCollection method), 1083
set_url() (matplotlib.collections.PolyCollection method), 1095
set_url() (matplotlib.collections.QuadMesh method), 1107
set_url() (matplotlib.collections.StarPolygonCollection method), 1132
set_url() (matplotlib.collections.TriMesh method), 1143
set_url() (matplotlib.collections.AsteriskPolygonCollection method), 986
set_url() (matplotlib.collections.BrokenBarHCollection method), 998
set_url() (matplotlib.collections.CircleCollection method), 1010
set_url() (matplotlib.collections.EllipseCollection method), 1034
set_url() (matplotlib.collections.EventCollection method), 1047
set_url() (matplotlib.collections.LineCollection method), 1059
set_url() (matplotlib.collections.PatchCollection method), 1071
set_url() (matplotlib.collections.PathCollection method), 1083
set_url() (matplotlib.collections.PolyCollection method), 1095
set_url() (matplotlib.collections.QuadMesh method), 1107
set_url() (matplotlib.collections.StarPolygonCollection method), 1132
set_url() (matplotlib.collections.TriMesh method), 1143
set_url() (matplotlib.collections.AsteriskPolygonCollection method), 986
set_url() (matplotlib.collections.BrokenBarHCollection method), 998
set_url() (matplotlib.collections.CircleCollection method), 1010
set_url() (matplotlib.collections.EllipseCollection method), 1034
set_url() (matplotlib.collections.EventCollection method), 1047
set_url() (matplotlib.collections.LineCollection method), 1059
set_url() (matplotlib.collections.PatchCollection method), 1071
set_url() (matplotlib.collections.PathCollection method), 1083
set_url() (matplotlib.collections.PolyCollection method), 1095
set_url() (matplotlib.collections.QuadMesh method), 1107

