Matplotlib

Release 1.3.x

John Hunter, Darren Dale, Eric Firing, Michael Droettboon

April 01, 2013

CONTENTS

Part I

User's Guide

CHAPTER

ONE

INTRODUCTION

matplotlib is a library for making 2D plots of arrays in Python. Although it has its origins in emulating the MATLAB® [1] 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 and 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

[1] MATLAB is a registered trademark of The MathWorks, Inc.

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 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.

CHAPTER

TWO

INSTALLING

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, the easiest thing for you to do is use one of the pre-packaged python distributions that already provide matplotlib built-in. The Enthought Python Distribution (EPD) for Windows, OS X or Redhat is an excellent choice that "just works" out of the box. Another excellent alternative for Windows users is Python (x, y) which tends to be updated a bit more frequently. Both of these packages include matplotlib and pylab, and *lots* of other useful tools. matplotlib is also packaged for almost every major Linux distribution. So if you are on Linux, your package manager will probably provide matplotlib prebuilt.

2.1 Manually installing pre-built packages

2.1.1 General instructions

For some people, the prepackaged pythons discussed above are not an option. That's OK, it's usually pretty easy to get a custom install working. You will first need to find out if you have python installed on your machine, and if not, install it. The official python builds are available for download here, but OS X users please read *Which python for OS X?*.

Once you have python up and running, you will need to install numpy. numpy provides high-performance array data structures and mathematical functions, and is a requirement for matplotlib. You can test your progress:

```
>>> import numpy
>>> print numpy.__version__
```

matplotlib requires numpy version 1.4 or later. Although it is not a requirement to use matplotlib, we strongly encourage you to install ipython, which is an interactive shell for python that is matplotlib-aware.

Next, we need to get matplotlib installed. We provide prebuilt binaries for OS X and Windows on the matplotlib download page. Click on the latest release of the "matplotlib" package, choose your python version (2.6, 2.7 or 3.2) and your platform (macosx or win32). If you have any problems, please check the *Installation*, search using Google, and/or post a question the mailing list.

If you are on Debian/Ubuntu linux, it suffices to do:

```
> sudo apt-get install python-matplotlib
```

Instructions for installing our OSX binaries are found in the FAQ *Installing OSX binaries*.

Once you have ipython, numpy and matplotlib installed, you can use ipython's "pylab" mode to have a MATLAB-like environment that automatically handles most of the configuration details for you, so you can get up and running quickly:

```
johnh@flag:~> ipython -pylab
Python 2.4.5 (#4, Apr 12 2008, 09:09:16)
IPython 0.9.0 -- An enhanced Interactive Python.

  Welcome to pylab, a matplotlib-based Python environment.
  For more information, type 'help(pylab)'.

In [1]: x = randn(10000)

In [2]: hist(x, 100)
```

Note that when testing matplotlib installations from the interactive python console, there are some issues relating to user interface toolkits and interactive settings that are discussed in *Using matplotlib in a python shell*.

2.1.2 Installing on Windows

If you don't already have python installed, you may want to consider using the Enthought edition of python, which has scipy, numpy, and wxpython, plus many other useful packages, preinstalled - Enthought Python. With the Enthought edition of python + matplotlib installer, the following backends should work out of the box: agg, wx, wxagg, tkagg, ps, pdf and svg.

For standard python installations, you will also need to install numpy in addition to the matplotlib installer. On some systems you will also need to download msvcp71.dll library, which you can download from http://www.dll-files.com/dllindex/dll-files.shtml?msvcp71 or other sites. You will need to unzip the archive and drag the dll into `c:windowssystem32`.

All of the GUI backends run on Windows, but TkAgg is probably the best for interactive use from the standard python shell or ipython. The Windows installer (`*.exe`) on the download page contains all the code you need to get up and running. However, there are many examples that are not included in the Windows installer. If you want to try the many demos that come in the matplotlib source distribution, download the zip file and look in the `examples` subdirectory.

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 download page, or if you want to develop matplotlib or just need the latest bugfixed version, grab the latest git version *Source install from git*.

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.

2.3 Build requirements

These are external packages which you will need to install before installing matplotlib. Windows users only need the first two (python and numpy) since the others are built into the matplotlib Windows installers available for download at `the download page <https://github.com/matplotlib/matplotlib/downloads>_`. If you are building on OSX, see *Building on OSX*. 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.

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 the install the build dependencies, which will make building from source easier.

***python* 2.6, 2.7, 3.1 or 3.2** Download python.

***numpy* 1.4 (or later)** array support for python (download)

libpng 1.2 (or later) library for loading and saving *PNG* files (download). libpng requires zlib. If you are a Windows user, you can ignore this because we build support into the matplotlib single-click installer

***freetype* 1.4 (or later)** library for reading true type font files. If you are a windows user, you can ignore this since we build support into the matplotlib single click installer.

Optional

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.

***tk* 8.3 or later** The TCL/Tk widgets library used by the TkAgg backend

***pyqt* 3.1 or later** The Qt3 widgets library python wrappers for the QtAgg backend

***pyqt* 4.0 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

***pyfltk* 1.0 or later** The python wrappers of the FLTK widgets library for use with FLTKAgg

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.

***pytz* 2007g or later** timezone handling for python datetime objects. By default, matplotlib will install pytz if it isn't already installed on your system. To override the default, use `setup.cfg` to force or prevent installation of pytz.

***dateutil* 1.1 or later** provides extensions to python datetime handling. By default, matplotlib will install dateutil if it isn't already installed on your system. To override the default, use `setup.cfg` to force or prevent installation of dateutil.

2.4 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`.

CHAPTER

THREE

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: eg, create a figure, create a plotting area in a figure, plot some lines in a plotting area, decorate the plot with labels, etc.... matplotlib.pyplot is stateful, in that it keeps track of the current figure and plotting area, and the plotting functions are directed to the current axes

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:

```
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

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.

3.1 Controlling line properties

Lines have many attributes that you can set: linewidth, dash style, antialiased, etc; see `matplotlib.lines.Line2D`. There are several ways to set line properties

- Use keyword args:

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

- Use the setter methods of the `Line2D` instance. `plot` returns a list of lines; eg `line1, line2 = plot(x1,y1,x2,x2)`. Below I have only one line so it is a list of length 1. I use tuple unpacking in the `line, = plot(x, y, 'o')` to get the first element of the list:

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

- 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:

  ```
  lines = plt.plot(x1, y1, x2, y2)
  # use keyword args
  plt.setp(lines, color='r', linewidth=2.0)
  # or MATLAB style string value pairs
  plt.setp(lines, 'color', 'r', 'linewidth', 2.0)
  ```

Here are the available `Line2D` properties.

Property	Value Type
alpha	float
animated	[True | False]
antialiased or aa	[True | False]
clip_box	a matplotlib.transform.Bbox instance
clip_on	[True | False]
clip_path	a Path instance and a Transform instance, a Patch
color or c	any matplotlib color
contains	the hit testing function
dash_capstyle	['butt' | 'round' | 'projecting']
dash_joinstyle	['miter' | 'round' | 'bevel']
dashes	sequence of on/off ink in points
data	(np.array xdata, np.array ydata)
figure	a matplotlib.figure.Figure instance
label	any string
linestyle or ls	['-' | '--' | '-.' | ':' | 'steps' | ...]

Continued on next page

Table 3.1 – continued from previous page

Property	Value Type
linewidth or lw	float value in points
lod	[True | False]
marker	['+' | ',' | '.' | '1' | '2' | '3' | '4'
markeredgecolor or mec	any matplotlib color
markeredgewidth or mew	float value in points
markerfacecolor or mfc	any matplotlib color
markersize or ms	float
markevery	None | integer | (startind, stride)
picker	used in interactive line selection
pickradius	the line pick selection radius
solid_capstyle	['butt' | 'round' | 'projecting']
solid_joinstyle	['miter' | 'round' | 'bevel']
transform	a matplotlib.transforms.Transform instance
visible	[True | False]
xdata	np.array
ydata	np.array
zorder	any number

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]
  ...snip
```

3.2 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 (`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.

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 an 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, ie, 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-axes_demo* for an example of placing axes manually and *pylab_examples-line_styles* for an example with lots-o-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:

```
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
plt.subplot(211)             # make subplot(211) in figure1 current
plt.title('Easy as 1,2,3')   # subplot 211 title
```

You can clear the current figure with `clf()` and the current axes with `cla()`. If you find this statefulness, annoying, 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 a long sequence 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.

3.3 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 with lines above, you can customize the properties by passing keyword arguments into the text functions or using `setp()`:

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

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

3.3.1 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:

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

The `r` preceeding the title string is important – it signifies that the string is a *raw* string and not to treate backslashes and 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*.

3.3.2 Annotating text

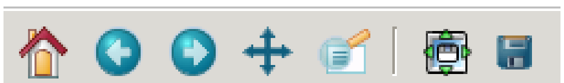
The uses of the basic `text()` command above place text at an arbitrary position on the Axes. A common use case of text is to annotate some feature of the plot, and the `annotate()` method provides helper functionality to make annotations easy. In an annotation, there are two points to consider: the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

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 *Annotating text* and *Annotating Axes* for details. More examples can be found in *pylab_examples-annotation_demo*.

CHAPTER

FOUR

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

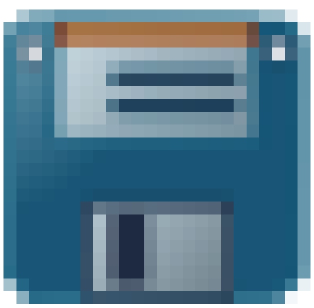
The Forward and Back buttons These are akin to the web browser forward and back buttons. They are used to navigate back and forth between previously defined views. They have no meaning unless you have already navigated somewhere else using the pan and zoom buttons. This is analogous to trying to click `Back` on your web browser before visiting a new page –nothing happens. `Home` always takes you to the first, default view of your data. For `Home`, `Forward` and `Back`, think web browser where data views are web pages. Use the pan and zoom to rectangle to define new views.

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 proportionate to the rightward movement and zoomed out proportionate to the leftward movement. Ditto for the y axis and up/down motions. The point under your mouse when you begin the zoom remains stationary, allowing you to zoom to an arbitrary point in the figure. 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 and 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 parameters of the subplot: the left, right, top, bottom, space between the rows and 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`.

4.1 Navigation Keyboard Shortcuts

The following table holds all the default keys, which can be overwritten by use of your matplotlibrc (#keymap.*).

Command	Keyboard Shortcut(s)
Home/Reset	**h** or **r** or **home**
Back	**c** or **left arrow** or **backspace**
Forward	**v** or **right arrow**
Pan/Zoom	**p**
Zoom-to-rect	**o**
Save	**ctrl** + **s**
Toggle fullscreen	**ctrl** + **f**
Close plot	**ctrl** + **w**
Constrain pan/zoom to x axis	hold **x** when panning/zooming with mouse
Constrain pan/zoom to y axis	hold **y** when panning/zooming with mouse
Preserve aspect ratio	hold **CONTROL** when panning/zooming with mouse
Toggle grid	**g** when mouse is over an axes
Toggle x axis scale (log/linear)	**L** or **k** when mouse is over an axes
Toggle y axis scale (log/linear)	**l** when mouse is over an axes

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:

```
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)
toolbar = NavigationToolbar(canvas, win)
vbox.pack_start(toolbar, False, False)

win.show_all()
gtk.main()
```

CHAPTER

FIVE

CUSTOMIZING MATPLOTLIB

5.1 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 three 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. `.matplotlib/matplotlibrc`, for the user's default customizations. See *.matplotlib directory location*.
3. `INSTALL/matplotlib/mpl-data/matplotlibrc`, where `INSTALL` is something like `/usr/lib/python2.5/site-packages` on Linux, and maybe `C:\Python25\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 `.matplotlib` directory.

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

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

See below for a sample *matplotlibrc file*.

5.2 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:

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

There `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.

5.2.1 A sample matplotlibrc file

```
### 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 HOME/.matplotlib/matplotlibrc (unix/linux
# like systems) and C:\Documents and Settings\yourname\.matplotlib
# (win32 systems).
#
# 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 or #ff00ff - a scalar
# grayscale intensity such as 0.75 - a legal html color name, eg red,
```

```
# blue, darkslategray

#### CONFIGURATION BEGINS HERE

# the default backend; one of GTK GTKAgg GTKCairo GTK3Agg GTK3Cairo
# CocoaAgg FltkAgg MacOSX QtAgg Qt4Agg 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      : Agg

# 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.

# 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 | classic | toolbar2
#timezone     : UTC        # a pytz timezone string, eg 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   : 1.0     # line width in points
#lines.linestyle   : -       # solid line
#lines.color       : blue    # has no affect on plot(); see axes.color_cycle
#lines.marker      : None    # the default marker
#lines.markeredgewidth  : 0.5     # 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 antialised (no jaggies)

### PATCHES
# Patches are graphical objects that fill 2D space, like polygons or
# circles.  See
# http://matplotlib.org/api/artist_api.html#module-matplotlib.patches
# information on patch properties
#patch.linewidth        : 1.0     # edge width in points
#patch.facecolor        : blue
#patch.edgecolor        : black
#patch.antialiased      : True    # render patches in antialised (no jaggies)

### 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.
#
# 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.
```

```
# 12pt 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           : 12.0
#font.serif          : Bitstream Vera Serif, New Century Schoolbook, Century Schoolbook L, Utop
#font.sans-serif     : Bitstream Vera Sans, Lucida Grande, Verdana, Geneva, Lucid, Arial, Helve
#font.cursive        : Apple Chancery, Textile, Zapf Chancery, Sand, cursive
#font.fantasy        : Comic Sans MS, Chicago, Charcoal, Impact, Western, fantasy
#font.monospace      : Bitstream Vera Sans Mono, Andale Mono, Nimbus Mono L, Courier New, Couri

### TEXT
# text properties used by text.Text.  See
# http://matplotlib.org/api/artist_api.html#module-matplotlib.text for more
# information on text properties

#text.color          : black

### LaTeX customizations. See http://www.scipy.org/Wiki/Cookbook/Matplotlib/UsingTex
#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
#text.latex.unicode : False # use "ucs" and "inputenc" LaTeX packages for handling
                            # unicode strings.
#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

#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

#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 : cm # Should be '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 fontsizes for ticklabels, and so on.  See
# http://matplotlib.org/api/axes_api.html#module-matplotlib.axes
#axes.hold           : True    # whether to clear the axes by default on
#axes.facecolor      : white   # axes background color
#axes.edgecolor      : black   # axes edge color
#axes.linewidth      : 1.0     # edge linewidth
#axes.grid           : False   # display grid or not
#axes.titlesize      : large   # fontsize of the axes title
#axes.labelsize      : medium  # fontsize of the x any y labels
#axes.labelweight    : normal  # weight of the x and y labels
#axes.labelcolor     : black
#axes.axisbelow      : False   # whether axis gridlines and ticks are below
                              # the axes elements (lines, text, etc)
#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.unicode_minus  : True    # use unicode for the minus symbol
                               # rather than hyphen.  See
                               # http://en.wikipedia.org/wiki/Plus_and_minus_signs#Character_co
#axes.color_cycle    : b, g, r, c, m, y, k  # color cycle for plot lines
                                            # as list of string colorspecs:
                                            # single letter, long name, or
                                            # web-style hex

#polaraxes.grid      : True    # display grid on polar axes
#axes3d.grid         : True    # display grid on 3d axes

### TICKS
# see http://matplotlib.org/api/axis_api.html#matplotlib.axis.Tick
#xtick.major.size     : 4      # major tick size in points
#xtick.minor.size     : 2      # minor tick size in points
#xtick.major.width    : 0.5    # major tick width in points
#xtick.minor.width    : 0.5    # minor tick width in points
#xtick.major.pad      : 4      # distance to major tick label in points
#xtick.minor.pad      : 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      : in     # direction: in, out, or inout

#ytick.major.size     : 4      # major tick size in points
#ytick.minor.size     : 2      # minor tick size in points
#ytick.major.width    : 0.5    # major tick width in points
#ytick.minor.width    : 0.5    # minor tick width in points
#ytick.major.pad      : 4      # distance to major tick label in points
#ytick.minor.pad      : 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      : in     # direction: in, out, or inout


### GRIDS
#grid.color       :   black   # grid color
#grid.linestyle   :   :       # dotted
#grid.linewidth   :   0.5     # in points
#grid.alpha       :   1.0     # transparency, between 0.0 and 1.0

### Legend
#legend.fancybox      : False  # if True, use a rounded box for the
                               # legend, else a rectangle
#legend.isaxes        : True
#legend.numpoints     : 2      # the number of points in the legend line
#legend.fontsize      : large
#legend.pad           : 0.0    # deprecated; the fractional whitespace inside the legend border
#legend.borderpad     : 0.5    # border whitespace in fontsize units
#legend.markerscale   : 1.0    # the relative size of legend markers vs. original
# the following dimensions are in axes coords
#legend.labelsep      : 0.010  # deprecated; the vertical space between the legend entries
#legend.labelspacing  : 0.5    # the vertical space between the legend entries in fraction of f
#legend.handlelen     : 0.05   # deprecated; the length of the legend lines
#legend.handlelength  : 2.     # the length of the legend lines in fraction of fontsize
#legend.handleheight  : 0.7     # the height of the legend handle in fraction of fontsize
#legend.handletextsep : 0.02   # deprecated; the space between the legend line and legend text
#legend.handletextpad : 0.8    # the space between the legend line and legend text in fraction
#legend.axespad       : 0.02   # deprecated; the border between the axes and legend edge
#legend.borderaxespad : 0.5   # the border between the axes and legend edge in fraction of font
#legend.columnspacing : 2.    # the border between the axes and legend edge in fraction of font
#legend.shadow        : False
#legend.frameon       : True   # whether or not to draw a frame around legend

### FIGURE
# See http://matplotlib.org/api/figure_api.html#matplotlib.figure.Figure
#figure.figsize   : 8, 6    # figure size in inches
#figure.dpi       : 80      # figure dots per inch
#figure.facecolor : 0.75    # 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

# The figure subplot parameters.  All dimensions are a fraction of the
# figure width or height
#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  : 0.1    # the bottom of the subplots of the figure
#figure.subplot.top     : 0.9    # the top of the subplots of the figure
#figure.subplot.wspace  : 0.2    # the amount of width reserved for blank space between subplot
#figure.subplot.hspace  : 0.2    # the amount of height reserved for white space between subplo

### IMAGES
#image.aspect : equal             # equal | auto | a number
#image.interpolation  : bilinear  # see help(imshow) for options
#image.cmap   : jet               # gray | jet etc...
#image.lut    : 256               # the size of the colormap lookup table
#image.origin : upper             # lower | upper
#image.resample  : False

### CONTOUR PLOTS
#contour.negative_linestyle :  dashed # dashed | solid

### 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.

# the default savefig params can be different from the display params
# Eg, you may want a higher resolution, or to make the figure
```

```
# background white
#savefig.dpi        : 100      # figure dots per inch
#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'.
#savefig.pad_inches : 0.1      # Padding to be used when bbox is set to 'tight'

# 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 quality
                                          # 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.image_noscale : False     # suppress scaling of raster data embedded in SVG
#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

# 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, eg --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               # 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                # 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

###ANIMATION settings
#animation.writer : ffmpeg         # MovieWriter 'backend' to use
#animation.codec : mp4             # 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 arugments to pass to mencoder
#animation.mencoder_path: 'ffmpeg' # Path to mencoder binary. Without full path
                                   # $PATH is searched
#animation.mencoder_args: ''       # Additional arugments to pass to mencoder
```

CHAPTER

SIX

USING MATPLOTLIB IN A PYTHON SHELL

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, eg, 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.

6.1 Ipython to the rescue

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 -pylab
Python 2.4.5 (#4, Apr 12 2008, 09:09:16)
IPython 0.9.0 -- An enhanced Interactive Python.

  Welcome to pylab, a matplotlib-based Python environment.
  For more information, type 'help(pylab)'.

In [1]: x = randn(10000)

In [2]: 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.

6.2 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:

```
>>> 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.

6.3 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)
```

CHAPTER

SEVEN

WORKING WITH TEXT

7.1 Text introduction

matplotlib has excellent text support, including mathematical expressions, truetype support for raster and vector outputs, newline separated text with arbitrary rotations, and unicode support. Because we embed the fonts directly in the output documents, eg for postscript or PDF, what you see on the screen is what you get in the hardcopy. freetype2 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.

You have total control over every text property (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, to support *mathematical expressions* anywhere in your figure.

7.2 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 an axis label to the x-axis; `matplotlib.axes.Axes.set_xlabel()` in the API.
- `ylabel()` - add an axis 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 bew configured with a variety of font and other properties. The example below shows all of these commands in action.

7.3 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 (eg `title()`, `xlabel()` and `text()`).

Property	Value Type
alpha	float
backgroundcolor	any matplotlib color
bbox	rectangle prop dict plus key 'pad' which is a pad in points
clip_box	a matplotlib.transform.Bbox instance
clip_on	[True | False]
clip_path	a Path instance and a Transform instance, a Patch
color	any matplotlib color
family	['serif' | 'sans-serif' | 'cursive' | 'fantasy' | 'monospace']
fontproperties	a matplotlib.font_manager.FontProperties instance
horizontalalignment or ha	['center' | 'right' | 'left']
label	any string
linespacing	float
multialignment	['left' | 'right' | 'center']
name or fontname	string eg, ['Sans' | 'Courier' | 'Helvetica' ...]
picker	[None|float|boolean|callable]
position	(x,y)
rotation	[angle in degrees 'vertical' | 'horizontal'
size or fontsize	[size in points | relative size eg 'smaller', 'x-large']
style or fontstyle	['normal' | 'italic' | 'oblique']
text	string or anything printable with '%s' conversion
transform	a matplotlib.transform transformation instance
variant	['normal' | 'small-caps']
verticalalignment or va	['center' | 'top' | 'bottom' | 'baseline']
visible	[True | False]
weight or fontweight	['normal' | 'bold' | 'heavy' | 'light' | 'ultrabold' | 'ultralight']
x	float
y	float
zorder	any number

You can layout 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.