set_urls() (matplotlib.collections.RegularPolyCollection method), 1119
set_urls() (matplotlib.collections.StarPolygonCollection method), 1132
set_urls() (matplotlib.collections.TriMesh method), 1143
set_useLocale() (matplotlib.ticker.ScalarFormatter method), 1640
set_useOffset() (matplotlib.ticker.ScalarFormatter method), 1640
set_usetex() (matplotlib.text.Text method), 1631
set_va() (matplotlib.text.Text method), 1631
set_val() (matplotlib.widgets.Slider method), 1698
set_variant() (matplotlib.font_manager.FontProperties method), 1223
set_variant() (matplotlib.text.Text method), 1631
set_vertical() (mpl_toolkits.axes_grid.axes_divider.Divider method), 1798
set_verticalalignment() (matplotlib.text.Text method), 1631
set_verts() (matplotlib.collections.BrokenBarHCollection method), 998
set_verts() (matplotlib.collections.EventCollection method), 1047
set_verts() (matplotlib.collections.LineCollection method), 1060
set_verts() (matplotlib.collections.PolyCollection method), 1095
set_verts() (mpl_toolkits.mplot3d.art3d.Poly3DCollection method), 1874
set_verts_and_codes() (matplotlib.collections.BrokenBarHCollection method), 998
set_verts_and_codes() (matplotlib.collections.PolyCollection method), 1095
set_verts_and_codes() (mpl_toolkits.mplot3d.art3d.Poly3DCollection method), 1874
set_view_interval() (matplotlib.axis.Axes method), 801
set_view_interval() (matplotlib.axis.XAxis method), 832
set_view_interval() (matplotlib.axis.YAxis method), 822
set_view_interval() (matplotlib.collections.AsteriskPolygonCollection method), 986
set_view_interval() (matplotlib.collections.BrokenBarHCollection method), 998
set_view_interval() (matplotlib.collections.CircleCollection method), 1010
set_view_interval() (matplotlib.collections.EllipseCollection method), 1022
set_view_interval() (matplotlib.collections.LineCollection method), 1047
set_view_interval() (matplotlib.collections.PatchCollection method), 1060
set_view_interval() (matplotlib.collections.PolyCollection method), 1071
set_view_interval() (matplotlib.collections.PathCollection method), 1083
set_view_interval() (matplotlib.collections.QuadMesh method), 1107
set_view_interval() (matplotlib.collections.RegularPolyCollection method), 1119
set_view_interval() (matplotlib.collections.StarPolygonCollection method), 1132
set_view_interval() (matplotlib.collections.TriMesh method), 1143
set_visible() (matplotlib.artist.Artist method), 545
set_visible() (matplotlib.axes.Axes method), 781
set_visible() (matplotlib.axis.Axes method), 888
set_visible() (matplotlib.axis.Tick method), 849
set_visible() (matplotlib.axis.XAxis method), 900
set_visible() (matplotlib.axis.YAxis method), 911
set_visible() (matplotlib.axis.XTick method), 860
set_visible() (matplotlib.axis.YTick method), 871
set_visible() (matplotlib.collections.AsteriskPolygonCollection method), 986
set_visible() (matplotlib.collections.BrokenBarHCollection method), 998
set_visible() (matplotlib.collections.CircleCollection method), 1010
set_visible() (matplotlib.collections.EllipseCollection method), 1022
set_visible() (matplotlib.collections.LineCollection method), 1047
set_visible() (matplotlib.collections.PatchCollection method), 1060
set_visible() (matplotlib.collections.PolyCollection method), 1071
set_visible() (matplotlib.collections.PathCollection method), 1083
set_visible() (matplotlib.collections.QuadMesh method), 1107
set_visible() (matplotlib.collections.StarPolygonCollection method), 1132
set_visible() (matplotlib.collections.TriMesh method), 1143
set_visible() (matplotlib.widgets.ToolHandles method), 1701
set_weight() (matplotlib.font_manager.FontProperties method), 1223
set_weight() (matplotlib.text.Text method), 1631
set_width() (matplotlib.offsetbox.OffsetBox method), 1320
set_width() (matplotlib.patches.FancyBboxPatch method), 1350
set_width() (matplotlib.patches.Rectangle method), 1359
set_width() (matplotlib.patches.Wedge method), 1364
set_width_ratios() (matplotlib.gridspec.GridSpec method), 1228
set_window_title() (matplotlib.backend_bases.FigureCanvasBase method), 918
set_window_title() (matplotlib.backend_bases.FigureManagerBase method), 919
set_wrap() (matplotlib.text.Text method), 1631
set_x() (matplotlib.patches.FancyBboxPatch method), 1350
set_x() (matplotlib.patches.Rectangle method), 1360
set_x() (matplotlib.text.Text method), 1631
set_x() (matplotlib.text.TextWithDash method), 1634
set_xbound() (matplotlib.axes.Axes method), 742
set_xdata() (matplotlib.lines.Line2D method), 1253
set_xlabel() (matplotlib.axes.Axes method), 743
set_xlim() (matplotlib.axes.Axes method), 738
set_xlim() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 1863
set_xlim3d() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 1863
set_xmargin() (matplotlib.axes.Axes method), 752
set_xscale() (matplotlib.axes.Axes method), 750
set_xscale() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 1864
set_xticklabels() (matplotlib.axes.Axes method), 760
set_xticks() (matplotlib.axes.Axes method), 761
set_zbound() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 1865
set_zlabel() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 1865
set_zlim() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 1865
set_zlim3d() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 1865
set_zmargin() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 1865
set_zoom() (matplotlib.offsetbox.OffsetImage method), 1320
set_zorder() (matplotlib.artist.Artist method), 546
set_zorder() (matplotlib.axes.Axes method), 781
set_zorder() (matplotlib.axis.Axis method), 889
set_zorder() (matplotlib.axis.Tick method), 849
set_zorder() (matplotlib.axis.XAxis method), 900
set_zorder() (matplotlib.axis.XTick method), 860
set_zorder() (matplotlib.axis.YAxis method), 911
set_zorder() (matplotlib.axis.YTick method), 872
set_zorder() (matplotlib.collections.AsteriskPolygonCollection method), 986
set_zorder() (matplotlib.collections.BrokenBarHCollection method), 998
set_zorder() (matplotlib.collections.CircleCollection method), 1010
set_zorder() (matplotlib.collections.Collection method), 1022
set_zorder() (matplotlib.collections.EllipseCollection method), 1034
set_zorder() (matplotlib.collections.EventCollection method), 1047
set_zorder() (matplotlib.collections.LineCollection method), 1060
set_zorder() (matplotlib.collections.PatchCollection method), 1072
set_zorder() (matplotlib.collections.PathCollection method), 1071
set_zorder() (matplotlib.collections.PolyCollection method), 1095
set_zorder() (matplotlib.collections.QuadMesh method), 1107
set_zorder() (matplotlib.collections.RegularPolyCollection method), 1119
set_zorder() (matplotlib.collections.StarPolygonCollection method), 1132
set_zorder() (matplotlib.collections.TriMesh method), 1143
set_zscale() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 1865
set_zsort() (mpl_toolkits.mplot3d.art3d.Poly3DCollection method), 1874
set_zticklabels() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 1866
set_zticks() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 1866
SetCursorBase (class in matplotlib.backend_tools), 938
setp() (in module matplotlib.artist), 552
setp() (in module matplotlib.pyplot), 1551
setup() (matplotlib.animation.FileMovieWriter method), 523
setup() (matplotlib.animation.MovieWriter method), 522
shade() (matplotlib.colors.LightSource method), 1156
shade_rgb() (matplotlib.colors.LightSource method), 1157
Shadow (class in matplotlib.patches), 1361
Ship (class in matplotlib.mathtext), 1274
short_float_fmt() (in module matplotlib.backends.backend_svg), 951
should_simplify (matplotlib.path.Path attribute), 1372
show() (in module matplotlib.pyplot), 1551
show() (matplotlib.backend_bases.FigureManagerBase method), 1919
show() (matplotlib.figure.Figure method), 1202
show_popup() (matplotlib.backend_bases.FigureManagerBase method), 920
ShowBase (class in matplotlib.backend_bases), 931
shrink() (matplotlib.mathtext.Accent method), 1261
shrink() (matplotlib.mathtext.Box method), 1262
shrink() (matplotlib.mathtext.Char method), 1262
shrink() (matplotlib.mathtext.Glue method), 1266
shrink() (matplotlib.mathtext.Kern method), 1267
shrink() (matplotlib.mathtext.List method), 1267
shrink() (matplotlib.mathtext.Node method), 1271
shrink() (matplotlib.transforms.BboxBase method), 1653
shrunk_to_aspect() (matplotlib.transforms.BboxBase method), 1653
silent_list (class in matplotlib.cbook), 966
silverman_factor() (matplotlib.mlab.GaussianKDE method), 1283
simple_group() (matplotlib.mathtext.Parser method), 1273
simple_linear_interpolation() (in module matplotlib.pyplot), 966
SimpleLineShadow (class in matplotlib.pathEffects), 1376
SimplePatchShadow (class in matplotlib.pathEffects), 1377
simplify_threshold (matplotlib.path.Path attribute), 1372
single_shot (matplotlib.backend_bases.TimerBase attribute), 932
size (matplotlib.dviread.DviFont attribute), 1182
size (matplotlib.transforms.BboxBase attribute), 1653
skew() (matplotlib.transforms.Affine2D method), 1664
skew_deg() (matplotlib.transforms.Affine2D method), 1664
Slider (class in matplotlib.widgets), 1697
slopes() (in module matplotlib.mlab), 1307
snowflake() (matplotlib.mathtext.Parser method), 1273
sort() (matplotlib.cbook.Sor ter method), 956
Sorter (class in matplotlib.cbook), 956
soundex() (in module matplotlib.cbook), 966
span_where() (matplotlib.mathtext.Parser method), 1273
span_where() (matplotlib.collections.BrokenBarHCollection static method), 999
SpanSelector (class in matplotlib.widgets), 1698
specgram() (in module matplotlib.mlab), 1307
specgram() (in module matplotlib.pyplot), 1552
specgram() (matplotlib.axes.Axes method), 631
tribute), 1047
sticky_edges (matplotlib.collections.LineCollection attribute), 1060
sticky_edges (matplotlib.collections.PatchCollection attribute), 1071
sticky_edges (matplotlib.collections.PathCollection attribute), 1083
sticky_edges (matplotlib.collections.PolyCollection attribute), 1095
sticky_edges (matplotlib.collections.QuadMesh attribute), 1107
sticky_edges (matplotlib.collections.RegularPolyCollection attribute), 1120
sticky_edges (matplotlib.collections.StarPolygonCollection attribute), 1132
sticky_edges (matplotlib.collections.TriMesh attribute), 1143
stinetman_interp() (in module matplotlib.mlab), 1309
STIXFontConstants (class in matplotlib.mathtext), 1274
STIXSansFontConstants (class in matplotlib.mathtext), 1275
STIXSansFonts (class in matplotlib.mathtext), 1275
STOP (matplotlib.path.Path attribute), 1369
stop() (matplotlib.backend_bases.TimerBase method), 932
stop_event_loop() (matplotlib.backend_bases.FigureCanvasBase method), 919
stop_event_loop_default() (matplotlib.backend_bases.FigureCanvasBase method), 919
stop_filter() (matplotlib.backend_bases.RendererBase method), 930
stop_rasterizing() (matplotlib.backend_bases.RendererBase method), 930
Stream (class in matplotlib.backends.backend_pdf), 951
streamplot() (in module matplotlib.pyplot), 1559
streamplot() (matplotlib.axes.Axes method), 730
strftime() (matplotlib.dates.DateFormatter method), 1170
strftime_pre_1900() (matplotlib.dates.DateFormatter method), 1170
stride_repeat() (in module matplotlib.mlab), 1310
stride_windows() (in module matplotlib.mlab), 1310
string_width_height() (matplotlib.afm.AFM method), 488
strip_math() (in module matplotlib.cbook), 966
strip_math() (matplotlib.backend_bases.RendererBase method), 930
StrMethodFormatter (class in matplotlib.ticker), 1639
Stroke (class in matplotlib.path.effects), 1377
sub1 (matplotlib.mathtext.ComputerModernFontConstants attribute), 1263
sub1 (matplotlib.mathtext.FontConstantsBase attribute), 1264
sub2 (matplotlib.mathtext.ComputerModernFontConstants attribute), 1263
sub2 (matplotlib.mathtext.FontConstantsBase attribute), 1264
sub2 (matplotlib.mathtext.STIXFontConstants attribute), 1274
subdrop (matplotlib.mathtext.ComputerModernFontConstants attribute), 1263
subdrop (matplotlib.mathtext.FontConstantsBase attribute), 1264
subplot() (in module matplotlib.pyplot), 1560
subplot2grid() (in module matplotlib.pyplot), 1561
subplot_tool() (in module matplotlib.pyplot), 1561
SubplotDivider (class in mpl_toolkits.axes_grid.axes_divider), 1799
SubplotParams (class in matplotlib.figure), 1205
subplots() (in module matplotlib.pyplot), 1562
subplots_adjust() (in module matplotlib.pyplot), 1564
subplots_adjust() (matplotlib.figure.Figure method), 1202
SubplotSpec (class in matplotlib.gridspec), 1229
SubplotTool (class in matplotlib.widgets), 1700
subs() (matplotlib.tick.LogLocator method), 1644
subsuper() (matplotlib.mathtext.Parser method), 1273
SubSuperCluster (class in matplotlib.mathtext), 1276
summer() (in module matplotlib.pyplot), 1564
sup1 (matplotlib.mathtext.ComputerModernFontConstants attribute), 1263
sup1 (matplotlib.mathtext.FontConstantsBase attribute), 1264
attribute), 1264
sup1 (matplotlib.mathtext.STIXFontConstants attribute), 1274
sup1 (matplotlib.mathtext.STIXSansFontConstants attribute), 1274
supports_blit (matplotlib.backend_bases.FigureCanvasBase attribute), 919
sup1 (matplotlib.figure.Figure method), 1202
SVG, 3018
switch_backend() (in module matplotlib.pyplot), 1564
switch_backends() (matplotlib.backend_bases.FigureCanvasBase method), 919
switch_orientation() (matplotlib.collections.EventCollection method), 1048
symbol() (matplotlib.mathtext.Parser method), 1273
SymLogNorm (class in matplotlib.colors), 1161
SymmetricalLogLocator (class in matplotlib.ticker), 1645
SymmetricalLogScale (class in matplotlib.scale), 1612
SymmetricalLogScale.InvertedSymmetricalLogTransform (class in matplotlib.scale), 1612
SymmetricalLogScale.SymmetricalLogTransform (class in matplotlib.scale), 1612
SymmetricalLogTransform (class in matplotlib.scale), 1613
T
table() (in module matplotlib.pyplot), 1565
table() (matplotlib.axes.Axes method), 717
target (matplotlib.mathtext.BakomaFonts attribute), 1262
TempCache (class in matplotlib.font_manager), 1223
test_lines_dists() (in module mpl_toolkits.mplot3d.proj3d), 1877
test_proj() (in module mpl_toolkits.mplot3d.proj3d), 1877
test_proj_draw_axes() (in module mpl_toolkits.mplot3d.proj3d), 1877
test_proj_make_M() (in module mpl_toolkits.mplot3d.proj3d), 1877
text_rot() (in module mpl_toolkits.mplot3d.proj3d), 1877
test_world() (in module mpl_toolkits.mplot3d.proj3d), 1877
texname (matplotlib.dviread.DviFont attribute), 1181, 1182
Text (class in matplotlib.text), 1624
text() (in module matplotlib.pyplot), 1565
text() (matplotlib.axes.Axes method), 717
text() (matplotlib.figure.Figure method), 1203
text() (mpl_toolkits.mplot3d.Axes3D method), 1844
text() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 1866
text2D() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 1866
Text3D (class in mpl_toolkits.mplot3d.art3d), 1874
text3D() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 1867
text_2d_to_3d() (in module mpl_toolkits.mplot3d.art3d), 1875
TextArea (class in matplotlib.offsetbox), 1321
TextWithDash (class in matplotlib.text), 1631
Tfm (class in matplotlib.dviread), 1183
ThetaFormatter (class in matplotlib.projections.polar), 1386
thetagrids() (in module matplotlib.pyplot), 1566
thumbnail() (module matplotlib.image), 1235
Tick (class in matplotlib.axis), 807
tick_bottom() (matplotlib.axis.XAxis method), 805
tick_left() (matplotlib.axis.YAxis method), 804
tick_params() (in module matplotlib.pyplot), 1567
tick_params() (matplotlib.axes.Axes method), 762
tick_params() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 1867
tick_right() (matplotlib.axis.YAxis method), 804
tick_top() (matplotlib.axis.XAxis method), 805
tick_update_position() (in module mpl_toolkits.mplot3d.axes3d), 1871
tick_values() (matplotlib.dates.AutoDateLocator method), 1173
tick_values() (matplotlib.dates.MicrosecondLocator method), 1175
tick_values() (matplotlib.dates.RRuleLocator method), 1172
tick_values() (matplotlib.dates.YearLocator method), 1173
tick_values() (matplotlib.ticker.AutoMinorLocator method), 1645