7.4 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 (preceed 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 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:

```
# 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.

7.4.1 Subscripts and superscripts

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

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

$$\alpha_i > \beta_i \tag{7.1}$$

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

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

$$\sum_{i=0}^{\infty} x_i \tag{7.2}$$

7.4.2 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

$$\frac{3}{4}\binom{3}{4}\stackrel{3}{4} \tag{7.3}$$

Fractions can be arbitrarily nested:

```
r'$\frac{5 - \frac{1}{x}}{4}$'
```

produces

$$\frac{5 - \frac{1}{x}}{4} \tag{7.4}$$

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:

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

$$(\frac{5 - \frac{1}{x}}{4}) \tag{7.5}$$

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

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

$$\left(\frac{5 - \frac{1}{x}}{4}\right) \tag{7.6}$$

7.4.3 Radicals

Radicals can be produced with the `\sqrt[]{}` command. For example:

```
r'$\sqrt{2}$'
```

$$\sqrt{2} \tag{7.7}$$

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:

```
r'$\sqrt[3]{x}$'
```

$$\sqrt[3]{x} \tag{7.8}$$

7.4.4 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, eg, to write "sin" in a Roman font, enclose the text in a font command:

```
r'$s(t) = \mathcal{A}\mathrm{sin}(2 \omega t)$'
```

$$s(t) = \mathcal{A}\mathrm{sin}(2\omega t) \tag{7.9}$$

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:

```
r'$s(t) = \mathcal{A}\sin(2 \omega t)$'
```

$$s(t) = \mathcal{A}\sin(2\omega t) \tag{7.10}$$

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 caligraphy `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)
```

$$s(t) = \mathcal{A}\/\sin(2\omega t) \tag{7.11}$$

The choices available with all fonts are:

Command	Result
`\mathrm{Roman}`	Roman
`\mathit{Italic}`	*Italic*
`\mathtt{Typewriter}`	`Typewriter`
`\mathcal{CALLIGRAPHY}`	$\mathcal{CALLIGRAPHY}$

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

Command	Result
`\mathbb{blackboard}`	⋖Ͽ⅂⋊Ͽ⟍
`\mathrm{\mathbb{blackboard}}`	⋖Ͽ⅂⋊Ͽ⟍
`\mathfrak{Fraktur}`	$\mathfrak{Fraktur}$
`\mathsf{sansserif}`	$\mathsf{sansserif}$
`\mathrm{\mathsf{sansserif}}`	$\mathsf{sansserif}$

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**

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

Additionally, you can use `\mathdefault{...}` or its alias `\mathregular{...}` to use the font used for regular text outside of mathtext. 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.

Parameter	Corresponds to
`mathtext.it`	`\mathit{}` or default italic
`mathtext.rm`	`\mathrm{}` Roman (upright)
`mathtext.tt`	`\mathtt{}` Typewriter (monospace)
`mathtext.bf`	`\mathbf{}` bold italic
`mathtext.cal`	`\mathcal{}` calligraphic
`mathtext.sf`	`\mathsf{}` sans-serif

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.

7.4.5 Accents

An accent command may precede any symbol to add an accent above it. There are long and short forms for some of them.

Command	Result
`\acute a` or `\'a`	$\acute{a}$
`\bar a`	$\bar{a}$
`\breve a`	$\breve{a}$
`\ddot a` or `\"a`	$\ddot{a}$
`\dot a` or `\.a`	$\dot{a}$
``\grave a`` or ``\`a``	$\grave{a}$
`\hat a` or `\^a`	$\hat{a}$
`\tilde a` or `\~a`	$\tilde{a}$
`\vec a`	$\vec{a}$
`\overline{abc}`	$\overline{abc}$

In addition, there are two special accents that automatically adjust to the width of the symbols below:

Command	Result
`\widehat{xyz}`	$\widehat{xyz}$
`\widetilde{xyz}`	$\widetilde{xyz}$

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"$\hat i\ \ \hat \imath$"
```

$$\hat i\ \ \hat \imath \tag{7.12}$$

7.4.6 Symbols

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

Lower-case Greek

α `\alpha`	β `\beta`	χ `\chi`	δ `\delta`	$\digamma$ `\digamma`
ϵ `\epsilon`	η `\eta`	γ `\gamma`	ι `\iota`	κ `\kappa`
λ `\lambda`	μ `\mu`	ν `\nu`	ω `\omega`	ϕ `\phi`
π `\pi`	ψ `\psi`	ρ `\rho`	σ `\sigma`	τ `\tau`
θ `\theta`	υ `\upsilon`	ε `\varepsilon`	$\varkappa$ `\varkappa`	φ `\varphi`
ϖ `\varpi`	ϱ `\varrho`	ς `\varsigma`	ϑ `\vartheta`	ξ `\xi`
ζ `\zeta`				

Upper-case Greek

Δ \Delta	Γ \Gamma	Λ \Lambda	Ω \Omega	Φ \Phi	Π \Pi
Ψ \Psi	Σ \Sigma	Θ \Theta	ϒ \Upsilon	Ξ \Xi	℧ \mho
∇ \nabla					

Hebrew

ℵ \aleph	ℶ \beth	ℸ \daleth	ℷ \gimel

Delimiters

<table>
<tr><td>/ /</td><td>[[</td><td>⇓ \Downarrow</td><td>⇑ \Uparrow</td><td>‖ \Vert</td><td>\ \backslash</td></tr>
<tr><td>↓ \downarrow</td><td>⟨ \langle</td><td>⌈ \lceil</td><td>⌊ \lfloor</td><td>⌞ \llcorner</td><td>⌟ \lrcorner</td></tr>
<tr><td>⟩ \rangle</td><td>⌉ \rceil</td><td>⌋ \rfloor</td><td>⌜ \ulcorner</td><td>↑ \uparrow</td><td>⌝ \urcorner</td></tr>
<tr><td>| \vert</td><td>{ \{</td><td>‖ \|</td><td>} \}</td><td>]]</td><td>| |</td></tr>
</table>

Big symbols

⋂ \bigcap	⋃ \bigcup	⨀ \bigodot	⨁ \bigoplus	⨂ \bigotimes
⨄ \biguplus	⋁ \bigvee	⋀ \bigwedge	∐ \coprod	∫ \int
∮ \oint	∏ \prod	∑ \sum		

Standard function names

Pr \Pr	arccos \arccos	arcsin \arcsin	arctan \arctan
arg \arg	cos \cos	cosh \cosh	cot \cot
coth \coth	csc \csc	deg \deg	det \det
dim \dim	exp \exp	gcd \gcd	hom \hom
inf \inf	ker \ker	lg \lg	lim \lim
lim inf \liminf	lim sup \limsup	ln \ln	log \log
max \max	min \min	sec \sec	sin \sin
sinh \sinh	sup \sup	tan \tan	tanh \tanh

Binary operation and relation symbols

≎ \Bumpeq	⋒ \Cap	⋓ \Cup
≑ \Doteq	⋈ \Join	⋐ \Subset
⋑ \Supset	⊩ \Vdash	⊪ \Vvdash
≈ \approx	≊ \approxeq	∗ \ast
≍ \asymp	϶ \backepsilon	∽ \backsim
⋍ \backsimeq	⊼ \barwedge	∵ \because
≬ \between	◯ \bigcirc	▽ \bigtriangledown
△ \bigtriangleup	◀ \blacktriangleleft	▶ \blacktriangleright
⊥ \bot	⋈ \bowtie	⊡ \boxdot
⊟ \boxminus	⊞ \boxplus	⊠ \boxtimes
● \bullet	≏ \bumpeq	∩ \cap
· \cdot	∘ \circ	≗ \circeq
≔ \coloneq	≅ \cong	∪ \cup
⋞ \curlyeqprec	⋟ \curlyeqsucc	⋎ \curlyvee
⋏ \curlywedge	† \dag	⊣ \dashv
‡ \ddag	⋄ \diamond	÷ \div
⋇ \divideontimes	≐ \doteq	≑ \doteqdot
∔ \dotplus	⩞ \doublebarwedge	≖ \eqcirc
≕ \eqcolon	≂ \eqsim	⪖ \eqslantgtr
⪕ \eqslantless	≡ \equiv	≒ \fallingdotseq

⌢ \frown	≥ \geq	≧ \geqq
⩾ \geqslant	≫ \gg	⋙ \ggg
⪊ \gnapprox	≩ \gneqq	⋧ \gnsim
⪆ \gtrapprox	⋗ \gtrdot	⋛ \gtreqless
⪌ \gtreqqless	≷ \gtrless	≳ \gtrsim
∈ \in	⊺ \intercal	⋋ \leftthreetimes
≤ \leq	≦ \leqq	⩽ \leqslant
⪅ \lessapprox	⋖ \lessdot	⋚ \lesseqgtr
⪋ \lesseqqgtr	≶ \lessgtr	≲ \lesssim
≪ \ll	⋘ \lll	⪉ \lnapprox
≨ \lneqq	⋦ \lnsim	⋉ \ltimes
∣ \mid	⊧ \models	∓ \mp
⊯ \nVDash	⊮ \nVdash	≉ \napprox
≇ \ncong	≠ \ne	≠ \neq
≠ \neq	≢ \nequiv	≱ \ngeq
≯ \ngtr	∋ \ni	≰ \nleq
≮ \nless	∤ \nmid	∉ \notin
∦ \nparallel	⊀ \nprec	≁ \nsim
⊄ \nsubset	⊈ \nsubseteq	⊁ \nsucc
⊅ \nsupset	⊉ \nsupseteq	⋪ \ntriangleleft

⋬ \ntrianglelefteq	⋫ \ntriangleright	⋭ \ntrianglerighteq
⊯ \nvDash	⊬ \nvdash	⊙ \odot
⊖ \ominus	⊕ \oplus	⊘ \oslash
⊗ \otimes	∥ \parallel	⊥ \perp
⋔ \pitchfork	± \pm	≺ \prec
⪷ \precapprox	≼ \preccurlyeq	⪯ \preceq
⪹ \precnapprox	⋨ \precnsim	≾ \precsim
∝ \propto	⋌ \rightthreetimes	≓ \risingdotseq
⋊ \rtimes	∼ \sim	≃ \simeq
/ \slash	⌣ \smile	⊓ \sqcap
⊔ \sqcup	⊏ \sqsubset	⊏ \sqsubset
⊑ \sqsubseteq	⊐ \sqsupset	⊐ \sqsupset
⊒ \sqsupseteq	⋆ \star	⊂ \subset
⊆ \subseteq	⫅ \subseteqq	⊊ \subsetneq
⫋ \subsetneqq	≻ \succ	⪸ \succapprox
≽ \succcurlyeq	⪰ \succeq	⪺ \succnapprox
⋩ \succnsim	≿ \succsim	⊃ \supset
⊇ \supseteq	⫆ \supseteqq	⊋ \supsetneq
⫌ \supsetneqq	∴ \therefore	× \times
⊤ \top	◂ \triangleleft	⊴ \trianglelefteq
≜ \triangleq	▸ \triangleright	⊵ \trianglerighteq
⊎ \uplus	⊨ \vDash	∝ \varpropto
⊲ \vartriangleleft	⊳ \vartriangleright	⊢ \vdash
∨ \vee	⊻ \veebar	∧ \wedge
≀ \wr		

Arrow symbols

⇓ \Downarrow	⇐ \Leftarrow
⇔ \Leftrightarrow	⇚ \Lleftarrow
⟸ \Longleftarrow	⟺ \Longleftrightarrow
⟹ \Longrightarrow	↰ \Lsh
⇗ \Nearrow	⇖ \Nwarrow
⇒ \Rightarrow	⇛ \Rrightarrow
↱ \Rsh	⇘ \Searrow
⇙ \Swarrow	⇑ \Uparrow
⇕ \Updownarrow	↺ \circlearrowleft
↻ \circlearrowright	↶ \curvearrowleft
↷ \curvearrowright	⇠ \dashleftarrow
⇢ \dashrightarrow	↓ \downarrow
⇊ \downdownarrows	⇃ \downharpoonleft
⇂ \downharpoonright	↩ \hookleftarrow
↪ \hookrightarrow	⇝ \leadsto
← \leftarrow	↢ \leftarrowtail
↽ \leftharpoondown	↼ \leftharpoonup
⇇ \leftleftarrows	↔ \leftrightarrow
⇆ \leftrightarrows	⇋ \leftrightharpoons
↭ \leftrightsquigarrow	⇜ \leftsquigarrow

⟵ \longleftarrow	⟷ \longleftrightarrow
⟼ \longmapsto	⟶ \longrightarrow
↫ \looparrowleft	↬ \looparrowright
↦ \mapsto	⊸ \multimap
⇍ \nLeftarrow	⇎ \nLeftrightarrow
⇏ \nRightarrow	↗ \nearrow
↚ \nleftarrow	↮ \nleftrightarrow
↛ \nrightarrow	↖ \nwarrow
→ \rightarrow	↣ \rightarrowtail
⇁ \rightharpoondown	⇀ \rightharpoonup
⇄ \rightleftarrows	⇄ \rightleftarrows
⇌ \rightleftharpoons	⇌ \rightleftharpoons
⇉ \rightrightarrows	⇉ \rightrightarrows
⇝ \rightsquigarrow	↘ \searrow
↙ \swarrow	→ \to
↞ \twoheadleftarrow	↠ \twoheadrightarrow
↑ \uparrow	↕ \updownarrow
↕ \updownarrow	↿ \upharpoonleft
↾ \upharpoonright	⇈ \upuparrows

Miscellaneous symbols

$ \$	Å \AA	Ⅎ \Finv
⅁ \Game	ℑ \Im	¶ \P
ℜ \Re	§ \S	∠ \angle
‵ \backprime	★ \bigstar	■ \blacksquare
▲ \blacktriangle	▼ \blacktriangledown	⋯ \cdots
✓ \checkmark	® \circledR	Ⓢ \circledS
♣ \clubsuit	∁ \complement	© \copyright
⋱ \ddots	♢ \diamondsuit	ℓ \ell
∅ \emptyset	ð \eth	∃ \exists
♭ \flat	∀ \forall	ℏ \hbar
♡ \heartsuit	ℏ \hslash	∭ \iiint
∬ \iint	∬ \iint	ı \imath
∞ \infty	ȷ \jmath	… \ldots
∡ \measuredangle	♮ \natural	¬ \neg
∄ \nexists	∰ \oiiint	∂ \partial
′ \prime	♯ \sharp	♠ \spadesuit
∢ \sphericalangle	\ss	▽ \triangledown
∅ \varnothing	△ \vartriangle	⋮ \vdots
℘ \wp	¥ \yen	

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'$\u23ce$'
```

7.4.7 Example

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

7.5 Typesetting With XeLaTeX/LuaLaTeX

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

```
matplotlib.use('pgf')
```

or registering it for handling pdf output

```
from matplotlib.backends.backend_pgf import FigureCanvasPgf
matplotlib.backend_bases.register_backend('pdf', FigureCanvasPgf)
```

The second method allows you to keep using regular interactive backends and to save 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 `PATH`.

Rc parameters that control the behavior of the pgf backend:

Parameter	Documentation
pgf.preamble	Lines to be included in the LaTeX preamble
pgf.rcfonts	Setup fonts from rc params using the fontspec package
pgf.texsystem	Either "xelatex", "lualatex" or "pdflatex"

Note: TeX defines a set of secial characters, such as:

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

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

7.5.1 Font specification

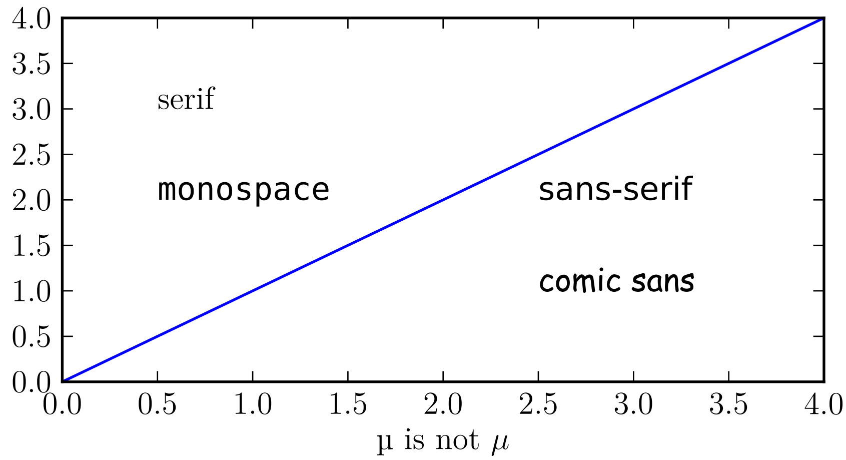
The fonts used for obtaining the size of text elements or when compiling figures to PDF are usually defined in the matplotlib rc parameters. You can also use the LaTeX default Computer Modern fonts by clearing the lists for `font.serif`, `font.sans-serif` or `font.monospace`. Please note that the glyph 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 Computer Modern Unicode fonts *CMU Serif*, *CMU Sans Serif*, etc.

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

import matplotlib as mpl
mpl.use("pgf")
pgf_with_rc_fonts = {
    "font.family": "serif",
    "font.serif": [],                   # use latex default serif font
    "font.sans-serif": ["DejaVu Sans"], # use a specific sans-serif font
}
mpl.rcParams.update(pgf_with_rc_fonts)

import matplotlib.pyplot as plt
```

```
plt.figure(figsize=(4.5,2.5))
plt.plot(range(5))
plt.text(0.5, 3., "serif")
plt.text(0.5, 2., "monospace", family="monospace")
plt.text(2.5, 2., "sans-serif", family="sans-serif")
plt.text(2.5, 1., "comic sans", family="Comic Sans MS")
plt.xlabel(u"µ is not $\\mu$")
plt.tight_layout(.5)
```



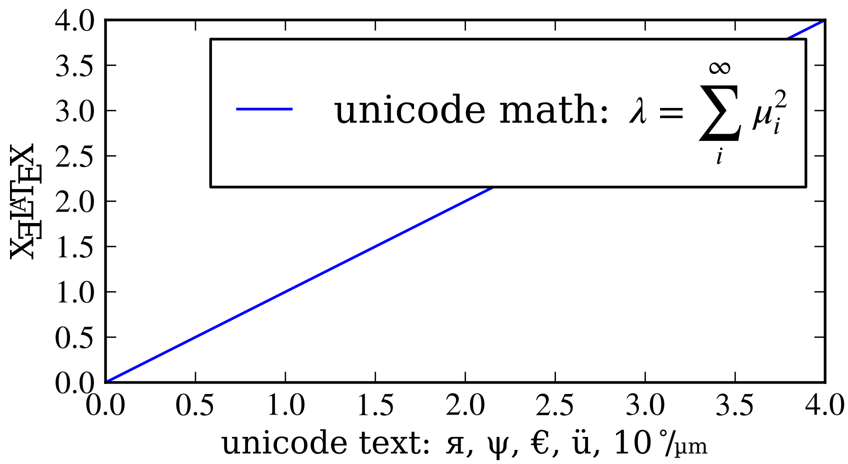
7.5.2 Custom preamble

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 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`.

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

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{metalogo}",         # load additional packages
         r"\usepackage{unicode-math}", # unicode math setup
         r"\setmathfont{XITS Math}",
         r"\setmainfont{DejaVu Serif}", # font setup via preamble
         ]
```

```
}
mpl.rcParams.update(pgf_with_custom_preamble)
```



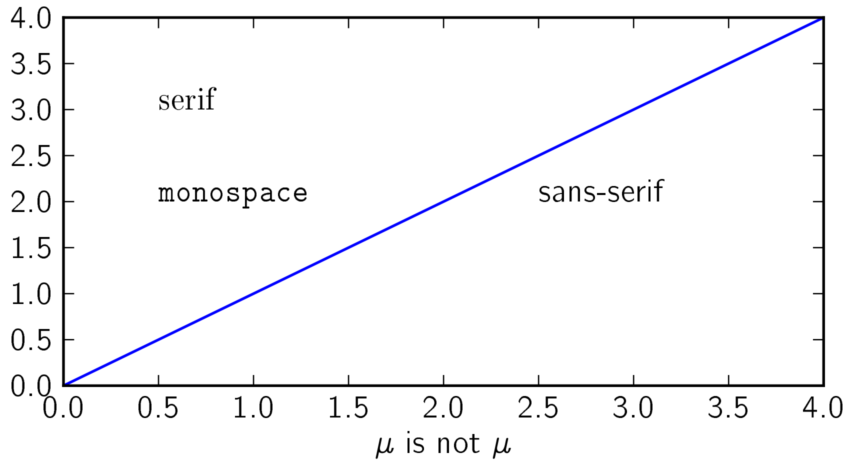
7.5.3 Choosing the TeX system

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{cmbright}",
         ]
}
mpl.rcParams.update(pgf_with_pdflatex)

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"µ is not $\\mu$")
plt.tight_layout(.5)
```



7.5.4 Possible hangups

- 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 before reporting problems.
- 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.
- 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.

7.5.5 Troubleshooting

- 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 before reporting problems.
- If you still need help, please see *Report a problem*

7.6 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:

```
font.family        : serif
font.serif         : Times, Palatino, New Century Schoolbook, Bookman, Computer Modern Roman
font.sans-serif    : Helvetica, Avant Garde, Computer Modern Sans serif
font.cursive       : Zapf Chancery
font.monospace     : Courier, Computer Modern Typewriter

text.usetex        : true
```

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:

```
from matplotlib import rc
rc('font',**{'family':'sans-serif','sans-serif':['Helvetica']})
## for Palatino and other serif fonts use:
#rc('font',**{'family':'serif','serif':['Palatino']})
rc('text', usetex=True)
```

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

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:

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

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

7.6.1 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`:

7.6.2 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 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.

7.6.3 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.

7.6.4 Troubleshooting

- Try deleting your `.matplotlib/tex.cache` directory. If you don't know where to find `.matplotlib`, see *.matplotlib directory location*.
- 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 *Report a problem*

7.7 Annotating text

For a more detailed introduction to annotations, see *Annotating Axes*.

The uses of the basic `text()` command above 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.

In this 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 – you can specify the coordinate system of `xy` and `xytext` with one of the following strings for `xycoords` and `textcoords` (default is 'data')

argument	coordinate system
'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'	0,1 is lower left of axes and 1,1 is upper right
'data'	use the axes data coordinate system

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

```
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',
            )
```

For physical coordinate systems (points or pixels) the origin is the (bottom, left) of the figure or axes. If the value is negative, however, the origin is from the (right, top) of the figure or axes, analogous to negative indexing of sequences.

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

`arrowprops` key	description
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
**kwargs	any key for `matplotlib.patches.Polygon`, e.g. `facecolor`

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 the '~matplotlib.Axes.annotate' to the ''Text` instance

For more on all the wild and wonderful things you can do with annotations, including fancy arrows, see *Annotating Axes* and *pylab_examples-annotation_demo*.

CHAPTER

EIGHT

IMAGE TUTORIAL

8.1 Startup commands

At the very least, you'll need to have access to the `imshow()` function. There are a couple of ways to do it. The easy way for an interactive environment:

```
$ipython -pylab
```

The imshow function is now directly accessible (it's in your namespace). See also *Pyplot tutorial*.

The more expressive, easier to understand later method (use this in your scripts to make it easier for others (including your future self) to read) is to use the matplotlib API (see *Artist tutorial*) where you use explicit namespaces and control object creation, etc...

```
In [1]: import matplotlib.pyplot as plt
In [2]: import matplotlib.image as mpimg
In [3]: import numpy as np
```

Examples below will use the latter method, for clarity. In these examples, if you use the -pylab method, you can skip the "mpimg." and "plt." prefixes.

8.2 Importing image data into Numpy arrays

Plotting image data is supported by the Python Image Library (PIL), . Natively, matplotlib only supports PNG images. The commands shown below fall back on PIL if the native read fails.

The image used in this example is a PNG file, but keep that PIL 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...