Index 3129
tick_values() (matplotlib.ticker.FixedLocator method), 1643

tick_values() (matplotlib.ticker.IndexLocator method), 1643

tick_values() (matplotlib.ticker.LinearLocator method), 1643

tick_values() (matplotlib.ticker.Locator method), 1642

tick_values() (matplotlib.ticker.LogLocator method), 1644

tick_values() (matplotlib.ticker.MaxNLocator method), 1645

tick_values() (matplotlib.ticker.MultipleLocator method), 1644

tick_values() (matplotlib.ticker.NullLocator method), 1643

tick_values() (matplotlib.ticker.SymmetricalLogLocator method), 1646

Ticker (class in matplotlib.axis), 794

TickHelper (class in matplotlib.ticker), 1637

ticklabel_format() (in module matplotlib.pyplot), 1568

ticklabel_format() (matplotlib.axes.Axes method), 762

ticklabel_format() (mpl_toolkits.mplot3d.axes3d.Axes3d method), 1868

TickLabels (class in mpl_toolkits.axes_grid.axis_artist), 1804

Ticks (class in mpl_toolkits.axes_grid.axis_artist), 1803

TIFF, 3018

tight_layout() (in module matplotlib.pyplot), 1569

tight_layout() (matplotlib.figure.Figure method), 1204

tight_layout() (matplotlib.gridspec.GridSpec method), 1227

TimedAnimation (class in matplotlib.animation), 525

TimerBase (class in matplotlib.backend_bases), 931

title() (in module matplotlib.pyplot), 1569

Tk, 3018

to_filehandle() (in module matplotlib.cbook), 966

to_hex() (in module matplotlib.colors), 1163

to_html5_video() (matplotlib.animation.Animation method), 495

to_mask() (matplotlib.mathtext.MathTextParser method), 1268

to_png() (matplotlib.mathtext.MathTextParser method), 1268

to_polygons() (matplotlib.path.Path method), 1372

to_rgb() (in module matplotlib.colors), 1163

to_rgba() (in module matplotlib.colors), 1163

to_rgba() (matplotlib.cm.ScalarMappable method), 972

to_rgba() (matplotlib.collections.AsteriskPolygonCollection method), 987

to_rgba() (matplotlib.collections.BrokenBarHCollection method), 999

to_rgba() (matplotlib.collections.CircleCollection method), 1011

to_rgba() (matplotlib.collections.Collection method), 1023

to_rgba() (matplotlib.collections.EllipseCollection method), 1035

to_rgba() (matplotlib.collections.EventCollection method), 1048

to_rgba() (matplotlib.collections.LineCollection method), 1060

to_rgba() (matplotlib.collections.PatchCollection method), 1072

to_rgba() (matplotlib.collections.PathCollection method), 1084

to_rgba() (matplotlib.collections.PolyCollection method), 1096

to_rgba() (matplotlib.collections.QuadMesh method), 1108

to_rgba() (matplotlib.collections.RegularPolyCollection method), 1120

to_rgba() (matplotlib.collections.StarPolygonCollection method), 1132

to_rgba() (matplotlib.collections.TriMesh method), 1144

to_rgba() (matplotlib.mathtext.MathTextParser method), 1268

to_rgba_array() (in module matplotlib.colors), 1163

to_values() (matplotlib.transforms.AffineTransform method), 1662

todore (class in matplotlib.cbook), 966

todatetime (class in matplotlib.cbook), 966

tofloat (class in matplotlib.cbook), 966

toggle() (mpl_toolkits.axes_grid.axis_artist.AxisArtist method), 1803

toggle_toolitem() (matplotlib.backend_bases.ToolContainerBase method), 933
toggled (matplotlib.backend_tools.ToolToggleBase attribute), 942
toint (class in matplotlib.cbook), 966
ToolBack (class in matplotlib.backend_tools), 938
ToolBase (class in matplotlib.backend_tools), 938
ToolContainerBase (class in matplotlib.backend_bases), 932
ToolCursorPosition (class in matplotlib.backend_tools), 940
ToolEnableAllNavigation (class in matplotlib.backend_tools), 940
ToolEnableNavigation (class in matplotlib.backend_tools), 940
ToolEvent (class in matplotlib.backend_managers), 934
ToolForward (class in matplotlib.backend_tools), 940
ToolFullScreen (class in matplotlib.backend_tools), 940
ToolGrid (class in matplotlib.backend_tools), 941
ToolHandles (class in matplotlib.widgets), 1700
ToolHome (class in matplotlib.backend_tools), 941
toolitems (matplotlib.backend_bases.NavigationToolbar2 attribute), 926
ToolManager (class in matplotlib.backend_managers), 934
toolmanager_connect() (matplotlib.backend_managers.ToolManager method), 935
toolmanager_disconnect() (matplotlib.backend_managers.ToolManager method), 936
ToolManagerMessageEvent (class in matplotlib.backend_managers), 936
ToolPan (class in matplotlib.backend_tools), 941
ToolQuit (class in matplotlib.backend_tools), 942
tools (matplotlib.backend_managers.ToolManager attribute), 936
ToolToggleBase (class in matplotlib.backend_tools), 942
ToolTriggerEvent (class in matplotlib.backend_managers), 936
ToolViewsPositions (class in matplotlib.backend_tools), 943
ToolXScale (class in matplotlib.backend_tools), 943
ToolYScale (class in matplotlib.backend_tools), 944
ToolZoom (class in matplotlib.backend_tools), 944
tostr (class in matplotlib.cbook), 967

tostr() (matplotlib.mlab.FormatFormatStr method), 1281
tostr() (matplotlib.mlab.FormatInt method), 1282
tostr() (matplotlib.mlab.FormatObj method), 1282
tostr() (matplotlib.mlab.FormatString method), 1282
toval() (matplotlib.mlab.FormatBool method), 1281
toval() (matplotlib.mlab.FormatDate method), 1281
toval() (matplotlib.mlab.FormatFloat method), 1281
toval() (matplotlib.mlab.FormatInt method), 1282
toval() (matplotlib.mlab.FormatObj method), 1282
tostr() (matplotlib.mlab.FormatFormatStr method), 1281
tostr() (matplotlib.mlab.FormatInt method), 1282
tostr() (matplotlib.mlab.FormatObj method), 1282
toolmanager_connect() (matplotlib.backend_managers.ToolManager method), 935
toolmanager_disconnect() (matplotlib.backend_managers.ToolManager method), 936
transform (class in matplotlib.transforms), 1657
transform() (in module mpl_toolkits.mplot3d.proj3d), 1877
transform() (matplotlib.transforms.AffineTransform method), 1660
transform() (matplotlib.transforms.IdentityTransform method), 1665
transform() (matplotlib.transforms.Transform method), 1658
transform() (matplotlib.type1font.Type1Font method), 1683
transform_affine() (matplotlib.transforms.Affine2DBase method), 1662
transform_affine() (matplotlib.transforms.AffineTransform method), 1660
transform_affine() (matplotlib.transforms.CompositeGenericTransform method), 1667
transform_affine() (matplotlib.transforms.IdentityTransform method), 1665
transform_affine() (matplotlib.transforms.Transform method), 1658
transform_angles() (matplotlib.transforms.Transform method), 1658
transform_bbox() (matplotlib.transforms.Transform method), 1659
transform_non_affine() (matplotlib.projections.polar.InvertedPolarTransform method), 1380
transform_non_affine() (matplotlib.projections.polar.PolarAxes.InvertedPolarTransform method), 1381
transform_non_affine() (matplotlib.projections.polar.PolarAxes.PolarTransform method), 1381
translate() (matplotlib.transforms.Affine2D method), 1664
translated() (matplotlib.transforms.BboxBase method), 1654
transmute() (matplotlib.patches.ArrowStyle.Fancy method), 1332
transmute() (matplotlib.patches.ArrowStyle.Simple method), 1333
transmute() (matplotlib.patches.ArrowStyle.Wedge method), 1333
transmute() (matplotlib.patches.BoxStyle.Circle method), 1335
transmute() (matplotlib.patches.BoxStyle.DArrow method), 1335
transmute() (matplotlib.patches.BoxStyle.LArrow method), 1335
transmute() (matplotlib.patches.BoxStyle.RArrow method), 1335
transmute() (matplotlib.patches.BoxStyle.Round method), 1335
transmute() (matplotlib.patches.BoxStyle.Round4 method), 1335
transmute() (matplotlib.patches.BoxStyle.Roundtooth method), 1336
transmute() (matplotlib.patches.BoxStyle.Sawtooth method), 1336
transmute() (matplotlib.patches.BoxStyle.Square method), 1336
TrapezoidMapTriFinder (class in matplotlib.tri), 1672
TriAnalyzer (class in matplotlib.tri), 1679
Triangulation (class in matplotlib.tri), 1671
tricontour() (in module matplotlib.pyplot), 1570
tricontour() (matplotlib.axes.Axes method), 706
tricontour() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 1868
tricontour() (in module matplotlib.pyplot), 1574
tricontour() (matplotlib.axes.Axes method), 710
tricontour() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 1868
TriFinder (class in matplotlib.tri), 1672
trigger() (matplotlib.backend_tools.AxisScaleBase method), 937
trigger() (matplotlib.backend_tools.RubberbandBase method), 938
trigger() (matplotlib.backend_tools.ToolBase method), 939
trigger() (matplotlib.backend_tools.ToolEnableAllNavigation method), 940
trigger() (matplotlib.backend_tools.ToolEnableNavigation method), 940
trigger() (matplotlib.backend_tools.ToolGrid method), 941
trigger() (matplotlib.backend_tools.ToolQuit method), 942
trigger() (matplotlib.backend_tools.ToolToggleBase method), 943
trigger() (matplotlib.backend_tools.ViewsPositionsBase method), 944
trigger() (matplotlib.backend_tools.ZoomPanBase method), 944
trigger_tool() (matplotlib.backend_bases.ToolContainerBase method), 933
trigger_tool() (matplotlib.backend_managers.ToolManager method), 936
TriInterpolator (class in matplotlib.tri), 1673
TriMesh (class in matplotlib.collections), 1133
tricontour() (in module matplotlib.pyplot), 1578
triplot() (in module matplotlib.pyplot), 1581
triplot() (matplotlib.axes.Axes method), 704
TriRefiner (class in matplotlib.tri), 1676
TrueTypeFonts (class in matplotlib.mathtext), 1276
tfdict_to_fnames() (in module matplotlib.font_manager), 1224
ttfFontProperty() (in module matplotlib.font_manager), 1224
tunit_cube() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 1869
tunit_edges() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 1869
twinx() (in module matplotlib.pyplot), 1583
twinx() (matplotlib.axes.Axes method), 766
twiny() (in module matplotlib.pyplot), 1583
twiny() (matplotlib.axes.Axes method), 766
Type1Font (class in matplotlib.type1font), 1683
U
unichr_safe() (in module matplotlib.mathtext), 1277
unicode_safe() (in module matplotlib.cbook), 967
UnicodeFonts (class in matplotlib.mathtext), 1276
UniformTriRefiner (class in matplotlib.tri), 1677
uninstall_repl_displayhook() (in module matplotlib.pyplot), 1583
union() (matplotlib.transforms.BboxBase static method), 1654
unique() (in module matplotlib.cbook), 967
unit() (matplotlib.transforms.Bbox static method), 1655
unit_bbox() (in module mpl_toolkits.mplot3d.axes3d), 1869
unit_circle() (matplotlib.path.Path class method), 1373
unit_circle_righthalf() (matplotlib.path.Path class method), 1373
unit_cube() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 1869
unit_rectangle() (matplotlib.path.Path class method), 1373
unit_regular_asterisk() (matplotlib.path.Path class method), 1373
unit_regular_polygon() (matplotlib.path.Path class method), 1373
unit_regular_star() (matplotlib.path.Path class method), 1373
unknown_symbol() (matplotlib.mathtext.Parser method), 1273
unmasked_index_ranges() (in module matplotlib.cbook), 967
update() (matplotlib.artist.Artist method), 541
update() (matplotlib.axes.Axes method), 778
update() (matplotlib.axis.Axis method), 889
update() (matplotlib.axis.Tick method), 849
update() (matplotlib.axis.XAxis method), 900
update() (matplotlib.axis.XTick method), 860
update() (matplotlib.axis.YAxis method), 911
update() (matplotlib.axis.YTick method), 872
update() (matplotlib.backend_bases.NavigationToolbar2 method), 926
update() (matplotlib.collections.AsteriskPolygonCollection method), 987
update() (matplotlib.collections.BrokenBarHCollection method), 999
update() (matplotlib.collections.CircleCollection method), 1011
update() (matplotlib.collections.Collection method), 1023
update() (matplotlib.collections.EllipseCollection method), 1035
update() (matplotlib.collections.EventCollection method), 1048
update() (matplotlib.collections.LineCollection method), 1061
update() (matplotlib.collections.PatchCollection method), 1072
update() (matplotlib.collections.PolyCollection method), 1084
update() (matplotlib.collections.QuadMesh method), 1096
update() (matplotlib.collections.RegularPolyCollection method), 1108
update() (matplotlib.collections.RegularPolygonCollection method), 1120
update() (matplotlib.collections.StarPolygonCollection method), 1133
update() (matplotlib.collections.TriMesh method), 1144
update() (matplotlib.collections.TriMesh method), 1144
update() (matplotlib.collections.TriMesh method), 1144
update_bbox_position_size() (matplotlib.text.Text method), 1631
update_bruteforce() (matplotlib.colorbar.Colorbar method), 1145
update_coords() (matplotlib.text.TextWithDash method), 1634
update_datalim() (matplotlib.axes.Axes method), 741
update_datalim() (mpl_toolkits.mplot3d.axes3dAxes3D method), 1869
update_datalim_bounds() (matplotlib.axes.Axes method), 741
update_datalim_numerix() (matplotlib.axes.Axes method), 741
update_default_handler_map() (matplotlib.legend.Legend class method), 1241
update_default_handler_map() (matplotlib.legend.Legend class method), 1241
update_from() (matplotlib.artist.Artist method), 542
update_from() (matplotlib.axes.Axes method), 778
update_from() (matplotlib.axis.Axis method), 889
update_fonts() (matplotlib.font_manager.FontManager method), 1221
update_frame() (matplotlib.offsetbox.AnchoredOffsetbox method), 1314
update_frame() (matplotlib.offsetbox.PaddedBox method), 1321
update_from() (matplotlib.artist.Artist method), 542
update_from() (matplotlib.axes.Axes method), 778
update_from() (matplotlib.axis.Axis method), 889
update_from() (matplotlib.axis.Axis method), 889
update_from() (matplotlib.axis.Axis method), 889
update_from() (matplotlib.axis.Axis method), 889
update_from() (matplotlib.axis.Axis method), 889
method), 1000
update_scalarmappable() (matplotlib.collections.CircleCollection
method), 1011
update_scalarmappable() (matplotlib.collections.Collection
method), 1023
update_scalarmappable() (matplotlib.collections.EllipseCollection
method), 1035
update_scalarmappable() (matplotlib.collections.EventCollection
method), 1048
update_scalarmappable() (matplotlib.collections.LineCollection
method), 1061
update_scalarmappable() (matplotlib.collections.PatchCollection
method), 1072
update_scalarmappable() (matplotlib.collections.PathCollection
method), 1084
update_scalarmappable() (matplotlib.collections.PolyCollection
method), 1096
update_scalarmappable() (matplotlib.collections.QuadMesh
method), 1108
update_scalarmappable() (matplotlib.collections.RegularPolyCollection
method), 1121
update_scalarmappable() (matplotlib.collections.StarPolygonCollection
method), 1133
update_scalarmappable() (matplotlib.collections.TriMesh
method), 1144
update_ticks() (matplotlib.colorbar.ColorbarBase
method), 1147
update_units() (matplotlib.axis.Axes.Axes method), 803
update_units() (matplotlib.axis.XAxis.Axes method), 832
update_units() (matplotlib.axis.YAxis.Axes method), 822
update_view() (matplotlib.backend_tools.ToolViewsPositions
method), 943
use() (in module matplotlib), 483
use() (in module matplotlib.style), 1619
use_cmex (matplotlib.mathtext.DejaVuFonts attribute), 1263
use_cmex (matplotlib.mathtext.StixFonts attribute), 1275
use_cmex (matplotlib.mathtext.UnicodeFonts attribute), 1276
use_sticky_edges (matplotlib.axes.Axes.Axes attribute), 751
useLocale (matplotlib.ticker.ScalarFormatter attribute), 1640
useOffset (matplotlib.ticker.ScalarFormatter attribute), 1640