```
In [4]: img=mpimg.imread('stinkbug.png')
Out[4]:
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]],

       [[ 0.41176471,  0.41176471,  0.41176471],
        [ 0.41176471,  0.41176471,  0.41176471],
        [ 0.41176471,  0.41176471,  0.41176471],
        ...,
```

```
 [ 0.42745098,  0.42745098,  0.42745098],
 [ 0.42745098,  0.42745098,  0.42745098],
 [ 0.42745098,  0.42745098,  0.42745098]],

[[ 0.41960785,  0.41960785,  0.41960785],
 [ 0.41568628,  0.41568628,  0.41568628],
 [ 0.41568628,  0.41568628,  0.41568628],
 ...,
 [ 0.43137255,  0.43137255,  0.43137255],
 [ 0.43137255,  0.43137255,  0.43137255],
 [ 0.43137255,  0.43137255,  0.43137255]],

...,
[[ 0.43921569,  0.43921569,  0.43921569],
 [ 0.43529412,  0.43529412,  0.43529412],
 [ 0.43137255,  0.43137255,  0.43137255],
 ...,
 [ 0.45490196,  0.45490196,  0.45490196],
 [ 0.4509804 ,  0.4509804 ,  0.4509804 ],
 [ 0.4509804 ,  0.4509804 ,  0.4509804 ]],

[[ 0.44313726,  0.44313726,  0.44313726],
 [ 0.44313726,  0.44313726,  0.44313726],
 [ 0.43921569,  0.43921569,  0.43921569],
 ...,
 [ 0.4509804 ,  0.4509804 ,  0.4509804 ],
 [ 0.44705883,  0.44705883,  0.44705883],
 [ 0.44705883,  0.44705883,  0.44705883]],

[[ 0.44313726,  0.44313726,  0.44313726],
 [ 0.4509804 ,  0.4509804 ,  0.4509804 ],
 [ 0.4509804 ,  0.4509804 ,  0.4509804 ],
 ...,
 [ 0.44705883,  0.44705883,  0.44705883],
 [ 0.44705883,  0.44705883,  0.44705883],
 [ 0.44313726,  0.44313726,  0.44313726]]], dtype=float32)
```

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 PIL 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.

8.3 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.

```
In [5]: imgplot = plt.imshow(img)
```

You can also plot any numpy array - just remember that the datatype must be float32 (and range from 0.0 to 1.0) or uint8.

8.3.1 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 on channel of our data:

```
In [6]: lum_img = img[:,:,0]
```

This is array slicing. You can read more in the Numpy tutorial.

```
In [7]: imgplot = plt.imshow(lum_img)
```

Now, with a luminosity image, the default colormap (aka lookup table, LUT), is applied. The default is called jet. There are plenty of others to choose from. Let's set some others using the `set_cmap()` method on our image plot object:

```
In [8]: imgplot.set_cmap('hot')

In [9]: imgplot.set_cmap('spectral')
```

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

8.3.2 Color scale reference

It's helpful to have an idea of what value a color represents. We can do that by adding color bars. It's as easy as one line:

```
In [10]: 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.

8.3.3 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[10]: plt.hist(lum_img.flatten(), 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 calling the `set_clim()` method of the image plot object.

```
In[11]: imgplot.set_clim(0.0,0.7)
```

8.3.4 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.

```
In [8]: import Image
In [9]: img = Image.open('stinkbug.png')    # Open image as PIL image object
In [10]: rsize = img.resize((img.size[0]/10,img.size[1]/10)) # Use PIL to resize
In [11]: rsizeArr = np.asarray(rsize)  # Get array back
In [12]: imgplot = plt.imshow(rsizeArr)
```

Here we have the default interpolation, bilinear, since we did not give `imshow()` any interpolation argument.

Let's try some others:

```
In [10]: imgplot.set_interpolation('nearest')
```

```
In [10]: imgplot.set_interpolation('bicubic')
```

Bicubic interpolation is often used when blowing up photos - people tend to prefer blurry over pixelated.

CHAPTER

NINE

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 his 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 (eg. numpy arrays and strings) and create primitive Artist instances as needed (eg. 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 xdata')
```

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.

9.1 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, eg. rectangles, circles and polygons). Every matplotlib `Artist` has the following properties

Property	Description
alpha	The transparency - a scalar from 0-1
animated	A boolean that is used to facilitate animated drawing
axes	The axes that the Artist lives in, possibly None
clip_box	The bounding box that clips the Artist
clip_on	Whether clipping is enabled
clip_path	The path the artist is clipped to
contains	A picking function to test whether the artist contains the pick point
figure	The figure instance the artist lives in, possibly None
label	A text label (eg. for auto-labeling)
picker	A python object that controls object picking
transform	The transformation
visible	A boolean whether the artist should be drawn
zorder	A number which determines the drawing order

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:

```
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:

```
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, eg. `Figure` and `Rectangle`. Here are the `Figure` rectangle properties mentioned above:

```
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
edgecolor 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
transform = <Affine object at 0x134cca84>
verts = ((0, 0), (0, 1), (1, 1), (1, 0))
visible = True
width = 1
window_extent = <Bbox object at 0x134acbcc>
x = 0
y = 0
zorder = 1
```

The docstrings for all of the classes also contain the `Artist` properties, so you can consult the interactive "help" or the *artists* for a listing of properties for a given object.

9.2 Object containers

Now that we know how to inspect and set the properties of a given object we want to configure, we need to now 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.

9.3 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:

```
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

Figure attribute	Description
axes	A list of Axes instances (includes Subplot)
patch	The Rectangle background
images	A list of FigureImages patches - useful for raw pixel display
legends	A list of Figure Legend instances (different from Axes.legends)
lines	A list of Figure Line2D instances (rarely used, see Axes.lines)
patches	A list of Figure patches (rarely used, see Axes.patches)
texts	A list Figure Text instances

9.4 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:

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

When you call a plotting method, eg. 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:

```
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:

```
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:

```
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 xlimits 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 `Axes` helper methods for creating primitive `Artists` and adding them to their respective containers. The table below summarizes a small sampling of them, the kinds of `Artist` they create, and where they store them

Helper method	Artist	Container
ax.annotate - text annotations	Annotate	ax.texts
ax.bar - bar charts	Rectangle	ax.patches
ax.errorbar - error bar plots	Line2D and Rectangle	ax.lines and ax.patches
ax.fill - shared area	Polygon	ax.patches
ax.hist - histograms	Rectangle	ax.patches
ax.imshow - image data	AxesImage	ax.images
ax.legend - axes legends	Legend	ax.legends
ax.plot - xy plots	Line2D	ax.lines
ax.scatter - scatter charts	PolygonCollection	ax.collections
ax.text - text	Text	ax.texts

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

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

Below is a summary of the Artists that the Axes contains

Axes attribute	Description
artists	A list of Artist instances
patch	Rectangle instance for Axes background
collections	A list of Collection instances
images	A list of AxesImage
legends	A list of Legend instances
lines	A list of Line2D instances
patches	A list of Patch instances
texts	A list of Text instances
xaxis	matplotlib.axis.XAxis instance
yaxis	matplotlib.axis.YAxis instance

9.5 Axis containers

The `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 `Axis` also stores the data and view intervals used in auto-scaling, panning and zooming, as well as the `Locator` and `Formatter` instances which control where the ticks are placed and how they are represented as strings.

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

```
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>
```

Here is a summary of some of the useful accessor methods of the `Axis` (these have corresponding setters where useful, such as set_major_formatter)

Accessor method	Description
get_scale	The scale of the axis, eg 'log' or 'linear'
get_view_interval	The interval instance of the axis view limits
get_data_interval	The interval instance of the axis data limits
get_gridlines	A list of grid lines for the Axis
get_label	The axis label - a Text instance
get_ticklabels	A list of Text instances - keyword minor=True|False
get_ticklines	A list of Line2D instances - keyword minor=True|False
get_ticklocs	A list of Tick locations - keyword minor=True|False
get_major_locator	The matplotlib.ticker.Locator instance for major ticks
get_major_formatter	The matplotlib.ticker.Formatter instance for major ticks
get_minor_locator	The matplotlib.ticker.Locator instance for minor ticks
get_minor_formatter	The matplotlib.ticker.Formatter instance for minor ticks
get_major_ticks	A list of Tick instances for major ticks
get_minor_ticks	A list of Tick instances for minor ticks
grid	Turn the grid on or off for the major or minor ticks

Here is an example, not recommended for its beauty, which customizes the axes and tick properties

9.6 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.

Tick attribute	Description
tick1line	Line2D instance
tick2line	Line2D instance
gridline	Line2D instance
label1	Text instance
label2	Text instance
gridOn	boolean which determines whether to draw the tickline
tick1On	boolean which determines whether to draw the 1st tickline
tick2On	boolean which determines whether to draw the 2nd tickline
label1On	boolean which determines whether to draw tick label
label2On	boolean which determines whether to draw tick label

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

CHAPTER

TEN

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 "pyplot.subplot" but uses 0-based indexing and let subplot to occupy multiple cells.

10.1 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:

```
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,

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

For example, the following commands

```
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

10.2 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

```
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, SubplotSpec that spans multiple cells, use slice.

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

The above example becomes

```
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])
```

10.3 Adjust GridSpec layout

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

```
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

```
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

10.4 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.

```
gs0 = gridspec.GridSpec(1, 2)

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

10.5 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.

10.6 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.

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

ax1 = plt.subplot(gs[0])
```

```
ax2 = plt.subplot(gs[1])
ax3 = plt.subplot(gs[2])
ax4 = plt.subplot(gs[3])
```

CHAPTER

ELEVEN

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.

11.1 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.

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.

When you have multiple subplots, often you see labels of different axes overlapping each other.

`tight_layout()` will also adjust spacing between subplots to minimize the overlaps.

`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.

`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.

It works with subplots created with `subplot2grid()`. In general, subplots created from the gridspec (*Customizing Location of Subplot Using GridSpec*) will work.

Although not thoroughly tested, it seems to work for subplots with aspect != "auto" (e.g., axes with images).

11.1.1 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.

11.1.2 Use with GridSpec

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

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).

For example, this can be used for a figure with multiple gridspecs.

We may try to match the top and bottom of two grids

```
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.

11.1.3 Use with AxesGrid1

While limited, the axes_grid1 toolkit is also supported.

11.1.4 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 colobar as a subplot using the gridspec.

Another option is to use AxesGrid1 toolkit to explicitly create an axes for colorbar.

CHAPTER

TWELVE

LEGEND GUIDE

Do not proceed unless you already have read `legend()` and `matplotlib.legend.Legend`!

12.1 What to be displayed

The legend command has a following call signature:

```
legend(*args, **kwargs)
```

If len(args) is 2, the first argument should be a list of artist to be labeled, and the second argument should a list of string labels. If len(args) is 0, it automatically generate the legend from label properties of the child artists by calling `get_legend_handles_labels()` method. For example, *ax.legend()* is equivalent to:

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

The `get_legend_handles_labels()` method returns a tuple of two lists, i.e., list of artists and list of labels (python string). However, it does not return all of its child artists. It returns artists that are currently supported by matplotlib.

For matplotlib v1.0 and earlier, the supported artists are as follows.

- `Line2D`
- `Patch`
- `LineCollection`
- `RegularPolyCollection`
- `CircleCollection`

And, `get_legend_handles_labels()` returns all artists in *ax.lines*, *ax.patches* and artists in *ax.collection* which are instance of `LineCollection` or `RegularPolyCollection`. The label attributes (returned by get_label() method) of collected artists are used as text labels. If label attribute is empty string or starts with "_", those artists will be ignored.

Therefore, plots drawn by some *pyplot* commands are not supported by legend. For example, `fill_between()` creates `PolyCollection` that is not supported. Also support is limted for some commands that creat multiple artists. For example, `errorbar()` creates multiples `Line2D` instances.

Unfortunately, there is no easy workaround when you need legend for an artist not supported by matplotlib (You may use one of the supported artist as a proxy. See below)

In newer version of matplotlib (v1.1 and later), the matplotlib internals are revised to support

- complex plots that creates multiple artists (e.g., bar, errorbar, etc)
- custom legend handles

See below for details of new functionality.

12.1.1 Adjusting the Order of Legend items

When you want to customize the list of artists to be displayed in the legend, or their order of appearance. There are a two options. First, you can keep lists of artists and labels, and explicitly use these for the first two argument of the legend call.:

```
p1, = plot([1,2,3])
p2, = plot([3,2,1])
p3, = plot([2,3,1])
legend([p2, p1], ["line 2", "line 1"])
```

Or you may use `get_legend_handles_labels()` to retrieve list of artist and labels and manipulate them before feeding them to legend call.:

```
ax = subplot(1,1,1)
p1, = ax.plot([1,2,3], label="line 1")
p2, = ax.plot([3,2,1], label="line 2")
p3, = ax.plot([2,3,1], label="line 3")

handles, labels = ax.get_legend_handles_labels()

# reverse the order
ax.legend(handles[::-1], labels[::-1])

# or sort them by labels
import operator
hl = sorted(zip(handles, labels),
            key=operator.itemgetter(1))
handles2, labels2 = zip(*hl)
```

```
ax.legend(handles2, labels2)
```

12.1.2 Using Proxy Artist

When you want to display legend for an artist not supported by matplotlib, you may use another artist as a proxy. For example, you may create a proxy artist without adding it to the axes (so the proxy artist will not be drawn in the main axes) and feed it to the legend function.:

```
p = Rectangle((0, 0), 1, 1, fc="r")
legend([p], ["Red Rectangle"])
```

12.2 Multicolumn Legend

By specifying the keyword argument *ncol*, you can have a multi-column legend. Also, mode="expand" horizontally expand the legend to fill the axes area. See legend_demo3.py for example.

12.3 Legend location

The location of the legend can be specified by the keyword argument *loc*, either by string or a integer number.

String	Number
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

By default, the legend will anchor to the bbox of the axes (for legend) or the bbox of the figure (figlegend). You can specify your own bbox using *bbox_to_anchor* argument. *bbox_to_anchor* can be an instance of `BboxBase`, a tuple of 4 floats (x, y, width, height of the bbox), or a tuple of 2 floats (x, y with width=height=0). Unless *bbox_transform* argument is given, the coordinates (even for the bbox instance) are considered as normalized axes coordinates.

For example, if you want your axes legend located at the figure corner (instead of the axes corner):

```
l = legend(bbox_to_anchor=(0, 0, 1, 1), bbox_transform=gcf().transFigure)
```

Also, you can place above or outer right-hand side of the axes,

12.4 Multiple Legend

Sometime, you want to split the legend into multiple ones.:

```
p1, = plot([1,2,3])
p2, = plot([3,2,1])
legend([p1], ["Test1"], loc=1)
legend([p2], ["Test2"], loc=4)
```

However, the above code only shows the second legend. When the legend command is called, a new legend instance is created and old ones are removed from the axes. Thus, you need to manually add the removed legend.

12.5 Legend of Complex Plots

In matplotlib v1.1 and later, the legend is improved to support more plot commands and ease the customization.

12.5.1 Artist Container

The Artist Container is simple class (derived from tuple) that contains multiple artists. This is introduced primarily to support legends for complex plot commands that create multiple artists.

Axes instances now have a "containers" attribute (which is a list, and this is only intended to be used for generating a legend). The items in this attribute are also returned by `get_legend_handles_labels()`.

For example, "bar" command creates a series of Rectangle patches. Previously, it returned a list of these patches. With the current change, it creates a container object of these rectangle patches (and these patches are added to Axes.patches attribute as before) and return it instead. As the container class is derived from a tuple, it should be backward-compatible. Furthermore, the container object is added to the Axes.containers attributes so that legend command can properly create a legend for the bar. Thus, you may do

```
b1 = bar([0, 1, 2], [0.2, 0.3, 0.1], width=0.4,
         label="Bar 1", align="center")
legend()
```

or

```
b1 = bar([0, 1, 2], [0.2, 0.3, 0.1], width=0.4, align="center")
legend([b1], ["Bar 1"])
```

At this time of writing, however, only "bar", "errorbar", and "stem" are supported (hopefully the list will increase). Here is an example.

12.5.2 Legend Handler

One of the changes is that drawing of legend handles has been delegated to legend handlers. For example, `Line2D` instances are handled by `HandlerLine2D`. The mapping between the artists and their corresponding handlers are defined in a handler_map of the legend. The handler_map is a dictionary of key-handler pair, where key can be an artist instance or its class. And the handler is a Handler instance.

Let's consider the following sample code,

```
legend([p_1, p_2,..., p_i, ...], ["Test 1", "Test 2", ..., "Test i",...])
```

For each *p_i*, matplotlib

1. check if *p_i* is in the handler_map
2. if not, iterate over type(p_i).mro() until a matching key is found in the handler_map

Unless specified, the defaul handler_map is used. Below is a partial list of key-handler pairs included in the default handler map.

- Line2D : legend_handler.HandlerLine2D()
- Patch : legend_handler.HandlerPatch()
- LineCollection : legend_handler.HandlerLineCollection()
- ...

The legend() command takes an optional argument of "handler_map". When provided, the default handler map will be updated (using dict.update method) with the provided one.

```
p1, = plot(x, "ro", label="test1")
p2, = plot(y, "b+", ms=10, label="test2")

my_handler = HandlerLine2D(numpoints=1)

legend(handler_map={Line2D:my_handler})
```

The above example will use *my_handler* for any Line2D instances (p1 and p2).

```
legend(handler_map={p1:HandlerLine2D(numpoints=1)})
```

In the above example, only *p1* will be handled by *my_handler*, while others will be handled by default handlers.

The curent default handler_map has handlers for errorbar and bar plots. Also, it includes an entry for `tuple` which is mapped to `HandlerTuple`. It simply plots over all the handles for items in the given tuple. For example,

12.5.3 Implement a Custom Handler

Handler can be any callable object with following signature.

```
def __call__(self, legend, orig_handle,
             fontsize,
             handlebox):
```

Where *legend* is the legend itself, *orig_handle* is the original plot (*p_i* in the above example), *fontsize* is the fontsize in pixles, and *handlebox* is a OffsetBox instance. Within the call, you 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). See `legend_handler` for more details.

CHAPTER

THIRTEEN

EVENT HANDLING AND PICKING

matplotlib works with 6 user interface toolkits (wxpython, tkinter, qt, gtk, fltk 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.

13.1 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

Event name	Class and description
'button_press_event'	`MouseEvent` - mouse button is pressed
'button_release_event'	`MouseEvent` - mouse button is released
'draw_event'	`DrawEvent` - canvas draw
'key_press_event'	`KeyEvent` - key is pressed
'key_release_event'	`KeyEvent` - key is released
'motion_notify_event'	`MouseEvent` - mouse motion
'pick_event'	`PickEvent` - an object in the canvas is selected
'resize_event'	`ResizeEvent` - figure canvas is resized
'scroll_event'	`MouseEvent` - mouse scroll wheel is rolled
'figure_enter_event'	`LocationEvent` - mouse enters a new figure
'figure_leave_event'	`LocationEvent` - mouse leaves a figure
'axes_enter_event'	`LocationEvent` - mouse enters a new axes
'axes_leave_event'	`LocationEvent` - mouse leaves an axes

13.2 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:

```
from matplotlib import pyplot as plt

class LineBuilder:
    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'

13.2.1 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:

```
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'
        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 'x0=%f, xpress=%f, event.xdata=%f, dx=%f, x0+dx=%f'%(x0, xpress, event.xdata, d
        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:

```
# 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_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: 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()
```

13.3 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:

```
"""
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()
```

13.4 Object picking

You can enable picking by setting the `picker` property of an `Artist` (eg 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 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, eg 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. Eg:

```
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, eg 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 (eg all the points in the line that are within the specified epsilon tolerance)

13.4.1 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:

```
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
    print 'onpick points:', zip(xdata[ind], ydata[ind])

fig.canvas.mpl_connect('pick_event', onpick)

plt.show()
```

13.4.2 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:

```
"""
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)
xs = np.mean(X, axis=1)
ys = 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(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()
```

CHAPTER

FOURTEEN

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.

Coordinate	Transformation Object	Description
`data`	`ax.transData`	The userland data coordinate system, controlled by the xlim and ylim
`axes`	`ax.transAxes`	The coordinate system of the `Axes`; (0,0) is bottom left of the axes, and (1,1) is top right of the axes
`figure`	`fig.transFigure`	The coordinate system of the `Figure`; (0,0) is bottom left of the figure, and (1,1) is top right of the figure
`display`	`None`	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

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.

14.1 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.

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 your 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])
```

14.2 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, eg. 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.

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.

14.3 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.

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()
```

14.4 Using offset transforms to create a shadow effect

One use of transformations is to create a new transformation that is offset from another transformation, eg 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:

```
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:

```
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 <http://en.wikipedia.org/wiki/Point_%28typography%29>'_ 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.

14.5 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, eg. 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`:

```
self.transData = self.transScale + self.transProjection + \
    (self.transProjectionAffine + self.transAxes)
```

`transProjection` handles the projection from the space, eg. 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-custom_projection_example*.

CHAPTER

FIFTEEN

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

The following path codes are recognized

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

15.1 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

15.2 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, eg 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, eg 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:

```
# 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:

```
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:

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

Here is the result

CHAPTER

SIXTEEN

ANNOTATING AXES

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

16.1 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.

```
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:

```
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 *set_boxstyle* method.

```
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.

Class	Name	Attrs
LArrow	`larrow`	pad=0.3
RArrow	`rarrow`	pad=0.3
Round	`round`	pad=0.3,rounding_size=None
Round4	`round4`	pad=0.3,rounding_size=None
Roundtooth	`roundtooth`	pad=0.3,tooth_size=None
Sawtooth	`sawtooth`	pad=0.3,tooth_size=None
Square	`square`	pad=0.3

Note that the attributes 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")
```

16.2 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.

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

```
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 (*shirnkA* & *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 following styles are available.

Name	Attrs
`angle`	angleA=90,angleB=0,rad=0.0
`angle3`	angleA=90,angleB=0
`arc`	angleA=0,angleB=0,armA=None,armB=None,rad=0.0
`arc3`	rad=0.0
`bar`	armA=0.0,armB=0.0,fraction=0.3,angle=None

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 option only can 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>
<tr><th>Name</th><th>Attrs</th></tr>
<tr><td><code>-</code></td><td>None</td></tr>
<tr><td><code>-></code></td><td>head_length=0.4,head_width=0.2</td></tr>
<tr><td><code>-[</code></td><td>widthB=1.0,lengthB=0.2,angleB=None</td></tr>
<tr><td><code>|-|</code></td><td>widthA=1.0,widthB=1.0</td></tr>
<tr><td><code>-|></code></td><td>head_length=0.4,head_width=0.2</td></tr>
<tr><td><code><-</code></td><td>head_length=0.4,head_width=0.2</td></tr>
<tr><td><code><-></code></td><td>head_length=0.4,head_width=0.2</td></tr>
<tr><td><code><|-</code></td><td>head_length=0.4,head_width=0.2</td></tr>
<tr><td><code><|-|></code></td><td>head_length=0.4,head_width=0.2</td></tr>
<tr><td><code>fancy</code></td><td>head_length=0.4,head_width=0.4,tail_width=0.4</td></tr>
<tr><td><code>simple</code></td><td>head_length=0.5,head_width=0.5,tail_width=0.2</td></tr>
<tr><td><code>wedge</code></td><td>tail_width=0.3,shrink_factor=0.5</td></tr>
</table>

Some arrowstyles only work with connection style that generates a quadratic-spline segment. They are `fancy`, `simple`, and `wedge`. For these arrow styles, you must use "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.

16.3 Placing Artist at the anchored location of the Axes

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

```
from mpl_toolkits.axes_grid.anchored_artists import AnchoredText
at = AnchoredText("Figure 1a",
                  prop=dict(size=8), frameon=True,
                  loc=2,
                  )
at.patch.set_boxstyle("round,pad=0.,rounding_size=0.2")
ax.add_artist(at)
```

The *loc* keyword has 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 pixel). And user can add arbitrary artist to the drawing area. Note that the extents of the artists that are added to the drawing area has nothing to do with the placement of the drawing area itself. The initial size only matters.