V
validate_animation_writer_path() (in module matplotlib.rcsetup), 1594
validate_any() (in module matplotlib.rcsetup), 1594
validate_anylist() (in module matplotlib.rcsetup), 1594
validate_aspect() (in module matplotlib.rcsetup), 1594
validate_axisbelow() (in module matplotlib.rcsetup), 1594
validate_axiscenter() (in module matplotlib.rcsetup), 1594
validate_backend() (in module matplotlib.rcsetup), 1594
validate_bbox() (in module matplotlib.rcsetup), 1594
validate_capstylelist() (in module matplotlib.rcsetup), 1594
validate_color() (in module matplotlib.rcsetup), 1594
validate_color_for_prop_cycle() (in module matplotlib.rcsetup), 1594
validate_color_or_auto() (in module matplotlib.rcsetup), 1594
validate_color_or_inherit() (in module matplotlib.rcsetup), 1594
validate_colorlist() (in module matplotlib.rcsetup), 1594
validate_cycler() (in module matplotlib.rcsetup), 1594
validate_dashlist() (in module matplotlib.rcsetup), 1594
validate_dpi() (in module matplotlib.rcsetup), 1594
validate_fillstylelist() (in module matplotlib.rcsetup), 1595
validate_float() (in module matplotlib.rcsetup), 1595
validate_float_or_None() (in module matplotlib.rcsetup), 1595
validate_floatlist() (in module matplotlib.rcsetup), 1595
validate_font_properties() (in module matplotlib.rcsetup), 1595
validate_fontsize() (in module matplotlib.rcsetup), 1595
validate_fontszelist() (in module matplotlib.rcsetup), 1595
validate_fonttype() (in module matplotlib.rcsetup), 1595
validate_hatch() (in module matplotlib.rcsetup), 1595
validate_hatchlist() (in module matplotlib.rcsetup), 1595
validate_hatching() (in module matplotlib.rcsetup), 1596
validate_hist_bins() (in module matplotlib.rcsetup), 1596
validate_int() (in module matplotlib.rcsetup), 1596
validate_int_or_None() (in module matplotlib.rcsetup), 1596
validate_joinstylelist() (in module matplotlib.rcsetup), 1596
validate_maskedarray() (in module matplotlib.rcsetup), 1596
validate_negative_linestyle_legacy() (in module matplotlib.rcsetup), 1596
validate_nseq_float (class in matplotlib.rcsetup), 1596
validate_nseq_int (class in matplotlib.rcsetup), 1596
validate_path_exists() (in module matplotlib.rcsetup), 1596
validate_whiskers() (in module matplotlib.rcsetup), 1596
ValidateInStrings (class in matplotlib.rcsetup), 1593
ValidateInterval (class in matplotlib.rcsetup), 1593
validCap (matplotlib.lines.Line2D attribute), 1253
validCap (matplotlib.patches.Patch attribute), 1355
validJoin (matplotlib.lines.Line2D attribute), 1253
validJoin (matplotlib.patches.Patch attribute), 1355
value_escape() (in module matplotlib.fontconfig_pattern), 1225
value_unescape() (in module matplotlib.fontconfig_pattern), 1225
Vbox (class in matplotlib.mathtext), 1276
VCentered (class in matplotlib.mathtext), 1276
tvec_pad_ones() (in module mpl_toolkits.mplot3d.proj3d), 1877
vector graphics, 3018
vector_lengths() (in module matplotlib.mlab), 1311
Verbatim (class in matplotlib.backends.backend_pdf), 951
VertexSelector (class in matplotlib.lines), 1253
vertices (matplotlib.path.Path attribute), 1373
Vf (class in matplotlib.dviread), 1183
view_init() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 1869
view_limits() (matplotlib.ticker.LinearLocator method), 1643
view_limits() (matplotlib.ticker.Locator method), 1642
view_limits() (matplotlib.ticker.LogLocator method), 1644
view_limits() (matplotlib.ticker.MaxNLocator method), 1645
view_limits() (matplotlib.ticker.SymmetricalLogLocator method), 1644
view_transformation() (in module mpl_toolkits.mplot3d.proj3d), 1877
viewlim_to_dt() (matplotlib.dates.DateLocator method), 1171
ViewsPositionsBase (class in matplotlib.backend_tools), 944
violin() (matplotlib.axes.Axes method), 643
violin_stats() (in module matplotlib.cbook), 967
violinplot() (in module matplotlib.pyplot), 1583
violinspect() (in module matplotlib.pyplot), 1583
Index 3137
violinplot() (matplotlib.axes.Axes method), 641
viridis() (in module matplotlib.pyplot), 1585
vlines() (in module matplotlib.pyplot), 1585
vlines() (matplotlib.axes.Axes method), 597
Vlist (class in matplotlib.mathtext), 1276
vlist_out() (matplotlib.mathtext.Ship method), 1275
volume_overlay() (in module matplotlib.finance), 1216
volume_overlay2() (in module matplotlib.finance), 1216
volume_overlay3() (in module matplotlib.finance), 1217
vpack() (matplotlib.mathtext.Vlist method), 1276
VPacker (class in matplotlib.offsetbox), 1323
Vrule (class in matplotlib.mathtext), 1277

W

waitforbuttonpress() (in module matplotlib.pyplot), 1586
waitforbuttonpress() (matplotlib.figure.Figure method), 1204
warn_deprecated() (in module matplotlib.cbook), 968
Wedge (class in matplotlib.patches), 1362
wedge() (matplotlib.path.Path class method), 1373
WeekdayLocator (class in matplotlib.dates), 1173
weeks (matplotlib.dates.relativedelta attribute), 1178
weeks() (in module matplotlib.dates), 1179
weight_as_number() (in module matplotlib.font_manager), 1224
Widget (class in matplotlib.widgets), 1701
width (matplotlib.dviread.Tfm attribute), 1183
width (matplotlib.transforms.BboxBase attribute), 1654
widths (matplotlib.dviread.DviFont attribute), 1182
win32FontDirectory() (in module matplotlib.font_manager), 1224
win32InstalledFonts() (in module matplotlib.font_manager), 1225
window_hanning() (in module matplotlib.mlab), 1311
window_none() (in module matplotlib.mlab), 1311
winter() (in module matplotlib.pyplot), 1586
withSimplePatchShadow (class in matplotlib.pathEffects), 1377
withStroke (class in matplotlib.pathEffects), 1378
world_transformation() (in module mpl_toolkits.mplot3d.proj3d), 1877
wrap() (in module matplotlib.cbook), 969
write() (matplotlib.backends.backend_pdf.Stream method), 951
writeInfoDict() (matplotlib.backends.backend_pdf.PdfFile method), 949
writeTrailer() (matplotlib.backends.backend_pdf.PdfFile method), 949
writeXref() (matplotlib.backends.backend_pdf.PdfFile method), 949
wxpython, 3018
wxWidgets, 3018