```
from mpl_toolkits.axes_grid.anchored_artists import AnchoredDrawingArea

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

The artists that are added to the drawing area should not have transform set (they will be overridden) and the dimension of those artists are interpreted as a pixel coordinate, i.e., the radius of the circles in above example are 10 pixel and 5 pixel, respectively.

Sometimes, you want to your artists scale with data coordinate (or other coordinate than canvas pixel). 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.

```
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 corresponds to 0.1 and 0.4 in data coordinate 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).

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

16.4 Using Complex Coordinate with Annotation

The Annotation in matplotlib support several types of coordinate as described in *Annotating text*. For an advanced user who wants more control, it supports a few other options.

1. `Transform` instance. For example,

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

   is identical to

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

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

   ```
   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.

   ```
   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="->"))
   ```

   Note that it is your responsibility that the extent of the coordinate artist (*an1* in above example) is determined before *an2* gets drawn. In most cases, it means that an2 needs to be drawn later than *an1*.

3. A callable object that returns an instance of either `BboxBase` or `Transform`. If a transform is returned, it is same as 1 and if bbox is returned, it is same as 2. The callable object should take a single argument of renderer instance. For example, following two commands give identical results

```
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 specification. The first item is for x-coordinate and the second is for y-coordinate. For example,

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

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

5. Sometimes, you want your annotation with some "offset points", but not from the annotated point but from other point. `OffsetFrom` is a helper class for such case.

   You may take a look at this example *pylab_examples-annotation_demo3*.

16.5 Using ConnectorPatch

The ConnectorPatch is like an annotation without a text. While the annotate function is recommended in most of situation, 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 data coordinate of `ax1` to point xy int data coordinate of `ax2`. Here is a simple example.

While the ConnectorPatch instance can be added to any axes, but you may want it to be added to the axes in the latter (?) of the axes drawing order to prevent overlap (?) by other axes.

16.5.1 Advanced Topics

16.6 Zoom effect between Axes

mpl_toolkits.axes_grid.inset_locator defines some patch classes useful for interconnect two axes. Understanding the code requires some knowledge of how mpl's transform works. But, utilizing it will be straight forward.

16.7 Define Custom BoxStyle

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

```
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-ration 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.

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

CHAPTER

SEVENTEEN

OUR FAVORITE RECIPES

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.

17.1 Sharing axis limits and views

It's common to make two or more plots which share an axis, eg 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)

In [100]: ax2.plot(t, np.sin(4*np.pi*t))
Out[100]: [<matplotlib.lines.Line2D object at 0xb7d8fec>]
```

17.2 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. Eg

```
# 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 off x and y sharing for the whole bunch. You can either unpack the axes individually:

```
# 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:

```
# new style method 2; use an axes array
fig, axs = plt.subplots(2, 2, sharex=True, sharey=True)
axs[0,0].plot(x)
```

17.3 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', '|O4'), ('', '|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,

```
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.

Another annoyance is that if you hover the mouse over a 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, eg “Dec 2004”. What we’d like is for the location in the toolbar to have a higher degree of precision, eg 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.

Now when you hover your mouse over the plotted data, you’ll see date format strings like 2004-12-01 in the toolbar.

17.4 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, eg 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.

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.

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 showsn 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 shade that region blue.

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-axhspan_demo*.

17.5 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:

```
ax.legend(loc='upper right')
```

Other times you don't know where your data is, and loc='best' will try and place the legend:

```
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.

17.6 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.

CHAPTER

EIGHTEEN

SCREENSHOTS

Here you will find a host of example figures with the code that generated them

18.1 Simple Plot

The most basic `plot()`, with text labels

18.2 Subplot demo

Multiple regular axes (numrows by numcolumns) are created with the `subplot()` command.

18.3 Histograms

The `hist()` command automatically generates histograms and will return the bin counts or probabilities

18.4 Path demo

You can add aribitrary paths in matplotlib as of release 0.98. See the `matplotlib.path`.

18.5 mplot3d

The mplot3d toolkit (see *mplot3d tutorial* and *mplot3d-examples-index*) has support for simple 3d graphs including surface, wireframe, scatter, and bar charts (added in matlpotlib-0.99). Thanks to

John Porter, Jonathon Taylor and Reinier Heeres for the mplot3d toolkit. The toolkit is included with all standard matplotlib installs.

18.6 Ellipses

In support of the Phoenix mission to Mars, which used matplotlib in ground tracking of the spacecraft, Michael Droettboom built on work by Charlie Moad to provide an extremely accurate 8-spline approximation to elliptical arcs (see `Arc`) in the viewport. This provides a scale free, accurate graph of the arc regardless of zoom level

18.7 Bar charts

The `bar()` command takes error bars as an optional argument. You can also use up and down bars, stacked bars, candlestick bars, etc, ... See bar_stacked.py for another example. You can make horizontal bar charts with the `barh()` command.

18.8 Pie charts

The `pie()` command uses a MATLAB compatible syntax to produce pie charts. Optional features include auto-labeling the percentage of area, exploding one or more wedges out from the center of the pie, and a shadow effect. Take a close look at the attached code that produced this figure; nine lines of code.

18.9 Table demo

The `table()` command will place a text table on the axes

18.10 Scatter demo

The `scatter()` command makes a scatter plot with (optional) size and color arguments. This example plots changes in Google stock price from one day to the next with the sizes coding trading volume and the colors coding price change in day i. Here the alpha attribute is used to make semitransparent circle markers with the Agg backend (see *What is a backend?*)

18.11 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`

18.12 Fill demo

The `fill()` command lets you plot filled polygons. Thanks to Andrew Straw for providing this function

18.13 Date demo

You can plot date data with major and minor ticks and custom tick formatters for both the major and minor ticks; see matplotlib.ticker and matplotlib.dates for details and usage.

18.14 Financial charts

You can make much more sophisticated financial plots. This example emulates one of the ChartDirector financial plots. Some of the data in the plot, are real financial data, some are random traces that I used since the goal was to illustrate plotting techniques, not market analysis!

18.15 Basemap demo

Jeff Whitaker's *Basemap* add-on toolkit makes it possible to plot data on many different map projections. This example shows how to plot contours, markers and text on an orthographic projection, with NASA's "blue marble" satellite image as a background.

18.16 Log plots

The `semilogx()`, `semilogy()` and `loglog()` functions generate log scaling on the respective axes. The lower subplot uses a base10 log on the xaxis and a base 4 log on the yaxis. Thanks to Andrew Straw, Darren Dale and Gregory Lielens for contributions to the log scaling infrastructure.

18.17 Polar plots

The polar() command generates polar plots.

18.18 Legends

The legend() command automatically generates figure legends, with MATLAB compatible legend placement commands. Thanks to Charles Twardy for input on the legend command

18.19 Mathtext_examples

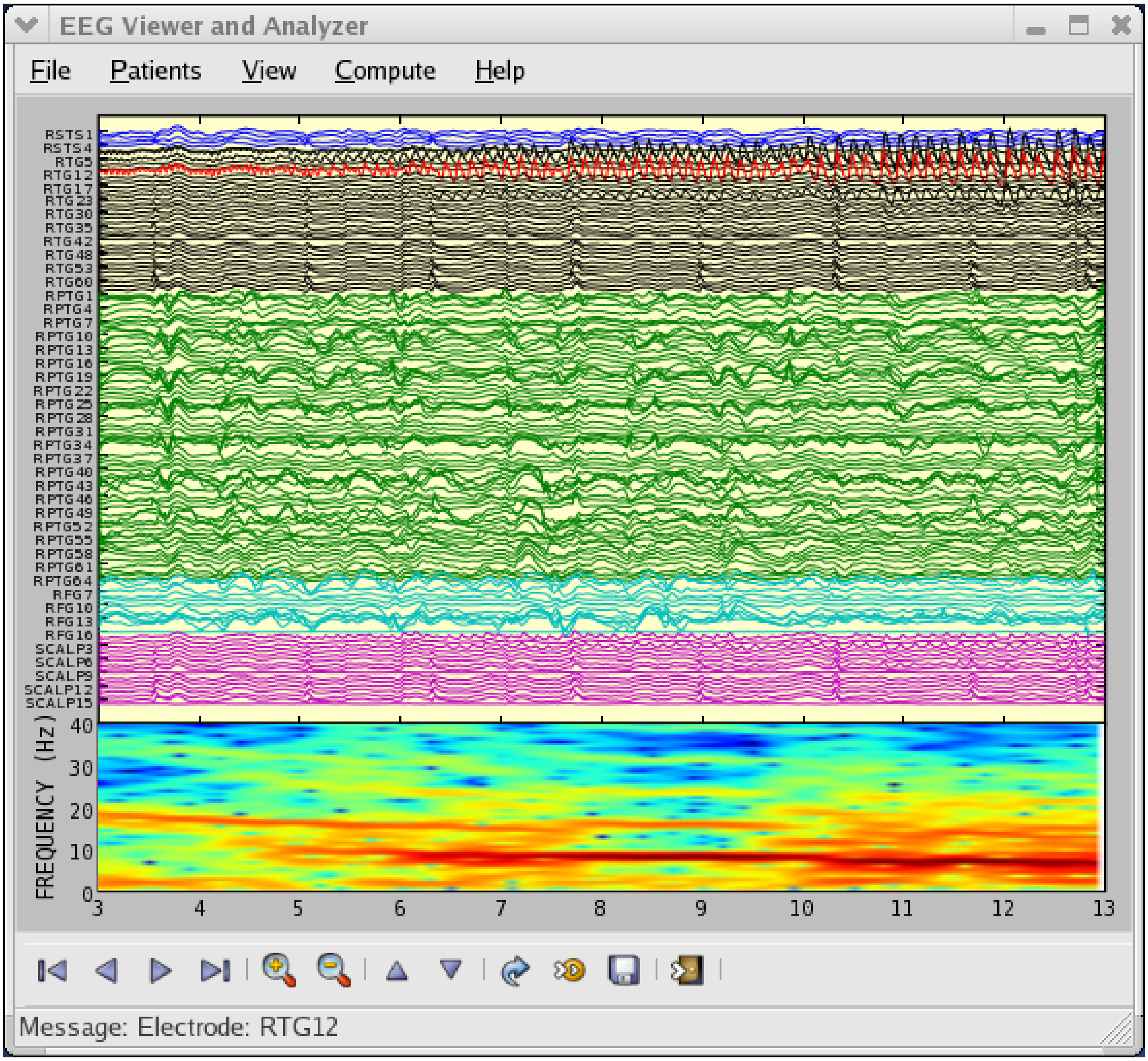
A sampling of the many TeX expressions now supported by matplotlib's internal mathtext engine. The mathtext module provides TeX style mathematical expressions using freetype2 and the BaKoMa computer modern or STIX fonts. See the matplotlib.mathtext module for additional. matplotlib mathtext is an independent implementation, and does not required TeX or any external packages installed on your computer. See the tutorial at *Writing mathematical expressions*.

18.20 Native TeX rendering

Although matplotlib's internal math rendering engine is quite powerful, sometimes you need TeX, and matplotlib supports external TeX rendering of strings with the *usetex* option.

18.21 EEG demo

You can embed matplotlib into pygtk, wxpython, Tk, FLTK or Qt applications. Here is a screenshot of an eeg viewer called pbrain which is part of the NeuroImaging in Python suite NIPY. Pbrain is written in pygtk using matplotlib. The lower axes uses specgram() to plot the spectrogram of one of the EEG channels. For an example of how to use the navigation toolbar in your applications, see *user_interfaces-embedding_in_gtk2*. If you want to use matplotlib in a wx application, see *user_interfaces-embedding_in_wx2*. If you want to work with glade, see *user_interfaces-mpl_with_glade*.


EEG Viewer and Analyzer
File
Patients
View
Compute
Help
FREQUENCY (Hz)
Message: Electrode: RTG12

CHAPTER

NINETEEN

WHAT'S NEW IN MATPLOTLIB

This page just covers the highlights – for the full story, see the CHANGELOG

For a list of all of the issues and pull requests since the last revision, see the *Github stats*.

Note: Matplotlib version 1.1 is the last major release compatible with Python versions 2.4 to 2.7. matplotlib 1.2 and later require versions 2.6, 2.7, and 3.1 and higher.

19.1 new in matplotlib-1.2

19.1.1 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.

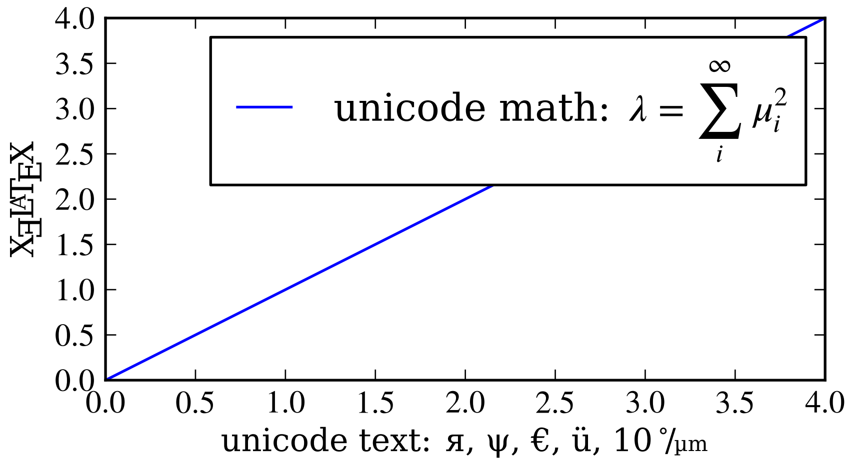
This work was done by Michael Droettboom, the Cape Town Python Users' Group, many others and supported financially in part by the SAGE project.

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.

19.1.2 PGF/TikZ backend

Peter Würtz wrote a backend that allows matplotlib to export figures as drawing commands for LaTeX that can be processed by PdfLaTeX, XeLaTeX or LuaLaTeX using the PGF/TikZ package. Usage examples and documentation are found in *Typesetting With XeLaTeX/LuaLaTeX*.



19.1.3 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:

```
>>> import matplotlib.ticker as mticker
>>> locator = mticker.MaxNLocator(nbins=5)
>>> print(locator.tick_values(10, 19.5))
[ 10.  12.  14.  16.  18.  20.]
```

19.1.4 Tri-Surface Plots

Damon McDougall added a new plotting method for the `mplot3d` toolkit called `plot_trisurf()`.

19.1.5 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.

19.1.6 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.

19.1.7 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.

19.1.8 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.

19.1.9 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:

```
>>> mpl.rc_file('mpl.rc')
```

To temporarily use RC parameters:

```
>>> with mpl.rc_context(fname='mpl.rc', rc={'text.usetex': True}):
>>>     ...
```

19.1.10 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.

19.1.11 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.

19.1.12 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

19.1.13 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.

19.1.14 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.

19.1.15 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.

19.2 new in matplotlib-1.1

19.2.1 Sankey Diagrams

Kevin Davies has extended Yannick Copin's original Sankey example into a module (`sankey`) and provided new examples (*api-sankey_demo_basics*, *api-sankey_demo_links*, *api-sankey_demo_rankine*).

19.2.2 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 *examples-index* listings. Examples using the new framework are in *animation-examples-index*; 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.

19.2.3 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 philosphy 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

```
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.

19.2.4 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.

19.2.5 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 *Legend of Complex Plots* for more detailed explanation and examples.

19.2.6 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.

- `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:
- `contourf()` gains *zdir* and *offset* kwargs. You can now do this:

19.2.7 Numerix support removed

After more than two years of deprecation warnings, Numerix support has now been completely removed from matplotlib.

19.2.8 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.

19.2.9 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.
- 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.

19.3 new in matplotlib-1.0

19.3.1 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.

19.3.2 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.

19.3.3 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). Eg:

```
fig, axarr = plt.subplots(2, 2)
axarr[0,0].plot([1,2,3])   # upper, left
```

See *pylab_examples-subplots_demo* for several code examples.

19.3.4 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.

19.3.5 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.

19.3.6 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.

19.3.7 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()`.

19.3.8 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

19.3.9 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.

19.3.10 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.

19.4 new in matplotlib-0.99

19.4.1 New documentation

Jae-Joon Lee has written two new guides *Legend guide* and *Annotating Axes*. Michael Sarahan has written *Image tutorial*. John Hunter has written two new tutorials on working with paths and transformations: *Path Tutorial* and *Transformations Tutorial*.

19.4.2 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-index* and *mplot3d tutorial*

19.4.3 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-index* and *The Matplotlib AxesGrid Toolkit User's Guide*.

19.4.4 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-spine_placement_demo* and `matplotlib.spines.Spine`.

19.5 new in 0.98.4

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

19.5.1 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`.

19.5.2 Fancy annotations and arrows

Jae-Joon has added lot's of support to annotations for drawing fancy boxes and connectors in annotations. See `annotate()` and `BoxStyle`, `ArrowStyle`, and `ConnectionStyle`.

19.5.3 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:

```
> python myfile.py -dmacosx
```

19.5.4 psd amplitude scaling

Ryan May did a lot of work to rationalize the amplitude scaling of `psd()` and friends. See *pylab_examples-psd_demo2*. and *pylab_examples-psd_demo3*. The changes should increase MATLAB compatabililty and increase scaling options.

19.5.5 Fill between

Added a `fill_between()` function to make it easier to do shaded region plots in the presence of masked data. You can pass an *x* array and a *ylower* and *yupper* array to fill betweem, and an optional *where* argument which is a logical mask where you want to do the filling.

19.5.6 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 in the set_*style commands.  Return the list of
pprinted 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, eg 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.  Ther 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 vlaues as loc. -JJL

Improve checks for external dependencies, using subprocess
(instead of deprecated popen*) and distutils (for version
checking) - DSD

Reimplementaion of the legend which supports baseline alignement,
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 hypen 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, respecitively.  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

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 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
```

CHAPTER

TWENTY

GITHUB STATS

GitHub stats for 2012/06/30 - 2012/09/07 (tag: v1.1.1)

These lists are automatically generated, and may be incomplete or contain duplicates.

The following 71 authors contributed 1151 commits.

- Aaron Boushley
- Ahmet Bakan
- Amy
- Andrew Dawson
- Arnaud Gardelein
- Ben Gamari
- Ben Root
- Bradley M. Froehle
- Brett Graham
- Bussonnier Matthias
- C. Gohlke
- Christoph Dann
- Christoph Gohlke
- Corey Farwell
- Craig M
- Craig Tenney
- Damon McDougall
- Daniel Hyams

- Darren Dale
- David Huard
- Eric Firing
- Ezra Peisach
- Gellule Xg
- Graham Poulter
- Hubert Holin
- Ian Thomas
- Ignas Anikevicius (gns_ank)
- Jack (aka Daniel) Kelly
- Jack Kelly
- Jae-Joon Lee
- James R. Evans
- Jason Grout
- Jens H. Nielsen
- Joe Kington
- John Hunter
- Jonathan Waltman
- Jouni K. Seppänen
- Lance Hepler
- Marc Abramowitz
- Martin Spacek
- Matthew Emmett
- Matthias BUSSONNIER
- Michael Droettboom
- Michiel de Hoon
- Mike Kaufman
- Neil
- Nelle Varoquaux
- Nikolay Vyahhi

- Paul Ivanov
- Peter Würtz
- Phil Elson
- Piti Ongmongkolkul
- Robert Johansson
- Russell Owen
- Ryan May
- Simon Cross
- Stefan van der Walt
- Takafumi Arakaki
- Thomas A Caswell
- Thomas Kluyver
- Thomas Robitaille
- Tobias Hoppe
- Tony S Yu
- Zach Pincus
- bev-a-tron
- endolith
- goir
- mcelrath
- pelson
- pwuertz
- vbr

We closed a total of 349 issues, 123 pull requests and 226 regular issues; this is the full list (generated with the script `tools/github_stats.py`):

Pull Requests (123):

- PR #1168: PEP8 compliance on artist.py
- PR #1213: Include username in tempdir
- PR #1182: Bezier pep8
- PR #1206: README and links fixes

- PR #1192: Issue835 2: replacement for #835
- PR #1187: Add a *simple* arrow example
- PR #1120: FAIL: matplotlib.tests.test_transforms.test_pre_transform_plotting.test on Python 3.x
- PR #714: Initial rework of gen_gallery.py
- PR #1150: the affine matrix is calculated in the display coordinate for interpolation='none'
- PR #1145: Fix formatter reset when twin{x,y}() is called
- PR #1201: Fix typo in object-oriented API
- PR #1061: Add log option to Axes.hist2d
- PR #1125: Reduce object-oriented boilerplate for users
- PR #1195: Fixed pickle tests to use the BufferIO object for python3 support.
- PR #1198: Fixed python2.6 support (by removing use of viewvalues on a dict).
- PR #1197: Handled future division changes for python3 (fixes #1194).
- PR #1162: FIX nose.tools.assert_is is only supported with python2.7
- PR #803: Return arrow collection as 2nd argument of streamplot.
- PR #1189: BUG: Fix streamplot when velocity component is exactly zero.
- PR #1191: Small bugfixes to the new pickle support.
- PR #1146: Fix invalid transformation in InvertedSymmetricalLogTransform.
- PR #1169: Subplot.twin[xy] returns a Subplot instance
- PR #1183: FIX undefined elements were used at several places in the mlab module
- PR #498: get_sample_data still broken on v.1.1.x
- PR #1170: Uses tight_layout.get_subplotspec_list to check if all axes are compatible w/ tight_layout
- PR #1174: closes #1173 - backporting python2.7 subprocess's check_output to be abl...
- PR #1175: Pickling support added. Various whitespace fixes as a result of reading *lots* of code.
- PR #1098: suppress exception upon quitting with qt4agg on osx
- PR #1171: backend_pgf: handle OSError when testing for xelatex/pdflatex
- PR #1164: doc: note contourf hatching in whats_new.rst
- PR #1153: PEP8 on artist
- PR #1163: tight_layout: fix regression for figures with non SubplotBase Axes

- PR #1159: FIX assert_raises cannot be called with `with`
- PR #1160: backend_pgf: clarifications and fixes in documentation
- PR #1154: six inclusion for dateutil on py3 doesn't work
- PR #1149: Add Phil Elson's percentage histogram example
- PR #1158: FIX - typo in lib/matplotlib/testing/compare.py
- PR #1155: workaround for fixed dpi assumption in adjust_bbox_pdf
- PR #1142: What's New: Python 3 paragraph
- PR #1130: Fix writing pdf on stdout
- PR #832: MPLCONFIGDIR tries to be created in read-only home
- PR #1140: BUG: Fix fill_between when NaN values are present
- PR #1144: Added tripcolor whats_new section.
- PR #1010: Port part of errorfill from Tony Yu's mpltools.
- PR #1141: backend_pgf: fix parentheses typo
- PR #1114: Make grid accept alpha rcParam
- PR #1124: PGF backend, fix #1116, #1118 and #1128
- PR #983: Issues with dateutil and pytz
- PR #1133: figure.py: import warnings, and make imports absolute
- PR #1132: clean out obsolete matplotlibrc-related bits to close #1123
- PR #1131: Cleanup after the gca test.
- PR #563: sankey.add() has mutable defaults
- PR #731: Plot limit with transform
- PR #1107: Added %s support for labels.
- PR #774: Allow automatic use of tight_layout.
- PR #1122: DOC: Add streamplot description to What's New page
- PR #1111: Fixed transoffset example from failing.
- PR #840: Documentation Errors for specgram
- PR #1088: For a text artist, if it has a _bbox_patch associated with it, the contains test should reflect this.
- PR #986: Add texinfo build target in doc/make.py
- PR #1076: PGF backend for XeLaTeX/LuaLaTeX support

- PR #1090: External transform api
- PR #1108: Fix documentation warnings
- PR #861: Add rcfile function (which loads rc params from a given file).
- PR #1062: increased the padding on FileMovieWritter.frame_format_str
- PR #1100: Doc multi version master
- PR #1105: Fixed comma between tests.
- PR #1095: Colormap byteorder bug
- PR #1103: colorbar: correct error introduced in commit 089024; closes #1102
- PR #1067: Support multi-version documentation on the website
- PR #1031: Added 'capthick' kwarg to errorbar()
- PR #1074: Added broadcasting support in some mplot3d methods
- PR #1064: Locator interface
- PR #850: Added tripcolor triangle-centred colour values.
- PR #1093: Exposed the callback id for the default key press handler so that it can be easily diabled. Fixes #215.
- PR #1065: fixed conversion from pt to inch in tight_layout
- PR #1082: doc: in pcolormesh docstring, say what it does.
- PR #1078: doc: note that IDLE doesn't work with interactive mode.
- PR #1071: patches.polygon: fix bug in handling of path closing, #1018.
- PR #1057: Contour norm scaling
- PR #1056: Test framework cleanups
- PR #778: Make tests faster
- PR #1024: broken links in the gallery
- PR #1054: stix_fonts_demo.py fails with bad refcount
- PR #960: Fixed logformatting for non integer bases.
- PR #897: GUI icon in Tkinter
- PR #1053: Move Python 3 import of reload() to the module that uses it
- PR #1049: Update examples/user_interfaces/embedding_in_wx2.py
- PR #1050: Update examples/user_interfaces/embedding_in_wx4.py
- PR #1051: Update examples/user_interfaces/mathtext_wx.py

- PR #1052: Update examples/user_interfaces/wxcursor_demo.py
- PR #1047: Enable building on Python 3.3 for Windows
- PR #1036: Move all figures to the front with a non-interactive show() in macosx backend.
- PR #1042: Three more plot_directive configuration options
- PR #1022: contour: map extended ranges to "under" and "over" values
- PR #1007: modifying GTK3 example to use pygobject, and adding a simple example to demonstrate NavigationToolbar in GTK3
- PR #1004: Added savefig.bbox option to matplotlibrc
- PR #976: Fix embedding_in_qt4_wtoolbar.py on Python 3
- PR #1034: MdH = allow compilation on recent Mac OS X without compiler warnings
- PR #1028: Fix use() so that it is possible to reset the rcParam.
- PR #1033: Py3k: reload->imp.reload
- PR #1002: Fixed potential overflow exception in the lines.contains() method
- PR #1025: Timers
- PR #989: Animation subprocess bug
- PR #898: Added warnings for easily confusible subplot/subplots invokations
- PR #963: Add detection of file extension for file-like objects
- PR #973: Fix sankey.py pep8 and py3 compatibility
- PR #972: Force closing PIL image files
- PR #981: Fix pathpatch3d_demo.py on Python 3
- PR #980: Fix basic_units.py on Python 3. PEP8 and PyLint cleanup.
- PR #1014: qt4: remove duplicate file save button; and remove trailing whitespace
- PR #1011: fix for bug #996 and related issues
- PR #985: support current and future FreeBSD releases
- PR #1000: Fix traceback for vlines/hlines, when an empty list or array passed in for x/y.
- PR #994: Fix bug in pcolorfast introduced by #901
- PR #993: Fix typo
- PR #908: use Qt window title as default savefig filename
- PR #971: Close fd temp file following rec2csv_bad_shape test
- PR #851: Simple GUI interface enhancements

- PR #979: Fix test_mouseclicks.py on Python 3
- PR #977: Fix lasso_selector_demo.py on Python 3
- PR #970: Fix tiff and jpeg export via PIL
- PR #961: Issue 807 auto minor locator

Issues (226):

- #1096: Documentation bug: pyplot.arrow does not list enough keywords to successfully draw an arrow
- #1168: PEP8 compliance on artist.py
- #1213: Include username in tempdir
- #1182: Bezier pep8
- #1177: Handled baseline image folder identification for non matplotlib projects.
- #1091: Update README.txt for v1.2
- #1206: README and links fixes
- #1192: Issue835 2: replacement for #835
- #1187: Add a *simple* arrow example
- #1120: FAIL: matplotlib.tests.test_transforms.test_pre_transform_plotting.test on Python 3.x
- #835: add documentation for figure show method in backend_bases and backend_template
- #714: Initial rework of gen_gallery.py
- #1150: the affine matrix is calculated in the display coordinate for interpolation='none'
- #1087: Update whats new section
- #385: BUG: plot_directive: look for plot script files relative to the .rst file
- #1110: twiny overrides formatter and adds another x-axis
- #1145: Fix formatter reset when twin{x,y}() is called
- #547: undocumented scatter marker definition change
- #1201: Fix typo in object-oriented API
- #1061: Add log option to Axes.hist2d
- #1094: Feature request - make it simpler to use full OO interface
- #1125: Reduce object-oriented boilerplate for users
- #1085: Images shifted relative to other plot feature in vector graphic output formats
- #1195: Fixed pickle tests to use the BufferIO object for python3 support.

- #1198: Fixed python2.6 support (by removing use of viewvalues on a dict).
- #1194: Streamplot result python version dependent
- #1197: Handled future division changes for python3 (fixes #1194).
- #557: Crash during date axis setup
- #600: errorbar(): kwarg 'markevery' not working as expected.
- #174: Memory leak in example simple_idle_wx.py
- #232: format in plot_direcitive sphinx>=1.0.6 compatible patch
- #1162: FIX nose.tools.assert_is is only supported with python2.7
- #1165: tight_layout fails on twinx, twiny
- #803: Return arrow collection as 2nd argument of streamplot.
- #1189: BUG: Fix streamplot when velocity component is exactly zero.
- #1191: Small bugfixes to the new pickle support.
- #323: native format for figures
- #1146: Fix invalid transformation in InvertedSymmetricalLogTransform.
- #1169: Subplot.twin[xy] returns a Subplot instance
- #1183: FIX undefined elements were used at several places in the mlab module
- #498: get_sample_data still broken on v.1.1.x
- #1170: Uses tight_layout.get_subplotspec_list to check if all axes are compatible w/ tight_layout
- #1173: The PGF backend only works on python2.7 and +
- #1174: closes #1173 - backporting python2.7 subprocess's check_output to be abl...
- #1175: Pickling support added. Various whitespace fixes as a result of reading *lots* of code.
- #1179: Attempt at making travis output shorter.
- #1020: Picklable figures
- #1098: suppress exception upon quitting with qt4agg on osx
- #1171: backend_pgf: handle OSError when testing for xelatex/pdflatex
- #1164: doc: note contourf hatching in whats_new.rst
- #606: Unable to configure grid using axisartist
- #1153: PEP8 on artist
- #1163: tight_layout: fix regression for figures with non SubplotBase Axes

- #1117: ERROR: matplotlib.tests.test_axes.test_contour_colorbar.test fails on Python 3
- #1159: FIX assert_raises cannot be called with `with`
- #206: hist normed=True problem?
- #1160: backend_pgf: clarifications and fixes in documentation
- #1154: six inclusion for dateutil on py3 doesn't work
- #320: hist plot in percent
- #1149: Add Phil Elson's percentage histogram example
- #1158: FIX - typo in lib/matplotlib/testing/compare.py
- #1135: Problems with bbox_inches='tight'
- #1155: workaround for fixed dpi assumption in adjust_bbox_pdf
- #1142: What's New: Python 3 paragraph
- #1138: tight_bbox made assumptions about the display-units without checking the figure
- #1130: Fix writing pdf on stdout
- #832: MPLCONFIGDIR tries to be created in read-only home
- #1140: BUG: Fix fill_between when NaN values are present
- #1144: Added tripcolor whats_new section.
- #1010: Port part of errorfill from Tony Yu's mpltools.
- #1141: backend_pgf: fix parentheses typo
- #1114: Make grid accept alpha rcParam
- #1118: ERROR: matplotlib.tests.test_backend_pgf.test_pdflatex on Python 3.x
- #1116: ERROR: matplotlib.tests.test_backend_pgf.test_xelatex
- #1124: PGF backend, fix #1116, #1118 and #1128
- #745: Cannot run tests with Python 3.x on MacOS 10.7
- #983: Issues with dateutil and pytz
- #1137: PGF/Tikz: savefig could not handle a filename
- #1128: PGF back-end fails on simple graph
- #1133: figure.py: import warnings, and make imports absolute
- #1123: Rationalize the number of ancillary (default matplotlibrc) files
- #1132: clean out obsolete matplotlibrc-related bits to close #1123
- #1131: Cleanup after the gca test.

- #563: sankey.add() has mutable defaults
- #238: patch for qt4 backend
- #731: Plot limit with transform
- #1107: Added %s support for labels.
- #720: Bug with bbox_inches='tight'
- #1084: doc/mpl_examples/pylab_examples/transoffset.py not working as expected
- #774: Allow automatic use of tight_layout.
- #1122: DOC: Add streamplot description to What's New page
- #1111: Fixed transoffset example from failing.
- #840: Documentation Errors for specgram
- #1088: For a text artist, if it has a _bbox_patch associated with it, the contains test should reflect this.
- #1119: ERROR: matplotlib.tests.test_image.test_imread_pil_uint16 on Python 3.x
- #353: Improved output of text in SVG and PDF
- #291: size information from print_figure
- #986: Add texinfo build target in doc/make.py
- #1076: PGF backend for XeLaTeX/LuaLaTeX support
- #1090: External transform api
- #1108: Fix documentation warnings
- #811: Allow tripcolor to directly plot triangle-centered functions
- #1005: imshow with big-endian data types on OS X
- #892: Update animation.py docstrings to "raw" Sphinx format
- #861: Add rcfile function (which loads rc params from a given file).
- #988: Trim white spaces while saving from Navigation toolbar
- #670: Add a printer button to the navigation toolbar
- #1062: increased the padding on FileMovieWritter.frame_format_str
- #188: MacOSX backend brings up GUI unnecessarily
- #1041: make.osx SDK location needs updating
- #1043: Fix show command for Qt backend to raise window to top
- #1046: test failing on master

- #962: Bug with figure.savefig(): using AGG, PIL, JPG and StringIO
- #1045: 1.1.1 not in pypi
- #1100: Doc multi version master
- #1106: Published docs for v1.1.1 missing pyplot.polar
- #569: 3D bar graphs with variable depth
- #359: new plot style: stackplot
- #297: pip/easy_install installs old version of matplotlib
- #152: Scatter3D: arguments (c,s,...) are not taken into account
- #1105: Fixed comma between tests.
- #1095: Colormap byteorder bug
- #1102: examples/pylab_examples/contour_demo.py fails
- #1103: colorbar: correct error introduced in commit 089024; closes #1102
- #1067: Support multi-version documentation on the website
- #1031: Added 'capthick' kwarg to errorbar()
- #1074: Added broadcasting support in some mplot3d methods
- #1032: Axesbase
- #1064: Locator interface
- #850: Added tripcolor triangle-centred colour values.
- #1059: Matplotlib figure window freezes during interactive mode
- #215: skipping mpl-axes-interaction during key_press_event's
- #1093: Exposed the callback id for the default key press handler so that it can be easily diabled. Fixes #215.
- #909: Log Formatter for tick labels can't handle non-integer base
- #1065: fixed conversion from pt to inch in tight_layout
- #1086: Problem with subplot / matplotlib.dates interaction
- #782: mplot3d: grid doesn't update after adding a slider to figure?
- #703: pcolormesh help not helpful
- #1082: doc: in pcolormesh docstring, say what it does.
- #1068: Add stairstep plotting functionality
- #1078: doc: note that IDLE doesn't work with interactive mode.

- #704: ignore case in `edgecolors` keyword in `pcolormesh` (and possibly other places)
- #708: set_clim not working with NonUniformImage
- #768: Add "tight_layout" button to toolbar
- #791: v1.1.1 release candidate testing
- #844: imsave/imshow and cmaps
- #939: test failure: matplotlib.tests.test_mathtext.mathfont_stix_14_test.test
- #875: Replace "jet" with "hot" as the default colormap
- #881: "Qualitative" colormaps represented as continuous
- #1072: For a text artist, if it has a _bbox_patch associated with it, the conta...
- #1071: patches.polygon: fix bug in handling of path closing, #1018.
- #1018: BUG: check for closed path in Polygon.set_xy()
- #1066: fix limit calculation of step* histogram
- #1073: Mplot3d/input broadcast
- #906: User-specified medians and conf. intervals in boxplots
- #899: Update for building matplotlib under Mac OS X 10.7 Lion and XCode > 4.2
- #1057: Contour norm scaling
- #1035: Added a GTK3 implementation of the SubplotTool window.
- #807: Crash when using zoom tools on a plot: AutoMinorLocator after MultipleLocator gives "ValueError: Need at least two major ticks to find minor tick locations"
- #1023: New button to toolbar for tight_layout.
- #1056: Test framework cleanups
- #778: Make tests faster
- #1048: some matplotlib examples incompatible with wxpython 2.9
- #1024: broken links in the gallery
- #1054: stix_fonts_demo.py fails with bad refcount
- #960: Fixed logformatting for non integer bases.
- #897: GUI icon in Tkinter
- #1053: Move Python 3 import of reload() to the module that uses it
- #1049: Update examples/user_interfaces/embedding_in_wx2.py
- #1050: Update examples/user_interfaces/embedding_in_wx4.py

- #1051: Update examples/user_interfaces/mathtext_wx.py
- #1052: Update examples/user_interfaces/wxcursor_demo.py
- #1047: Enable building on Python 3.3 for Windows
- #819: Add new plot style: stackplot
- #1036: Move all figures to the front with a non-interactive show() in macosx backend.
- #1042: Three more plot_directive configuration options
- #1044: plots not being displayed in OSX 10.8
- #1022: contour: map extended ranges to "under" and "over" values
- #1007: modifying GTK3 example to use pygobject, and adding a simple example to demonstrate NavigationToolbar in GTK3
- #1004: Added savefig.bbox option to matplotlibrc
- #976: Fix embedding_in_qt4_wtoolbar.py on Python 3
- #1013: compilation warnings in _macosx.m
- #1034: MdH = allow compilation on recent Mac OS X without compiler warnings
- #964: Animation clear_temp=False reuses old frames
- #1028: Fix use() so that it is possible to reset the rcParam.
- #1033: Py3k: reload->imp.reload
- #1002: Fixed potential overflow exception in the lines.contains() method
- #1025: Timers
- #989: Animation subprocess bug
- #898: Added warnings for easily confusible subplot/subplots invokations
- #963: Add detection of file extension for file-like objects
- #973: Fix sankey.py pep8 and py3 compatibility
- #972: Force closing PIL image files
- #981: Fix pathpatch3d_demo.py on Python 3
- #980: Fix basic_units.py on Python 3. PEP8 and PyLint cleanup.
- #996: macosx backend broken by #901: QuadMesh fails so colorbar fails
- #1017: axes.Axes.step() function not documented
- #1014: qt4: remove duplicate file save button; and remove trailing whitespace
- #655: implement path_effects for Line2D objects

- #999: pcolormesh edgecolor of "None"
- #1011: fix for bug #996 and related issues
- #1009: Simplify import statement
- #982: Supported FreeBSD10 as per #225
- #225: Add support for FreeBSD >6.x
- #985: support current and future FreeBSD releases
- #1006: MacOSX backend throws exception when plotting a quadmesh
- #1000: Fix traceback for vlines/hlines, when an empty list or array passed in for x/y.
- #1001: Bug fix for issue #955
- #994: Fix bug in pcolorfast introduced by #901
- #993: Fix typo
- #908: use Qt window title as default savefig filename
- #830: standard key for closing figure ("q")
- #971: Close fd temp file following rec2csv_bad_shape test
- #851: Simple GUI interface enhancements
- #979: Fix test_mouseclicks.py on Python 3
- #977: Fix lasso_selector_demo.py on Python 3
- #970: Fix tiff and jpeg export via PIL
- #707: key_press_event in pyqt4 embedded matplotlib
- #243: Debug version/symbols for win32
- #255: Classes in _transforms.h in global namespace
- #961: Issue 807 auto minor locator
- #345: string symbol markers ("scattertext" plot)
- #247: DLL load failed
- #808: pip install matplotlib fails
- #168: setupext.py incorrect for Mac OS X
- #213: Fixing library path in setupext.py for Mac

CHAPTER

TWENTYONE

LICENSE

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 (eg LGPL) are acceptable in matplotlib *toolkits-index*. For a discussion of the motivations behind the licencing choice, see *Licenses*.

21.1 License agreement for matplotlib 1.3.x

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 1.3.x 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 1.3.x 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 1.3.x 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 1.3.x.

4. JDH is making matplotlib 1.3.x 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 1.3.x WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. JDH SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF MATPLOTLIB 1.3.x FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A

RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING MATPLOTLIB 1.3.x, 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 1.3.x, Licensee agrees to be bound by the terms and conditions of this License Agreement.

CHAPTER

TWENTYTWO

CREDITS

matplotlib was written by John Hunter and is now developed and maintained by a number of active developers. The current lead developer of matplotlib is Michael Droettboom.

Special thanks to those who have made valuable contributions (roughly in order of first contribution by date). Any list like this is bound to be incomplete and can't capture the thousands and thousands of contributions over the years from these and others:

Jeremy O'Donoghue wrote the wx backend

Andrew Straw Provided much of the log scaling architecture, the fill command, PIL support for imshow, and provided many examples. He also wrote the support for dropped axis spines and the buildbot unit testing infrastructure which triggers the JPL/James Evans platform specific builds and regression test image comparisons from svn matplotlib across platforms on svn commits.

Charles Twardy provided the impetus code for the legend class and has made countless bug reports and suggestions for improvement.

Gary Ruben made many enhancements to errorbar to support x and y errorbar plots, and added a number of new marker types to plot.

John Gill wrote the table class and examples, helped with support for auto-legend placement, and added support for legending scatter plots.

David Moore wrote the paint backend (no longer used)

Todd Miller supported by STSCI contributed the TkAgg backend and the numerix module, which allows matplotlib to work with either numeric or numarray. He also ported image support to the postscript backend, with much pain and suffering.

Paul Barrett supported by STSCI overhauled font management to provide an improved, free-standing, platform independent font manager with a WC3 compliant font finder and cache mechanism and ported truetype and mathtext to PS.

Perry Greenfield supported by STSCI overhauled and modernized the goals and priorities page, implemented an improved colormap framework, and has provided many suggestions and a lot of insight to the overall design and organization of matplotlib.

Jared Wahlstrand wrote the initial SVG backend.

Steve Chaplin served as the GTK maintainer and wrote the Cairo and GTKCairo backends.

Jim Benson provided the patch to handle vertical mathttext.

Gregory Lielens provided the FltkAgg backend and several patches for the frontend, including contributions to toolbar2, and support for log ticking with alternate bases and major and minor log ticking.

Darren Dale

did the work to do mathtext exponential labeling for log plots, added improved support for scalar formatting, and did the lions share of the psfrag LaTeX support for postscript. He has made substantial contributions to extending and maintaining the PS and Qt backends, and wrote the site.cfg and matplotlib.conf build and runtime configuration support. He setup the infrastructure for the sphinx documentation that powers the mpl docs.

Paul Mcguire provided the pyparsing module on which mathtext relies, and made a number of optimizations to the matplotlib mathtext grammar.

Fernando Perez has provided numerous bug reports and patches for cleaning up backend imports and expanding pylab functionality, and provided matplotlib support in the pylab mode for ipython. He also provided the `matshow()` command, and wrote TConfig, which is the basis for the experimental traited mpl configuration.

Andrew Dalke of Dalke Scientific Software contributed the strftime formatting code to handle years earlier than 1900.

Jochen Voss served as PS backend maintainer and has contributed several bugfixes.

Nadia Dencheva

supported by STSCI provided the contouring and contour labeling code.

Baptiste Carvello provided the key ideas in a patch for proper shared axes support that underlies ganged plots and multiscale plots.

Jeffrey Whitaker at NOAA wrote the *Basemap* tookit

Sigve Tjoraand, Ted Drain, James Evans and colleagues at the JPL collaborated on the QtAgg backend and sponsored development of a number of features including custom unit types, datetime support, scale free ellipses, broken bar plots and more. The JPL team wrote the unit testing image comparison infrastructure for regression test image comparisons.

James Amundson did the initial work porting the qt backend to qt4

Eric Firing has contributed significantly to contouring, masked array, pcolor, image and quiver support, in addition to ongoing support and enhancements in performance, design and code quality in most aspects of matplotlib.

Daishi Harada added support for “Dashed Text”. See dashpointlabel.py and `TextWithDash`.

Nicolas Young added support for byte images to imshow, which are more efficient in CPU and memory, and added support for irregularly sampled images.

The brainvisa Orsay team and Fernando Perez added Qt support to ipython in pylab mode.

Charlie Moad contributed work to matplotlib's Cocoa support and has done a lot of work on the OSX and win32 binary releases.

Jouni K. Seppänen wrote the PDF backend and contributed numerous fixes to the code, to tex support and to the get_sample_data handler

Paul Kienzle improved the picking infrastruture for interactive plots, and with Alex Mont contributed fast rendering code for quadrilateral meshes.

Michael Droettboom supported by STSCI wrote the enhanced mathtext support, implementing Knuth's box layout algorithms, saving to file-like objects across backends, and is responsible for numerous bug-fixes, much better font and unicode support, and feature and performance enhancements across the matplotlib code base. He also rewrote the transformation infrastructure to support custom projections and scales.

John Porter, Jonathon Taylor and Reinier Heeres John Porter wrote the mplot3d module for basic 3D plotting in matplotlib, and Jonathon Taylor and Reinier Heeres ported it to the refactored transform trunk.

Jae-Joon Lee Implemented fancy arrows and boxes, rewrote the legend support to handle multiple columns and fancy text boxes, wrote the axes grid toolkit, and has made numerous contributions to the code and documentation

Paul Ivanov Has worked on getting matplotlib integrated better with other tools, such as Sage and IPython, and getting the test infrastructure faster, lighter and meaner. Listen to his podcast.

Tony Yu Has been involved in matplotlib since the early days, and recently has contributed stream plotting among many other improvements. He is the author of mpltools.

Michiel de Hoon Wrote and maintains the macosx backend.

Ian Thomas Contributed, among other things, the triangulation (tricolor and tripcontour) methods.

Benjamin Root Has significantly improved the capabilities of the 3D plotting. He has improved matplotlib's documentation and code quality throughout, and does invaluable triaging of pull requests and bugs.

Phil Elson Fixed some deep-seated bugs in the transforms framework, and has been laser-focused on improving polish throughout matplotlib, tackling things that have been considered to large and daunting for a long time.

Damon McDougall Added triangulated 3D surfaces and stack plots to matplotlib.

Part II

The Matplotlib FAQ

CHAPTER

TWENTYTHREE

INSTALLATION

Contents

23.1 Report a compilation problem

See *Report a problem*.

23.2 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:

```
from pylab import *
plot([1,2,3])
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 *Report a problem*.