X

x (matplotlib.backend_bases.LocationEvent attribute), 924
x (matplotlib.backend_bases.MouseEvent attribute), 924
x (matplotlib.widgets.ToolHandles attribute), 1701
x0 (matplotlib.transforms.BboxBase attribute), 1654
x1 (matplotlib.transforms.BboxBase attribute), 1654
XAxis (class in matplotlib.axis), 794
XAxis (class in mpl_toolkits.mplot3d.axis3d), 1870
xaxis_date() (matplotlib.axes.Axes method), 756
xaxis_inverted() (matplotlib.axes.Axes method), 738
xcorr() (in module matplotlib.pyplot), 1586
xcorr() (matplotlib.axes.Axes method), 635
xdata (matplotlib.backend_bases.LocationEvent attribute), 924
xdata (matplotlib.backend_bases.MouseEvent attribute), 925
xkcd() (in module matplotlib.pyplot), 1587
xlabel() (in module matplotlib.pyplot), 1588
xlat() (matplotlib.cbook.Xlator method), 957
Xlator (class in matplotlib.cbook), 957
xlim() (in module matplotlib.pyplot), 1588
xmax (matplotlib.transforms.BboxBase attribute), 1654
xmin (matplotlib.transforms.BboxBase attribute), 1654
xscale() (in module matplotlib.pyplot), 1588
XTick (class in matplotlib.axis), 808
xticks() (in module matplotlib.pyplot), 1589
xy (matplotlib.patches.Polygon attribute), 1358
xy (matplotlib.patches.Rectangle attribute), 1360
xy (matplotlib.patches.RegularPolygon attribute), 1361
xy() (matplotlib.cbook.MemoryMonitor method), 956
xyann (matplotlib.offsetbox.AnnotationBbox attribute), 1316
xyann (matplotlib.text.Annotation attribute), 1624
Y
y (matplotlib.backend_bases.LocationEvent attribute), 924
y (matplotlib.backend_bases.MouseEvent attribute), 925
y (matplotlib.widgets.ToolHandles attribute), 1701
y0 (matplotlib.transforms.BboxBase attribute), 1654
y1 (matplotlib.transforms.BboxBase attribute), 1654
YAArrow (class in matplotlib.patches), 1364
YAxis (class in matplotlib.axis), 794
YAxis (class in mpl_toolkits.mplot3d.axis3d), 1870
yaxis_date() (matplotlib.axes.Axes method), 756
yaxis_inverted() (matplotlib.axes.Axes method), 738
ydata (matplotlib.backend_bases.LocationEvent attribute), 924
ydata (matplotlib.backend_bases.MouseEvent attribute), 925
YearLocator (class in matplotlib.dates), 1173
ylabel() (in module matplotlib.pyplot), 1590
ylim() (in module matplotlib.pyplot), 1590
ymax (matplotlib.transforms.BboxBase attribute), 1654
ymin (matplotlib.transforms.BboxBase attribute), 1654
yscale() (in module matplotlib.pyplot), 1590
YTick (class in matplotlib.axis), 808
yticks() (in module matplotlib.pyplot), 1591
Z
zalpha() (in module mpl_toolkits.mplot3d.art3d), 1876
ZAxis (class in mpl_toolkits.mplot3d.axis3d), 1871
zaxis_date() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 1869
zaxis_inverted() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 1869
zoom() (matplotlib.axis.Axes method), 803
zoom() (matplotlib.axis.XAxis method), 832
zoom() (matplotlib.axis.YAxis method), 823
zoom() (matplotlib.backend_bases.NavigationToolbar2 method), 926
zoom() (matplotlib.ticker.Locator method), 1642
zoomed_inset_axes() (in module mpl_toolkits.axes_grid1.inset_locator), 1820
ZoomPanBase (class in matplotlib.backend_tools), 944
zorder (matplotlib.artist.Artist attribute), 545
zorder (matplotlib.axes.Axes attribute), 889
zorder (matplotlib.axis.Tick attribute), 849
zorder (matplotlib.axis.XAxis attribute), 900
zorder (matplotlib.axis.XTick attribute), 861
zorder (matplotlib.axis.YAxis attribute), 911
zorder (matplotlib.axis.YTick attribute), 872
zorder (matplotlib.collections.AsteriskPolygonCollection attribute), 988
zorder (matplotlib.collections.BrokenBarHCollection attribute), 1000
zorder (matplotlib.collections.CircleCollection attribute), 1011
zorder (matplotlib.collections.Collection attribute), 1023
zorder (matplotlib.collections.EllipseCollection attribute), 1035
zorder (matplotlib.collections.EventCollection attribute), 1048
zorder (matplotlib.collections.LineCollection attribute), 1061
zorder (matplotlib.collections.PatchCollection attribute), 1072
zorder (matplotlib.collections.PathCollection attribute), 1084
zorder (matplotlib.collections.PolyCollection attribute), 1096
zorder (matplotlib.collections.QuadMesh attribute), 1108
zorder (matplotlib.collections.RegularPolyCollection attribute), 1121
zorder (matplotlib.collections.StarPolygonCollection attribute), 1133
zorder (matplotlib.collections.TriMesh attribute), 1144
zorder (matplotlib.image.FigureImage attribute), 1232
zorder (matplotlib.legend.Legend attribute), 1241
zorder (matplotlib.lines.Line2D attribute), 1253
zorder (matplotlib.offsetbox.AnchoredOffsetbox attribute), 1314
zorder (matplotlib.offsetbox.AnnotationBbox attribute), 1314
attribute), 1316
zorder (matplotlib.patches.Patch attribute), 1355
zorder (matplotlib.text.Text attribute), 1631
ZORDER (mpl_toolkits.axes_grid.axis_artist.AxisArtist
attribute), 1802