23.3 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.

23.3.1 Easy Install

1. Delete the caches from your *.matplotlib configuration directory*.
2. Run:

   ```
   easy_install -m matplotlib
   ```

3. Delete any .egg files or directories from your *installation directory*.

23.3.2 Windows installer

1. Delete the caches from your *.matplotlib configuration directory*.
2. Use *Start* → *Control Panel* to start the **Add and Remove Software** utility.

23.3.3 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*.

23.4 How to Install

23.4.1 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 setupegg.py develop
```

This creates links in the right places and installs the command line script to the appropriate places.

Note: Mac OSX users please see the *Building on OSX* 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 setupegg.py develop` command again to compile them.

There is more information on *using git* in the developer docs.

23.5 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.

23.6 OS-X Notes

23.6.1 Which python for OS X?

Apple ships with its own python, and many users have had trouble with it. There are several alternative versions of python that can be used. If it is feasible, we recommend that you use the enthought python distribution EPD for OS X (which comes with matplotlib and much more). Also available is MacPython or the official OS X version from python.org.

Note: Before installing any of the binary packages, 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.

23.6.2 Installing OSX binaries

If you want to install matplotlib from one of the binary installers we build, you have two choices: a mpkg installer, which is a typical Installer.app, or a binary OSX egg, which you can install via setuptools' easy_install.

The mkpg installer will have a "zip" extension, and will have a name like `matplotlib-1.2.0-py2.7-macosx10.5_mpkg.zip`. The name of the installer depends on which versions of python, matplotlib, and OSX it was built for. You need to unzip this file using either the "unzip" command, or simply double clicking on the it. Then when you double-click on the resulting mpkd, which will have a name like `matplotlib-1.2.0-py2.7-macosx10.5.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/Python/2.7/site-packages/` (exact path depends on your python version). This directory may not be in your python 'path' variable, so you should test your installation with:

```
> python -c 'import matplotlib; print matplotlib.__version__, matplotlib.__file__'
```

If you get an error like:

```
Traceback (most recent call last):
  File "<string>", line 1, in <module>
ImportError: No module named matplotlib
```

then you will need to set your PYTHONPATH, eg:

```
export PYTHONPATH=/Library/Python/2.7/site-packages:$PYTHONPATH
```

See also ref:`environment-variables`.

23.6.3 easy_install from egg

You can also use the eggs we build for OSX (see the installation instructions for easy_install if you do not have it on your system already). You can try:

```
> easy_install matplotlib
```

which should grab the latest egg from github, but sometimes the naming conventions for OSX eggs can be broken (see below). Therefore, there is no guarantee the right egg will be found. We recommend you download the latest egg from our download site directly to your harddrive, and manually install it, eg:

```
> easy_install --install-dir=~/path/to/site-packages/  matplotlib-1.2.0-py2.7-macosx-10.5-i386
```

23.6.4 Building and installing from source on OSX with EPD

If you have the EPD installed (*Which python for OS X?*), it might turn out to be rather tricky to install a new version of matplotlib from source on the Mac OS 10.5 . Here's a procedure that seems to work, at least sometimes:

0. Remove the ~/.matplotlib folder ("rm -rf ~/.matplotlib").

1. Edit the file (make a backup before you start, just in case): `/Library/Frameworks/Python.framework/Versions/Current/lib/python2.5/config/Makefile`, removing all occurrences of the string `-arch ppc`, changing the line `MACOSX_DEPLOYMENT_TARGET=10.3` to `MACOSX_DEPLOYMENT_TARGET=10.5` and changing the occurrences of `MacOSX10.4u.sdk` into `MacOSX10.5.sdk`

2. In `/Library/Frameworks/Python.framework/Versions/Current/lib/pythonX.Y/site-packages/` (where X.Y is the version of Python you are building against) Comment out the line containing the name of the directory in which the previous version of MPL was installed (Looks something like `./matplotlib-0.98.5.2n2-py2.5-macosx-10.3-fat.egg`).

3. Save the following as a shell script, for example `./install-matplotlib-epd-osx.sh`:

```
NAME=matplotlib
VERSION=v1.1.x
PREFIX=$HOME
#branch="release"
branch="master"
git clone git://github.com/matplotlib/matplotlib.git
cd matplotlib
if [ $branch = "release" ]
    then
    echo getting the maintenance branch
    git checkout -b $VERSION origin/$VERSION
fi
export CFLAGS="-Os -arch i386"
export LDFLAGS="-Os -arch i386"
export PKG_CONFIG_PATH="/usr/x11/lib/pkgconfig"
export ARCHFLAGS="-arch i386"
python setup.py build
# use --prefix if you don't want it installed in the default location:
python setup.py install #--prefix=$PREFIX
cd ..
```

Run this script (for example `sh ./install-matplotlib-epd-osx.sh`) in the directory in which you want the source code to be placed, or simply type the commands in the terminal command line. This script sets some local variable (CFLAGS, LDFLAGS, PKG_CONFIG_PATH, ARCHFLAGS), removes previous installations, checks out the source from github, builds and installs it. The backend should to be set to MacOSX.

23.7 Windows Notes

23.7.1 Binary installers for Windows

If you have already installed python, you can use one of the matplotlib binary installers for windows – you can get these from the download site. Choose the files that match your version of python (eg `py2.7` if you installed Python 2.7) which have the `exe` extension. If you haven't already installed python, you can get the official version from the python web site.

There are also two packaged distributions of python that come preloaded with matplotlib and many other tools like ipython, numpy, scipy, vtk and user interface toolkits. These packages are quite large because they come with so much, but you get everything with a single click installer.

- The Enthought Python Distribution EPD
- python (x, y)

CHAPTER

TWENTYFOUR

USAGE

Contents

24.1 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.

24.2 Matplotlib, pylab, and pyplot: how are they related?

Matplotlib is the whole package; `pylab` is a module in matplotlib that gets installed alongside `matplotlib`; and `matplotlib.pyplot` is a module in matplotlib.

Pyplot provides the state-machine interface to the underlying plotting library in matplotlib. This means that figures and axes are implicitly and automatically created to achieve the desired plot. For example, calling `plot` from pyplot will automatically create the necessary figure and axes to achieve the desired plot. Setting a title will then automatically set that title to the current axes object:

```
import matplotlib.pyplot as plt

plt.plot(range(10), range(10))
plt.title("Simple Plot")
plt.show()
```

Pylab combines the pyplot functionality (for plotting) with the numpy functionality (for mathematics and for working with arrays) in a single namespace, making that namespace (or environment) even more MATLAB-like. For example, one can call the `sin` and `cos` functions just like you could in MATLAB, as well as having all the features of pyplot.

The pyplot interface is generally preferred for non-interactive plotting (i.e., scripting). The pylab interface is convenient for interactive calculations and plotting, as it minimizes typing. Note that this is what you get if you use the *ipython* shell with the *-pylab* option, which imports everything from pylab and makes plotting fully interactive.

24.3 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 preferred pyplot style, the imports at the top of your scripts will typically be:

```
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. So, a simple example in this style would be:

```
import matplotlib.pyplot as plt
import numpy as np
x = np.arange(0, 10, 0.2)
y = np.sin(x)
plt.plot(x, y)
plt.show()
```

Note that this example used pyplot's state-machine to automatically and implicitly create a figure and an axes. For full control of your plots and more advanced usage, use the pyplot interface for creating figures, and then use the object methods for the rest:

```
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()
```

Next, the same example using a pure MATLAB-style:

```
from pylab import *
x = arange(0, 10, 0.2)
y = sin(x)
plot(x, y)
show()
```

So, why all the extra typing as one moves away from the pure MATLAB-style? 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.

24.4 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, ie 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, qt, macosx, or fltk; 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 a two primary ways to configure your backend. One is to set the `backend` parameter in your `matplotlibrc` file (see *Customizing matplotlib*):

```
backend : WXAgg   # use wxpython with antigrain (agg) rendering
```

The other is to use the matplotlib `use()` directive:

```
import matplotlib
matplotlib.use('PS')   # generate postscript output by default
```

If you use the `use` directive, this must be done before importing `matplotlib.pyplot` or `matplotlib.pylab`.

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 either of the two 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, eg `WXAgg`, `GTKAgg`, `QT4Agg`, `TkAgg`. In addition, some of the user interfaces support other rendering engines. For example, with GTK, you can also select GDK rendering (backend `GTK`) 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):

Renderer	Filetypes	Description
AGG	*png*	*raster graphics* – high quality images using the Anti-Grain Geometry engine
PS	*ps eps*	*vector graphics* – Postscript output
PDF	*pdf*	*vector graphics* – Portable Document Format
SVG	*svg*	*vector graphics* – Scalable Vector Graphics
Cairo	*png ps pdf svg* ...	*vector graphics* – Cairo graphics
GDK	*png jpg tiff* ...	*raster graphics* – the Gimp Drawing Kit

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:

Backend	Description
GTKAgg	Agg rendering to a *GTK* canvas (requires PyGTK)
GTK	GDK rendering to a *GTK* canvas (not recommended) (requires PyGTK)
GTKCairo	Cairo rendering to a *GTK* Canvas (requires PyGTK)
WXAgg	Agg rendering to to a *wxWidgets* canvas (requires wxPython)
WX	Native *wxWidgets* drawing to a *wxWidgets* Canvas (not recommended) (requires wxPython)
TkAgg	Agg rendering to a *Tk* canvas (requires TkInter)
QtAgg	Agg rendering to a *Qt* canvas (requires PyQt) (not recommended; use Qt4Agg)
Qt4Agg	Agg rendering to a *Qt4* canvas (requires PyQt4)
FLTKAgg	Agg rendering to a *FLTK* canvas (requires pyFLTK) (not widely used; consider TKAgg, GTKAgg, WXAgg, or QT4Agg instead)
macosx	Cocoa rendering in OSX windows (presently lacks blocking show() behavior when matplotlib is in non-interactive mode)

24.5 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.

24.5.1 Interactive example

From an ordinary python prompt, or after invoking ipython with no options, try this:

```
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:

```
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:

```
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()`:

```
plt.draw()
```

Now you should see the new line added to the plot.

24.5.2 Non-interactive example

Start a fresh session as in the previous example, but now turn interactive mode off:

```
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:

```
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:

```
import numpy as np
import matplotlib.pyplot as plt
plt.ioff()
for i in range(3):
    plt.plot(np.random.rand(10))
    plt.show()
```

which makes three plots, one at a time.

24.5.3 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

TWENTYFIVE

HOW-TO

Contents

25.1 Plotting: howto

25.1.1 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 (eg 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:

```
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:

```
import matplotlib.text as text
for o in fig.findobj(text.Text):
    o.set_fontstyle('italic')
```

25.1.2 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, eg 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*). Eg:

```
fig = plt.figure()
fig.patch.set_alpha(0.5)
ax = fig.add_subplot(111)
ax.patch.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, eg:

```
ax.plot(x, y, alpha=0.5)
ax.set_xlabel('volts', alpha=0.5)
```

25.1.3 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:

```
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:

```
plt.savefig(pp, format='pdf')
```

An easier way is to call `PdfPages.savefig`:

```
pp.savefig()
```

Finally, the multipage pdf object has to be closed:

```
pp.close()
```

25.1.4 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:

```
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:

```
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

***hspace* = 0.2** the amount of height reserved for white space between subplots

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:

```
ax = fig.add_axes([left, bottom, width, height])
```

where all values are in fractional (0 to 1) coordinates. See *pylab_examples-axes_demo* for an example of placing axes manually.

25.1.5 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 draw by connecting (`mpl_connect()`) to the "on_draw" signal (`DrawEvent`) and get the window extent there, and then do something with it, eg 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

25.1.6 Configure the tick linewidths

In matplotlib, the ticks are *markers*. All `Line2D` objects support a line (solid, dashed, etc) and a marker (circle, square, tick). The tick linewidth is controlled by the "markeredgewidth" property:

```
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*.

25.1.7 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.

25.1.8 Skip dates where there is no data

When plotting time series, eg financial time series, one often wants to leave out days on which there is no data, eg weekends. By passing in dates on the x-xaxis, you get large horizontal gaps on periods when there is not data. The solution is to pass in some proxy x-data, eg 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:

```
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')

fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(ind, r.adj_close, 'o-')
```

```
ax.xaxis.set_major_formatter(ticker.FuncFormatter(format_date))
fig.autofmt_xdate()

plt.show()
```

25.1.9 Test whether a point is inside a polygon

The nxutils provides two high-performance methods: for a single point use pnpoly() and for an array of points use points_inside_poly(). For a discussion of the implementation see pnpoly.

```
In [25]: import numpy as np

In [26]: import matplotlib.nxutils as nx

In [27]: verts = np.array([ [0,0], [0, 1], [1, 1], [1,0]], float)

In [28]: nx.pnpoly( 0.5, 0.5, verts)
Out[28]: 1

In [29]: nx.pnpoly( 0.5, 1.5, verts)
Out[29]: 0

In [30]: points = np.random.rand(10,2)*2

In [31]: points
Out[31]:
array([[ 1.03597426,  0.61029911],
       [ 1.94061056,  0.65233947],
       [ 1.08593748,  1.16010789],
       [ 0.9255139 ,  1.79098751],
       [ 1.54564936,  1.15604046],
       [ 1.71514397,  1.26147554],
       [ 1.19133536,  0.56787764],
       [ 0.40939549,  0.35190339],
       [ 1.8944715 ,  0.61785408],
       [ 0.03128518,  0.48144145]])

In [32]: nx.points_inside_poly(points, verts)
Out[32]: array([False, False, False, False, False, False, False,  True, False, True], dtype=bo
```

25.1.10 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:

```
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.

25.1.11 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:

```
ax = fig.add_subplot(111, aspect='equal')
```

25.1.12 Make a movie

If you want to take an animated plot and turn it into a movie, the best approach is to save a series of image files (eg PNG) and use an external tool to convert them to a movie. You can use mencoder, which is part of the mplayer suite for this:

```
#fps (frames per second) controls the play speed
mencoder 'mf://*.png' -mf type=png:fps=10 -ovc \\
   lavc -lavcopts vcodec=wmv2 -oac copy -o animation.avi
```

The swiss army knife of image tools, ImageMagick's convert works for this as well.

Here is a simple example script that saves some PNGs, makes them into a movie, and then cleans up:

```
import os, sys
import matplotlib.pyplot as plt

files = []
fig = plt.figure(figsize=(5,5))
ax = fig.add_subplot(111)
for i in range(50):  # 50 frames
    ax.cla()
    ax.imshow(rand(5,5), interpolation='nearest')
    fname = '_tmp%03d.png'%i
    print 'Saving frame', fname
    fig.savefig(fname)
    files.append(fname)

print 'Making movie animation.mpg - this make take a while'
os.system("mencoder 'mf://_tmp*.png' -mf type=png:fps=10 \\
  -ovc lavc -lavcopts vcodec=wmv2 -oac copy -o animation.mpg")
```

25.1.13 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.

25.1.14 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:

```
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.

25.1.15 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 interperator 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:

```
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:

```
# 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`.

25.2 Contributing: howto

25.2.1 Submit a patch

See *Making patches* for information on how to make a patch with git.

If you are posting a patch to fix a code bug, please explain your patch in words – what was broken before and how you fixed it. Also, even if your patch is particularly simple, just a few lines or a single function replacement, we encourage people to submit git diffs against HEAD of the branch they are patching. It just makes life easier for us, since we (fortunately) get a lot of contributions, and want to receive them in a standard format. If possible, for any non-trivial change, please include a complete, free-standing example that the developers can run unmodified which shows the undesired behavior pre-patch and the desired behavior post-patch, with a clear verbal description of what to look for. A developer may have written the function you are working on years ago, and may no longer be with the project, so it is quite possible you are the world expert on the code you are patching and we want to hear as much detail as you can offer.

When emailing your patch and examples, feel free to paste any code into the text of the message, indeed we encourage it, but also attach the patches and examples since many email clients screw up the formatting of plain text, and we spend lots of needless time trying to reformat the code to make it usable.

You should check out the guide to developing matplotlib to make sure your patch abides by our coding conventions *The Matplotlib Developers' Guide*.

25.2.2 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 (or here for this document).

The sphinx website is a good resource for learning sphinx, but we have put together a cheat-sheet at *Documenting matplotlib* which shows you how to get started, and outlines the matplotlib conventions and extensions, eg 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 *Submit a patch*. Looking for something to do? Search for TODO.

25.3 Matplotlib in a web application server

Many users report initial problems trying to use maptlotlib 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:

```
# 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-agg_oo*.

You can either generate hardcopy on the filesystem by calling savefig:

```
# 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:

```
import sys
fig.savefig(sys.stdout)
```

Here is an example using the Python Imaging Library (PIL). First, the figure is saved to a StringIO object which is then fed to PIL for further processing:

```
import StringIO, Image
imgdata = StringIO.StringIO()
fig.savefig(imgdata, format='png')
imgdata.seek(0)  # rewind the data
im = Image.open(imgdata)
```

25.3.1 matplotlib with apache

TODO; see *Contribute to matplotlib documentation*.

25.3.2 matplotlib with django

TODO; see *Contribute to matplotlib documentation*.

25.3.3 matplotlib with zope

TODO; see *Contribute to matplotlib documentation*.

25.3.4 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 and add a SWF backend to support these kind of images. If you are interested in contributing to these efforts that would be great.

25.4 Search examples

The nearly 300 code *examples-index* 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`.

25.5 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 here):

```
@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&SrcAuth=A
```

CHAPTER

TWENTYSIX

TROUBLESHOOTING

Contents

26.1 Obtaining matplotlib version

To find out your matplotlib version number, import it and print the `__version__` attribute:

```
>>> import matplotlib
>>> matplotlib.__version__
'0.98.0'
```

26.2 `matplotlib` install location

You can find what directory matplotlib is installed in by importing it and printing the `__file__` attribute:

```
>>> import matplotlib
>>> matplotlib.__file__
'/home/jdhunter/dev/lib64/python2.5/site-packages/matplotlib/__init__.pyc'
```

26.3 .matplotlib directory location

Each user has a .matplotlib/ directory which may contain a *matplotlibrc* file and various caches to improve matplotlib's performance. To locate your .matplotlib/ directory, use matplotlib.get_configdir():

```
>>> import matplotlib as mpl
>>> mpl.get_configdir()
'/home/darren/.matplotlib'
```

On unix-like systems, this directory is generally located in your HOME directory. On windows, it is in your documents and settings directory by default:

```
>>> import matplotlib
>>> mpl.get_configdir()
'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*.

26.4 Report a problem

If you are having a problem with matplotlib, search the mailing lists first: there's a good chance someone else has already run into your problem.

If not, 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:

  ```
  python -c `import matplotlib; print matplotlib.__version__`
  ```

- where you obtained matplotlib (e.g. your Linux distribution's packages or the matplotlib Sourceforge site, or the enthought python distribution EPD).
- 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 matlotlib by running your script with a `verbose-helpful` or `--verbose-debug` flags and posting the verbose output the lists:

```
> 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:

  ```
  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 – eg, `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.

26.5 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:

```
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 *Report a problem*.

CHAPTER

TWENTYSEVEN

ENVIRONMENT VARIABLES

Contents

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.gettmpdir()` is used to find a base directory in which the `matplotlib` subdirectory is created.

27.1 Setting environment variables in Linux and OS-X

To list the current value of `PYTHONPATH`, which may be empty, try:

```
echo $PYTHONPATH
```

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:

```
echo $SHELL
```

27.1.1 BASH/KSH

To create a new environment variable:

```
export PYTHONPATH=~/Python
```

To prepend to an existing environment variable:

```
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:

```
export PATH=${PATH}:~/bin
```

To make your changes available in the future, add the commands to your `~/.bashrc` file.

27.1.2 CSH/TCSH

To create a new environment variable:

```
setenv PYTHONPATH ~/Python
```

To prepend to an existing environment variable:

```
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:

```
setenv PATH ${PATH}:~/bin
```

To make your changes available in the future, add the commands to your `~/.cshrc` file.

27.2 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*.

Part III

The Matplotlib Developers' Guide

CHAPTER

TWENTYEIGHT

CODING GUIDE

28.1 Committing changes

When committing changes to matplotlib, there are a few things to bear in mind.

- if your changes are non-trivial, please make an entry in the `CHANGELOG`
- if you change the API, please document it in `doc/api/api_changes.rst`, and consider posting to matplotlib-devel
- Are your changes python2.6 compatible? We support python2.6 and later
- Can you pass `examples/tests/backend_driver.py`? This is our poor man's unit test.
- Can you add a test to `lib/matplotlib/tests` to test your changes?
- If you have altered extension code, do you pass `unit/memleak_hawaii3.py`?
- if you have added new files or directories, or reorganized existing ones, are the new files included in the match patterns in `MANIFEST.in`. This file determines what goes into the source distribution of the mpl build.
- Keep the maintenance branches and master in sync where it makes sense.

28.2 Style guide

28.2.1 Importing and name spaces

For numpy, use:

```
import numpy as np
a = np.array([1,2,3])
```

For masked arrays, use:

```
import numpy.ma as ma
```

For matplotlib main module, use:

```
import matplotlib as mpl
mpl.rcParams['xtick.major.pad'] = 6
```

For matplotlib modules (or any other modules), use:

```
import matplotlib.cbook as cbook

if cbook.iterable(z):
    pass
```

We prefer this over the equivalent `from matplotlib import cbook` because the latter is ambiguous as to whether `cbook` is a module or a function. The former makes it explicit that you are importing a module or package. There are some modules with names that match commonly used local variable names, eg `matplotlib.lines` or `matplotlib.colors`. To avoid the clash, use the prefix 'm' with the `import some.thing as mthing` syntax, eg:

```
import matplotlib.lines as mlines
import matplotlib.transforms as transforms   # OK
import matplotlib.transforms as mtransforms  # OK, if you want to disambiguate
import matplotlib.transforms as mtrans       # OK, if you want to abbreviate
```

28.2.2 Naming, spacing, and formatting conventions

In general, we want to hew as closely as possible to the standard coding guidelines for python written by Guido in PEP 0008, though we do not do this throughout.

- functions and class methods: `lower` or `lower_underscore_separated`
- attributes and variables: `lower` or `lowerUpper`
- classes: `Upper` or `MixedCase`

Prefer the shortest names that are still readable.

Configure your editor to use spaces, not hard tabs. The standard indentation unit is always four spaces; if there is a file with tabs or a different number of spaces it is a bug – please fix it. To detect and fix these and other whitespace errors (see below), use reindent.py as a command-line script. Unless you are sure your editor always does the right thing, please use reindent.py before committing your changes in git.

Keep docstrings uniformly indented as in the example below, with nothing to the left of the triple quotes. The `matplotlib.cbook.dedent()` function is needed to remove excess indentation only if something will be interpolated into the docstring, again as in the example below.

Limit line length to 80 characters. If a logical line needs to be longer, use parentheses to break it; do not use an escaped newline. It may be preferable to use a temporary variable to replace a single long line with two shorter and more readable lines.

Please do not commit lines with trailing white space, as it causes noise in git diffs. Tell your editor to strip whitespace from line ends when saving a file. If you are an emacs user, the following in your `.emacs` will cause emacs to strip trailing white space upon saving for python, C and C++:

```
; and similarly for c++-mode-hook and c-mode-hook
(add-hook 'python-mode-hook
          (lambda ()
          (add-hook 'write-file-functions 'delete-trailing-whitespace)))
```

for older versions of emacs (emacs<22) you need to do:

```
(add-hook 'python-mode-hook
          (lambda ()
          (add-hook 'local-write-file-hooks 'delete-trailing-whitespace)))
```

28.2.3 Keyword argument processing

Matplotlib makes extensive use of `**kwargs` for pass-through customizations from one function to another. A typical example is in `matplotlib.pylab.text()`. The definition of the pylab text function is a simple pass-through to `matplotlib.axes.Axes.text()`:

```
# 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.Text.__init__()`:

```
# in 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.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...
```

28.3 Documentation and docstrings

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. Eg. in `matplotlib.lines.Line2D`:

```
# in lines.py
def set_linestyle(self, linestyle):
    """
    Set the linestyle of the line

    ACCEPTS: [ '-' | '--' | '-.' | ':' | 'steps' | 'None' | ' ' | '' ]
    """
```

Since matplotlib uses a lot of pass-through `kwargs`, eg. 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 functions `matplotlib.artist.kwdocd` and `matplotlib.artist.kwdoc()` to facilitate this. They combine python string interpolation in the docstring with the matplotlib artist introspection facility that underlies `setp` and `getp`. The `kwdocd` is a single dictionary that maps class name to a docstring of `kwargs`. Here is an example from `matplotlib.lines`:

```
# in lines.py
artist.kwdocd['Line2D'] = artist.kwdoc(Line2D)
```

Then in any function accepting `Line2D` pass-through `kwargs`, eg. `matplotlib.axes.Axes.plot()`:

```
# in axes.py
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
plot.__doc__ = cbook.dedent(plot.__doc__) % artist.kwdocd
```

Note there is a problem for `Artist` `__init__` methods, eg. `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 `artist.kwdocd['Patch']` setting in `matplotlib.patches`.

28.4 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 use it on one of several ways

- in matplotlibrc:

  ```
  backend : module://my_backend
  ```

- with the use directive is your script:

  ```
  import matplotlib
  matplotlib.use('module://my_backend')
  ```

- from the command shell with the -d flag:

  ```
  > python simple_plot.py -d module://my_backend
  ```

28.5 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. Many people find these examples from the website, and do not have ready access to the file:`examples` directory in which they reside. Thus any example data that is required for the example should be added to the sample_data git repository. Then in your example code you can load it into a file handle with:

```
import matplotlib.cbook as cbook
fh = cbook.get_sample_data('mydata.dat')
```

The file will be fetched from the git repo using urllib and updated when the revision number changes.

If you prefer just to get the full path to the file instead of a file object:

```
import matplotlib.cbook as cbook
datafile = cbook.get_sample_data('mydata.dat', asfileobj=False)
print 'datafile', datafile
```

28.6 Writing a new pyplot function

A large portion of the pyplot interface is automatically generated by the `boilerplate.py` script (in the root of the source tree). To add or remove a plotting method from pyplot, edit the appropriate list in `boilerplate.py` and then run the script which will update the content in `lib/matplotlib/pyplot.py`. Both the changes in `boilerplate.py` and `lib/matplotlib/pyplot.py` should be checked into the repository.

28.7 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.)

28.7.1 Requirements

The following software is required to run the tests:

- nose, version 1.0 or later
- Ghostscript (to render PDF files)
- Inkscape (to render SVG files)

28.7.2 Running the tests

Running the tests is simple. Make sure you have nose installed and run the script `tests.py` in the root directory of the distribution. The script can take any of the usual nosetest arguments, such as

`-v`	increase verbosity
`-d`	detailed error messages
`--with-coverage`	enable collecting coverage information

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, eg. (this is assuming the test is installed):

```
python tests.py matplotlib.tests.test_simplification:test_clipping
```

An alternative implementation that does not look at command line arguments works from within Python:

```
import matplotlib
matplotlib.test()
```

Running tests by any means other than `matplotlib.test()` does not load the nose "knownfailureif" (Known failing tests) plugin, causing known-failing tests to fail for real.

28.7.3 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`:

```
from nose.tools import assert_equal

def test_simple():
    '''very simple example test'''
    assert_equal(1+1,2)
```

Nose determines which functions are tests by searching for functions beginning with "test" in their name.

28.7.4 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:

```
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'])
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.*) to the baseline_images tree in the source directory (in this case lib/matplotlib/tests/baseline_images/test_category) and put them under source code revision control (with git add). When rerunning the tests, they should now pass.

There are two optional keyword arguments to the image_comparison decorator:

- extensions: If you only wish to test some of the image formats (rather than the default png, svg and pdf formats), pass a list of the extensions to test.
- 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.

28.7.5 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:

```
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.

28.7.6 Creating a new module in matplotlib.tests

Let's say you've added a new module named matplotlib.tests.test_whizbang_features. To add this module to the list of default tests, append its name to default_test_modules in lib/matplotlib/__init__.py.

28.7.7 Using tox

Tox is a tool for running tests against multiple Python environments, including multiple versions of Python (e.g.: 2.6, 2.7, 3.2, etc.) and even different Python implementations altogether (e.g.: CPython, PyPy, Jython, etc.)

Testing all 4 versions of Python (2.6, 2.7, 3.1, and 3.2) requires having four 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, or use pythonbrew.

tox makes it easy to determine if your working copy introduced any regressions before submitting a pull request. Here's how to use it:

```
$ pip install tox
$ tox
```

You can also run tox on a subset of environments:

```
$ 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:

```
$ 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.

28.7.8 Using Travis CI

Travis CI is a hosted CI system "in the cloud".

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.

Once this is configured, you can see the Travis CI results at http://travis-ci.org/#!/your_GitHub_user_name/matplotlib – here's an example.

28.8 Licenses

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 (eg basemap), but make sure you clearly state the licenses you are using.

28.8.1 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.

CHAPTER

TWENTYNINE

WORKING WITH *MATPLOTLIB* SOURCE CODE

Contents:

29.1 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*.

29.2 Install git

29.2.1 Overview

Debian / Ubuntu	`sudo apt-get install git-core`
Fedora	`sudo yum install git-core`
Windows	Download and install msysGit
OS X	Use the git-osx-installer

29.2.2 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: http://book.git-scm.com/2_installing_git.html

29.3 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 git repository from github
- update local copy from time to time

29.3.1 Get the local copy of the code

From the command line:

```
git clone git://github.com/matplotlib/matplotlib.git
```

You now have a copy of the code tree in the new `matplotlib` directory.

29.3.2 Updating the code

From time to time you may want to pull down the latest code. Do this with:

```
cd matplotlib
git pull
```

The tree in `matplotlib` will now have the latest changes from the initial repository.

29.4 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.

29.4.1 Making patches

Overview

```
# tell git who you are
git config --global user.email you@yourdomain.example.com
git config --global user.name "Your Name Comes Here"
# get the repository if you don't have it
git clone git://github.com/matplotlib/matplotlib.git
# make a branch for your patching
cd matplotlib
git branch the-fix-im-thinking-of
git checkout the-fix-im-thinking-of
# 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
```

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'
```

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:

```
git status
```

6. Finally, make your commits into patches. You want all the commits since you branched from the `master` branch:

```
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:

```
git checkout master
```

29.4.2 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:

```
# 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*.

29.5 Git for development

Contents:

29.5.1 Making your own copy (fork) of matplotlib

You need to do this only once. The instructions here are very similar to the instructions at http://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.

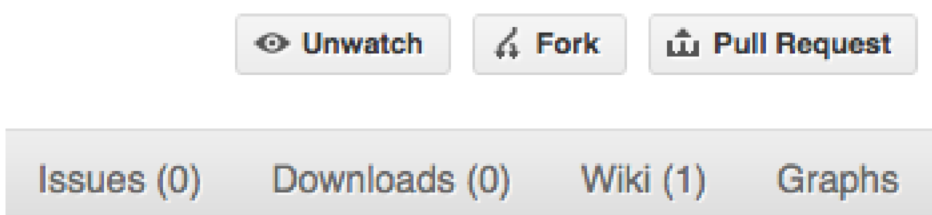
Set up and configure a github account

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 and some 'Hardcore forking action', you should find yourself at the home page for your own forked copy of matplotlib.

29.5.2 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

```
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.

Note this command needs to be run on every clone of the repository that you make. It is not tracked in your personal repository on github.

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)
```

29.5.3 Configure git

Overview

Your personal git configurations are saved in the `.gitconfig` file in your home directory. Here is an example `.gitconfig` file:

```
[user]
        name = Your Name
        email = you@yourdomain.example.com

[alias]
        ci = commit -a
        co = checkout
        st = status -a
        stat = status -a
        br = branch
        wdiff = diff --color-words

[core]
        editor = vim

[merge]
        summary = true

[apply]
        whitespace = fix

[core]
        autocrlf = input
```

You can edit this file directly or you can use the `git config --global` command:

```
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:

```
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:

```
[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:

```
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"
```

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
```

29.5.4 Development workflow

You already have your own forked copy of the matplotlib repository, by following *Making your own copy (fork) of matplotlib*, *Set up your fork*, and you have configured git by following *Configure git*.

Workflow summary

- Keep your `master` branch clean of edits that have not been merged to the main matplotlib development repo. Your `master` then will follow the main matplotlib repository.
- Start a new *feature branch* for each set of edits that you do.
- If you can avoid it, try not to merge other branches into your feature branch while you are working.
- Ask for review!

This way of working really helps to keep work well organized, and in keeping history as clear as possible.

See — for example — linux git workflow.

Making a new feature branch

```
git checkout -b my-new-feature master
```

This will create and immediately check out a feature branch based on `master`. To create a feature branch based on a maintenance branch, use:

```
git fetch origin
git checkout -b my-new-feature origin/v1.0.x
```

Generally, you will want to keep this also 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
```

You will need to use this exact command, rather than simply `git push` every time you want to push changes on your feature branch to your github repo. However, in git >1.7 you can set up a link by using the `--set-upstream` option:

```
git push --set-upstream origin my-new-feature
```

and then next time you need to push changes to your branch a simple `git push` will suffice. Note that `git push` pushes out all branches that are linked to a remote branch.

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:

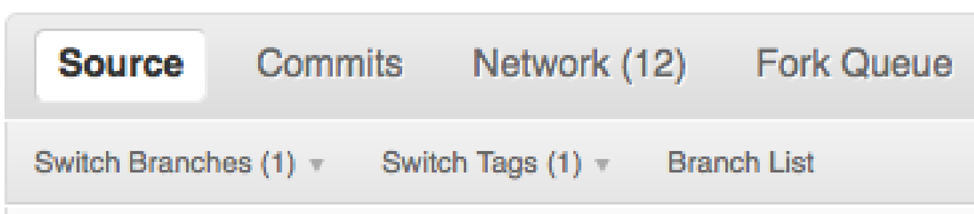
   ```
   # On branch ny-new-feature
   # Changed but not updated:
   #   (use "git add <file>..." to update what will be committed)
   ```

```
#   (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")
```

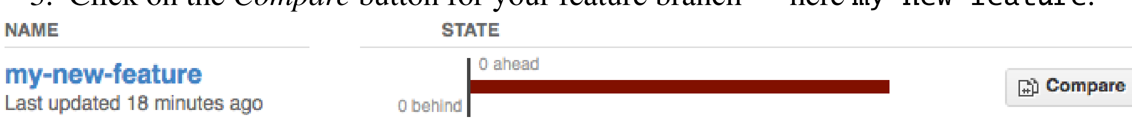
3. Check what the actual changes are with `git diff` (git diff).
4. Add any new files to version control `git add new_file_name` (see git add).
5. To commit all modified files into the local copy of your repo,, do `git commit -am 'A commit message'`. Note the `-am` options to `commit`. The `m` flag just 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`).

Asking for code review

1. Go to your repo URL — e.g. `http://github.com/your-user-name/matplotlib`.
2. Click on the *Branch list* button:



3. Click on the *Compare* button for your feature branch — here `my-new-feature`:



4. If asked, select the *base* and *comparison* branch names you want to compare. Usually these will be `master` and `my-new-feature` (where that is your feature branch name).
5. At this point you should get a nice summary of the changes. Copy the URL for this, and post it to the matplotlib mailing list, asking for review. The URL will look something like: `http://github.com/your-user-name/matplotlib/compare/master...my-new-feature`. There's an example at http://github.com/matthew-brett/nipy/compare/master...find-install-data See: http://github.com/blog/612-introducing-github-compare-view for more detail.

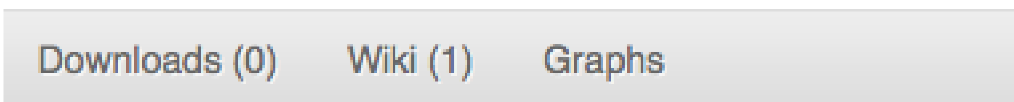
The generated comparison, is between your feature branch `my-new-feature`, and the place in `master` from which you branched `my-new-feature`. In other words, you can keep updating `master` without interfering with the output from the comparison. More detail? Note the three dots in the URL above (`master...my-new-feature`) and see *dot2-dot3*.

Asking for your changes to be merged into the main repo

When you are ready to ask for the merge of your code:

1. Go to the URL of your forked repo, say `http://github.com/your-user-name/matplotlib.git`.
2. Click on the 'Pull request' button:

   Enter a message; we suggest you select only `matplotlib` as the recipient. The message will go to the matplotlib mailing list. Please feel free to add others from the list as you like.
3. If the branch is to be merged into a maintenance branch on the main repo, make sure the "base branch" indicates the maintenance branch and not master. Github can not automatically determine the branch to merge into.

Staying up to date with changes in the central repository

This updates your working copy from the upstream matplotlib github repo.

Overview

```
# go to your master branch
git checkout master
# pull changes from github
git fetch upstream
# merge from upstream
git merge --ff-only upstream/master
```

In detail

We suggest that you do this only for your `master` branch, and leave your 'feature' branches unmerged, to keep their history as clean as possible. This makes code review easier:

```
git checkout master
```

Make sure you have done *Linking your repository to the upstream repo*.

Merge the upstream code into your current development by first pulling the upstream repo to a copy on your local machine:

```
git fetch upstream
```

then merging into your current branch:

```
git merge --ff-only upstream/master
```

The `--ff-only` option guarantees that if you have mistakenly committed code on your `master` branch, the merge fails at this point. If you were to merge `upstream/master` to your `master`, you would start to diverge from the upstream. If this command fails, see the section on accidents.

The letters 'ff' in `--ff-only` mean 'fast forward', which is a special case of merge where git can simply update your branch to point to the other branch and not do any actual merging of files. For `master` and other integration branches this is exactly what you want.

Other integration branches

Some people like to keep separate local branches corresponding to the maintenance branches on github. At the time of this writing, `v1.0.x` is the active maintenance branch. If you have such a local branch, treat is just as `master`: don't commit on it, and before starting new branches off of it, update it from upstream:

```
git checkout v1.0.x
git fetch upstream
git merge --ff-only upstream/v1.0.x
```

But you don't necessarily have to have such a branch. Instead, if you are preparing a bugfix that applies to the maintenance branch, fetch from upstream and base your bugfix on the remote branch:

```
git fetch upstream
git checkout -b my-bug-fix upstream/v1.0.x
```

Recovering from accidental commits on master

If you have accidentally committed changes on `master` and `git merge --ff-only` fails, don't panic! First find out how much you have diverged:

```
git diff upstream/master...master
```

If you find that you want simply to get rid of the changes, reset your `master` branch to the upstream version:

```
git reset --hard upstream/master
```

As you might surmise from the words 'reset' and 'hard', this command actually causes your changes to the current branch to be lost, so think twice.

If, on the other hand, you find that you want to preserve the changes, create a feature branch for them:

```
git checkout -b my-important-changes
```

Now `my-important-changes` points to the branch that has your changes, and you can safely reset `master` as above — but make sure to reset the correct branch:

```
git checkout master
git reset --hard upstream/master
```

Deleting 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 `test-branch`. See also: http://github.com/guides/remove-a-remote-branch

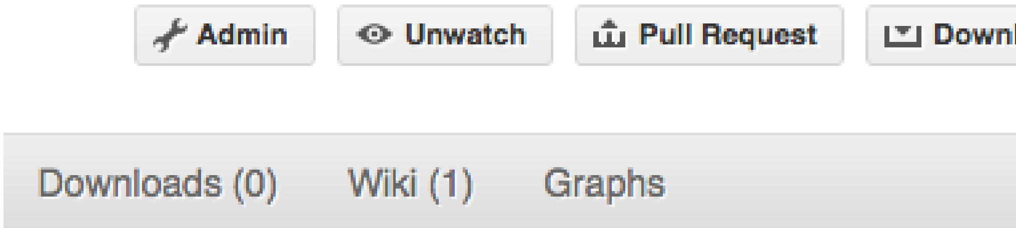
Several people sharing a single repository

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 `http://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@githhub.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
```

Exploring 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.

29.6 git resources

29.6.1 Tutorials and summaries

- github help has an excellent series of how-to guides.
- learn.github has an excellent series of tutorials
- 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 casts — video snippets giving git how-tos.
- git magic — extended introduction with intermediate detail
- The git parable is an easy read explaining the concepts behind git.
- Our own 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

29.6.2 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.

29.6.3 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

CHAPTER

THIRTY

DOCUMENTING MATPLOTLIB

30.1 Getting started

The documentation for matplotlib is generated from ReStructured Text using the Sphinx documentation generation tool. Sphinx-1.0 or later is required.

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/`.

30.2 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:

```
.. include:: ../../TODO
```

30.3 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 consistant 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:

  ```
  .. math::

    \int_{-\infty}^{\infty}\frac{e^{i\phi}}{1+x^2\frac{e^{i\phi}}{1+x^2}}
  ```

  yields:

  $$\int_{-\infty}^{\infty}\frac{e^{i\phi}}{1+x^2\frac{e^{i\phi}}{1+x^2}} \tag{30.1}$$

- Interactive IPython sessions can be illustrated in the documentation using the following directive:

  ```
  .. sourcecode:: 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.

- Use the *versionadded* and *versionchanged* directives, which have similar syntax to the *deprecated* role:

  ```
  .. 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:

  ```
  .. 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

[1] For 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.

30.3.1 Docstrings

In addition to the aforementioned formatting suggestions:

- 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.

30.4 Figures

30.4.1 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/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.:

```
.. plot:: mpl_examples/pylab_examples/simple_plot.py
```

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:

   ```
   .. 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:

   ```
   .. plot:: path/to/plot.py

      This is the caption for the plot
   ```

   Additionally, one my specify the name of a function to call (with no arguments) immediately after importing the module:

   ```
   .. plot:: path/to/plot.py plot_function1
   ```

2. Included as **inline content** to the directive:

   ```
   .. 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:

   ```
   .. 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] 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.

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_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 directoly where the file containing the directive is.)

plot_formats File formats to generate. List of tuples or strings:

```
[(suffix, dpi), suffix, ...]
```

that determine the file format and the DPI. For entries whose DPI was omitted, sensible defaults are chosen.

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 behaviour 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 resturctured text.

30.4.2 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:

```
.. plot:: pyplots/tex_unicode_demo.py
   :include-source:
```

30.4.3 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 -*-
```

30.4.4 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:

```
.. 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

```
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 John Hunter for the login password to upload youtube videos of google docs to the mplgithub account.

30.5 Referring to mpl documents

In the documentation, you may want to include to 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, eg, 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_*
mpl_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 (eg the font files are duplicated and large)

30.6 Internal section references

To maximize internal consistency in section labeling and references, use hypen separated, descriptive labels for section references, eg:

```
.. _howto-webapp:
```

and refer to it using the standard reference syntax:

```
See :ref:`howto-webapp`
```

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 necesssary, 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.

30.7 Section names, etc

For everything but top level chapters, please use `Upper lower` for section titles, eg `Possible hangups` rather than `Possible Hangups`

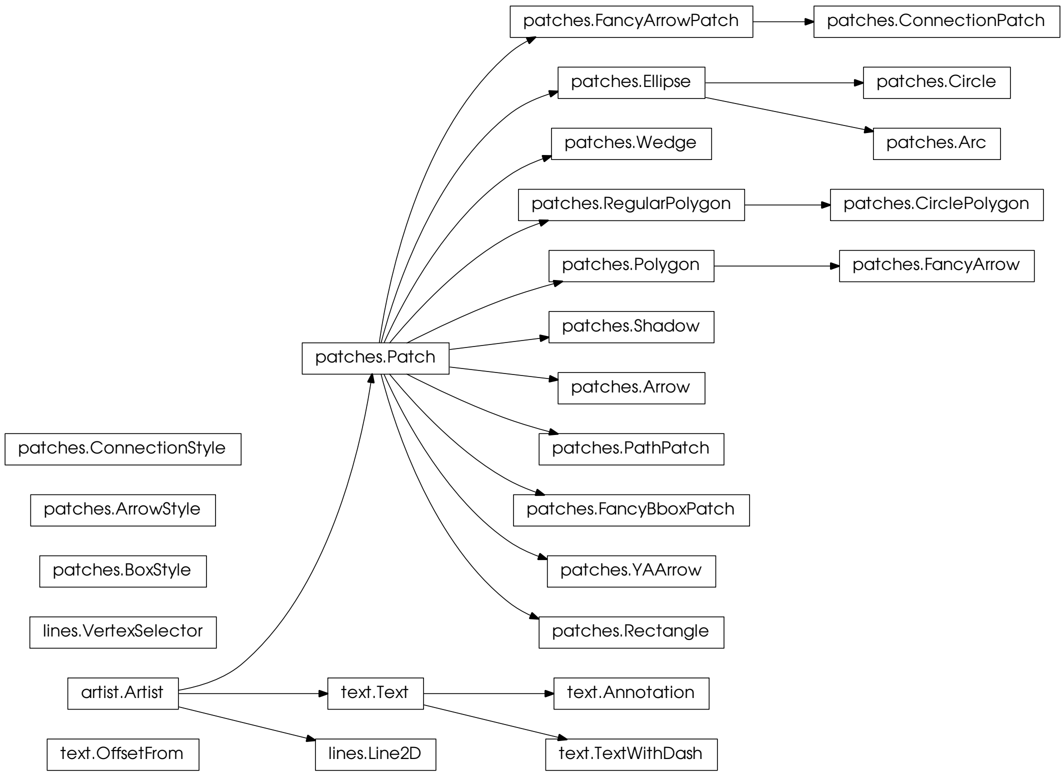
30.8 Inheritance diagrams

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
```



30.9 Emacs helpers

There is an emacs mode rst.el which automates many important ReST tasks like building and updateing 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
```

CHAPTER

THIRTYONE

DOING A MATPLOLIB RELEASE

A guide for developers who are doing a matplotlib release

- Edit `__init__.py` and bump the version number

When doing a release

31.1 Testing

- Run all of the regression tests by running the `tests.py` script at the root of the source tree.
- Run `unit/memleak_hawaii3.py` and make sure there are no memory leaks
- try some GUI examples, eg `simple_plot.py` with GTKAgg, TkAgg, etc...
- remove font cache and tex cache from `.matplotlib` and test with and without cache on some example script
- Optionally, make sure `examples/tests/backend_driver.py` runs without errors and check the output of the PNG, PDF, PS and SVG backends

31.2 Branching

Once all the tests are passing and you are ready to do a release, you need to create a release branch:

```
git checkout -b v1.1.x
git push git@github.com:matplotlib/matplotlib.git v1.1.x
```

On the branch, do any additional testing you want to do, and then build binaries and source distributions for testing as release candidates.

For each release candidate as well as for the final release version, please `git tag` the commit you will use for packaging like so:

```
git tag -a v1.1.0rc1
```

The `-a` flag will allow you to write a message about the tag, and affiliate your name with it. A reasonable tag message would be something like `v1.1.0 Release Candidate 1 (September 24, 2011)`. To tag a release after the fact, just track down the commit hash, and:

```
git tag -a v1.0.1 a9f3f3a50745
```

Tags allow developers to quickly checkout different releases by name, and also provides source download via zip and tarball on github.

31.3 Packaging

- Make sure the `MANIFEST.in` us up to date and remove `MANIFEST` so it will be rebuilt by MANIFEST.in
- run `git clean` in the mpl git directory before building the sdist
- unpack the sdist and make sure you can build from that directory
- Use `setup.cfg` to set the default backends. For windows and OSX, the default backend should be TkAgg. You should also turn on or off any platform specific build options you need. Importantly, you also need to make sure that you delete the `build` dir after any changes to `setup.cfg` before rebuilding since cruft in the `build` dir can get carried along.
- on windows, unix2dos the rc file
- We have a Makefile for the OS X builds in the mpl source dir `release/osx`, so use this to prepare the OS X releases.
- We have a Makefile for the win32 mingw builds in the mpl source dir `release/win32` which you can use this to prepare the windows releases.

31.4 Release candidate testing

Post the release candidates tarballs to the matplotlib download page. If you have developer rights, you should see an "Upload a new file" section there.

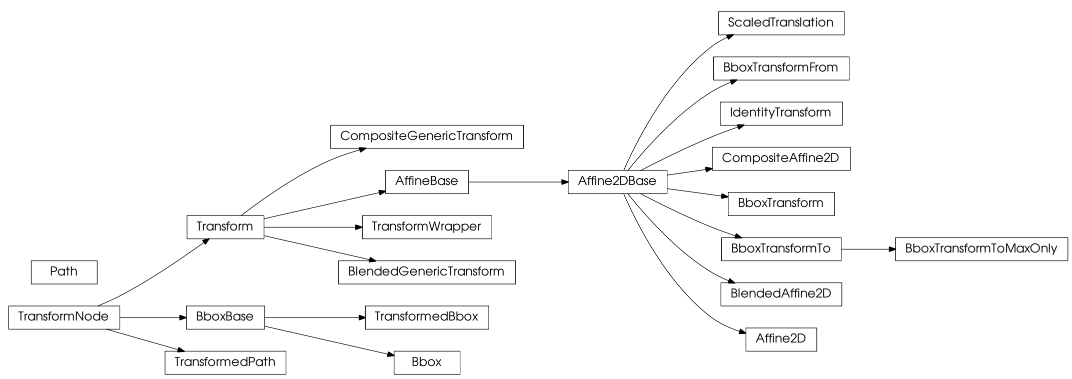
31.5 Announcing

Announce the release on matplotlib-announce, matplotlib-users and matplotlib-devel. Include a summary of highlights from the CHANGELOG and/or post the whole CHANGELOG since the last release.

CHAPTER

THIRTYTWO

WORKING WITH TRANSFORMATIONS

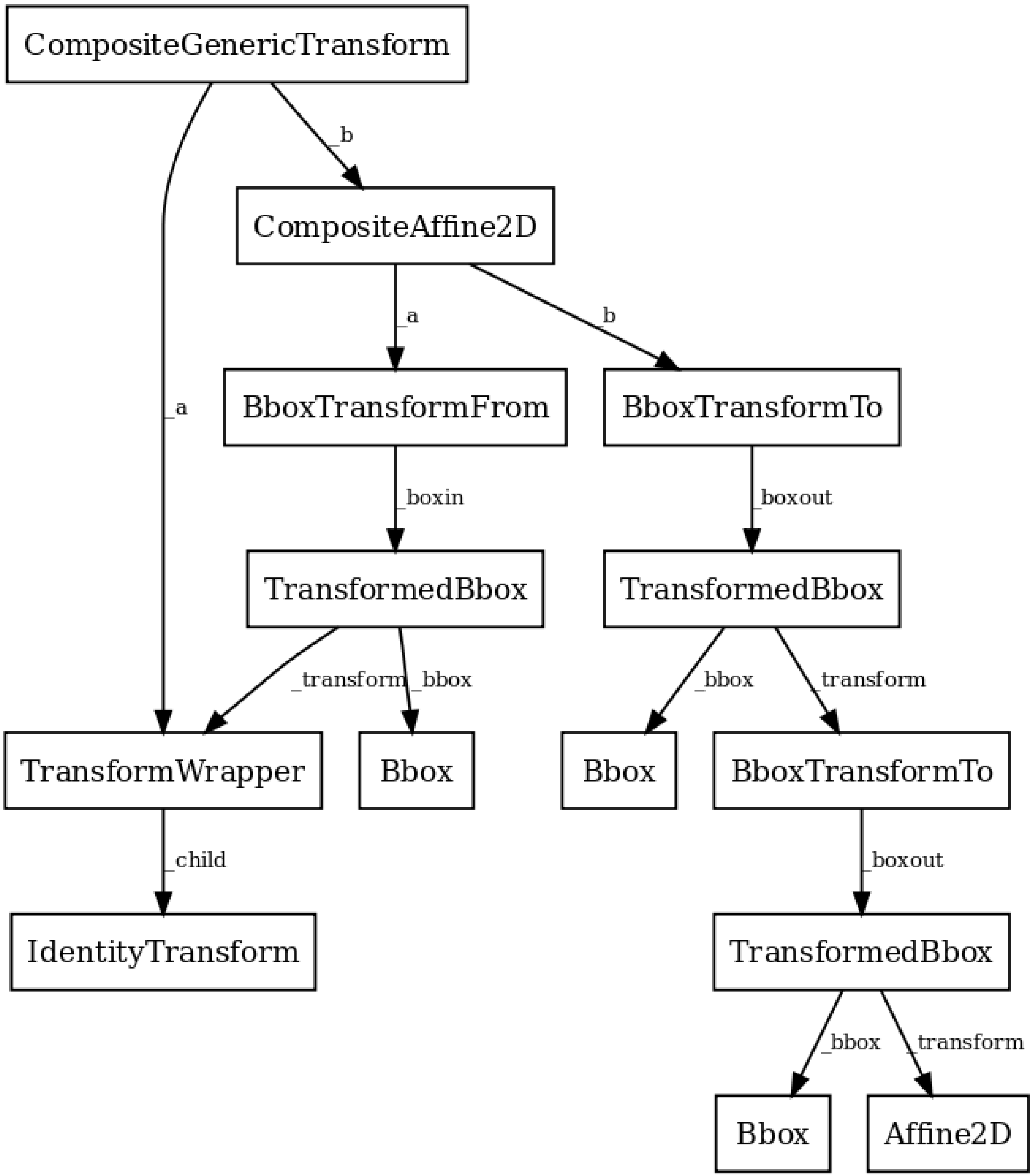


32.1 matplotlib.transforms

matplotlib includes a framework for arbitrary geometric transformations that is used 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:

```
full transform == non-affine part + 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 significance 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 significance 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 (`x0`, `y0`, `width`, `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 `x0` and `x1`.

containsy(*y*)
Returns True if *y* is between or equal to `y0` and `y1`.

corners()
Return an array of points which are the four corners of this rectangle. For example, if this `Bbox` is defined by the points (*a*, *b*) and (*c*, *d*), `corners()` returns (*a*, *b*), (*a*, *d*), (*c*, *b*) and (*c*, *d*).

count_contains(*vertices*)
Count the number of vertices contained in the `Bbox`.

vertices is a Nx2 Numpy array.

count_overlaps(*bboxes*)
Count the number of bounding boxes that overlap this one.

bboxes is a sequence of `BboxBase` objects

expanded(*sw*, *sh*)
Return a new `Bbox` which is this `Bbox` expanded around its center by the given factors *sw* and *sh*.

extents
(property) Returns (`x0`, `y0`, `x1`, `y1`).

frozen()
`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 `x0` and `x1`.

fully_containsy(*y*)
Returns True if *y* is between but not equal to `y0` and `y1`.

fully_overlaps(*other*)
Returns True if this bounding box overlaps with the given bounding box *other*, but not on its edge alone.

height
(property) The height of the bounding box. It may be negative if `y1` < `y0`.

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) `intervaly` 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.splitx(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()` or `update_from_data_xy()`.

value:

- When True, subsequent calls to `update_from_data()` will ignore the existing bounds of the `Bbox`.
- When False, subsequent calls to `update_from_data()` 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

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*)
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

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 seperate 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`).

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(value`

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`).

transform_angles(*angles*, *pts*, *radians=False*, *pushoff=1e-05*)
Performs transformation on a set of angles anchored at specific locations.

The *angles* must be a column vector (i.e., numpy array).

The *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 *angles*.

***radians* indicates whether or not input angles are given in** radians (True) or degrees (False; the default).

***pushoff* is the distance to move away from *pts* for** determining transformed angles (see discussion of method below).

The transformed angles are returned in an array with the same size as *angles*.

The generic version of this method uses a very generic algorithm that transforms *pts*, as well as locations very close to *pts*, to find the angle in the transformed system.

transform_non_affine(*values*)
Performs only the non-affine part of the transformation.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(value`

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`).

`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_affi`

`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_affi`

`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`).

`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(value`

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`).

`transform_non_affine``(points)`
Performs only the non-affine part of the transformation.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(value`

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`).

`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_affi`

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_affi`

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 === 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:

```
a c e
b d f
0 0 1
```

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(value`

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`).

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:

```
a c e
b d f
0 0 1
```

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:

```
a c e
b d f
0 0 1
```

.

get_matrix()

Get the underlying transformation matrix as a 3x3 numpy array:

```
a c e
b d f
0 0 1
```

.

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:

```
a c e
b d f
0 0 1
```

.

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 on 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(value

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).

transform_affine(*points*)

Performs only the non-affine part of the transformation.

transform(values) is always equivalent to transform_affine(transform_non_affine(value

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).

transform_non_affine(*points*)

Performs only the non-affine part of the transformation.

transform(values) is always equivalent to transform_affine(transform_non_affine(value

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).

transform_path(*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_affi`

transform_path_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_affi`

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_affi`

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 === self.inverted().transform(self.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(value`

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`).

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.

`get_matrix`()

Get the Affine transformation array for the affine part of this transform.

`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.

class `matplotlib.transforms.`**`CompositeGenericTransform`**(*a*, *b*, ***kwargs*)

Bases: `matplotlib.transforms.Transform`

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.

`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 === self.inverted().transform(self.transform(x))`

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(value`

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`).

transform_non_affine(*points*)
Performs only the non-affine part of the transformation.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(value`

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`).

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_affi`

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 affines.

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 Affine transformation array for the affine part of this transform.

matplotlib.transforms.**composite_transform_factory**(*a*, *b*)

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
```

class matplotlib.transforms.**BboxTransform**(*boxin*, *boxout*, ***kwargs*)

Bases: matplotlib.transforms.Affine2DBase

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*.

get_matrix()

Get the Affine transformation array for the affine part of this transform.

class matplotlib.transforms.**BboxTransformFrom**(*boxin*, ***kwargs*)

Bases: matplotlib.transforms.Affine2DBase

BboxTransformFrom linearly transforms points from a given Bbox to the unit bounding box.

get_matrix()

Get the Affine transformation array for the affine part of this transform.

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*.

get_matrix()

Get the Affine transformation array for the affine part of this transform.

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.

get_fully_transformed_path()

Return a fully-transformed copy of the child path.

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, returns -*expander*, *expander*.

CHAPTER

THIRTYTHREE

ADDING NEW SCALES AND PROJECTIONS TO MATPLOTLIB

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 `set_xscale()` and `set_xscale()`. Projections can be chosen using the `projection` keyword argument to the `plot()` or `subplot()` functions, e.g.:

```
plot(x, y, projection="custom")
```

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.

33.1 Creating a new scale

Adding a new scale consists of defining a subclass of `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 (`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

`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.

33.2 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:

  ```
  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:

```
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.

33.3 API documentation

33.3.1 matplotlib.scale

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`.

`set_default_locators_and_formatters`(*axis*)

Set the locators and formatters to reasonable defaults for linear scaling.

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.

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.

set_default_locators_and_formatters(*axis*)
Set the locators and formatters to specialized versions for log scaling.

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()

And optionally:

- set_default_locators_and_formatters()
- limit_range_for_scale()

get_transform()
Return the Transform object associated with this scale.

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.

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.

get_transform()
Return a `SymmetricalLogTransform` instance.

set_default_locators_and_formatters(*axis*)
Set the locators and formatters to specialized versions for symmetrical log scaling.

matplotlib.scale.**get_scale_docs**()
Helper function for generating docstrings related to scales.

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 | symlog]

33.3.2 matplotlib.projections

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.

`matplotlib.projections.projection_factory(`*projection*, *figure*, *rect*, ***kwargs*`)`
Get a new projection instance.

projection is a projection name.

figure is a figure to add the axes to.

rect is a `Bbox` object specifying the location of the axes within the figure.

Any other kwargs are passed along to the specific projection constructor being used. Deprecated since version 1.3.

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))`

`transform_non_affine(`*xy*`)`
Performs only the non-affine part of the transformation.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(value`

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`).

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.

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 `matplotlib.projections.polar.`**`PolarAxes`**(**args*, ***kwargs*)
Bases: `matplotlib.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 === self.inverted().transform(self.transform(x))`

`transform_non_affine`(*xy*)
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`).

class `PolarAxes.`**`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 PolarAxes.**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 === self.inverted().transform(self.transform(x))`

transform_non_affine(*tr*)
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`).

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_a`

class PolarAxes.**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 PolarAxes.**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.

`PolarAxes.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.

`PolarAxes.can_zoom()`
Return *True* if this axes supports the zoom box button functionality.

Polar axes do not support zoom boxes.

`PolarAxes.format_coord(theta, r)`
Return a format string formatting the coordinate using Unicode characters.

`PolarAxes.get_data_ratio()`
Return the aspect ratio of the data itself. For a polar plot, this should always be 1.0

`PolarAxes.get_theta_direction()`
Get the direction in which theta increases.

-1: Theta increases in the clockwise direction

1: Theta increases in the counterclockwise direction

`PolarAxes.get_theta_offset()`
Get the offset for the location of 0 in radians.

`PolarAxes.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 `len(radii)` list of strings of the labels to use at each radius.

If *labels* is None, the built-in formatter will be used.

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:

Property	Description
`agg_filter`	unknown
`alpha`	float (0.0 transparent through 1.0 opaque)
`animated`	[True | False]
`axes`	an `Axes` instance
`backgroundcolor`	any matplotlib color
`bbox`	rectangle prop dict
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]

Table 33.1 – cont

Property	Description
clip_path	[(Path, Transform) | Patch | None]
color	any matplotlib color
contains	a callable function
family or fontfamily or fontname or name	[FONTNAME | 'serif' | 'sans-serif' | 'cursive' | 'fantasy' | 'mono
figure	a matplotlib.figure.Figure instance
fontproperties or font_properties	a matplotlib.font_manager.FontProperties instance
gid	an id string
horizontalalignment or ha	['center' | 'right' | 'left']
label	string or anything printable with '%s' conversion.
linespacing	float (multiple of font size)
lod	[True | False]
multialignment	['left' | 'right' | 'center']
path_effects	unknown
picker	[None|float|boolean|callable]
position	(x,y)
rasterized	[True | False | None]
rotation	[angle in degrees | 'vertical' | 'horizontal']
rotation_mode	unknown
size or fontsize	[size in points | 'xx-small' | 'x-small' | 'small' | 'medium' | 'large
snap	unknown
stretch or fontstretch	[a numeric value in range 0-1000 | 'ultra-condensed' | 'extra-cond
style or fontstyle	['normal' | 'italic' | 'oblique']
text	string or anything printable with '%s' conversion.
transform	Transform instance
url	a url string
variant or fontvariant	['normal' | 'small-caps']
verticalalignment or va or ma	['center' | 'top' | 'bottom' | 'baseline']
visible	[True | False]
weight or fontweight	[a numeric value in range 0-1000 | 'ultralight' | 'light' | 'normal' |
x	float
y	float
zorder	any number

ACCEPTS: sequence of floats

PolarAxes.**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

PolarAxes.**set_theta_offset**(*offset*)
Set the offset for the location of 0 in radians.

PolarAxes.**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".

PolarAxes.**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). Eg. 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:

Property	Description
`agg_filter`	unknown
`alpha`	float (0.0 transparent through 1.0 opaque)
`animated`	[True | False]
`axes`	an `Axes` instance
`backgroundcolor`	any matplotlib color
`bbox`	rectangle prop dict
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color`	any matplotlib color
`contains`	a callable function
`family` or fontfamily or fontname or name	[FONTNAME | 'serif' | 'sans-serif' | 'cursive' | 'fantasy' | 'mono
`figure`	a `matplotlib.figure.Figure` instance
`fontproperties` or font_properties	a `matplotlib.font_manager.FontProperties` instance
`gid`	an id string
`horizontalalignment` or ha	['center' | 'right' | 'left']
`label`	string or anything printable with '%s' conversion.
`linespacing`	float (multiple of font size)
`lod`	[True | False]

Table 33.2 – cont

Property	Description
multialignment	['left' | 'right' | 'center']
path_effects	unknown
picker	[None|float|boolean|callable]
position	(x,y)
rasterized	[True | False | None]
rotation	[angle in degrees | 'vertical' | 'horizontal']
rotation_mode	unknown
size or fontsize	[size in points | 'xx-small' | 'x-small' | 'small' | 'medium' | 'large
snap	unknown
stretch or fontstretch	[a numeric value in range 0-1000 | 'ultra-condensed' | 'extra-conc
style or fontstyle	['normal' | 'italic' | 'oblique']
text	string or anything printable with '%s' conversion.
transform	Transform instance
url	a url string
variant or fontvariant	['normal' | 'small-caps']
verticalalignment or va or ma	['center' | 'top' | 'bottom' | 'baseline']
visible	[True | False]
weight or fontweight	[a numeric value in range 0-1000 | 'ultralight' | 'light' | 'normal'
x	float
y	float
zorder	any number

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 *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 === self.inverted().transform(self.transform(x))`

`transform_non_affine``(`*tr*`)`

Performs only the non-affine part of the transformation.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(value`

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`).

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_affi`

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.`**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.

CHAPTER

THIRTYFOUR

DOCS OUTLINE

Proposed chapters for the docs, who has responsibility for them, and who reviews them. The "unit" doesn't have to be a full chapter (though in some cases it will be), it may be a chapter or a section in a chapter.

User's guide unit	Author	Status	Reviewer
plotting 2-D arrays	Eric	has author	Perry ? Darren
colormapping	Eric	has author	?
quiver plots	Eric	has author	?
histograms	Manuel ?	no author	Erik Tollerud ?
bar / errorbar	?	no author	?
x-y plots	?	no author	Darren
time series plots	?	no author	?
date plots	John	has author	?
working with data	John	has author	Darren
custom ticking	?	no author	?
masked data	Eric	has author	?
patches	?	no author	?
legends	?	no author	?
animation	John	has author	?
collections	?	no author	?
text - mathtext	Michael	accepted	John
text - usetex	Darren	accepted	John
text - annotations	John	submitted	?
fonts et al	Michael ?	no author	Darren
pyplot tut	John	submitted	Eric
configuration	Darren	submitted	?
win32 install	Charlie ?	no author	Darren
os x install	Charlie ?	no author	?
linux install	Darren	has author	?
artist api	John	submitted	?

Continued on next page

Table 34.1 – continued from previous page

User's guide unit	Author	Status	Reviewer
event handling	John	submitted	?
navigation	John	submitted	?
interactive usage	?	no author	?
widgets	?	no author	?
ui - gtk	?	no author	?
ui - wx	?	no author	?
ui - tk	?	no author	?
ui - qt	Darren	has author	?
backend - pdf	Jouni ?	no author	?
backend - ps	Darren	has author	?
backend - svg	?	no author	?
backend - agg	?	no author	?
backend - cairo	?	no author	?

Here is the ouline for the dev guide, much less fleshed out

Developer's guide unit	Author	Status	Reviewer
the renderer	John	has author	Michael ?
the canvas	John	has author	?
the artist	John	has author	?
transforms	Michael	submitted	John
documenting mpl	Darren	submitted	John, Eric, Mike?
coding guide	John	complete	Eric
and_much_more	?	?	?

We also have some work to do converting docstrings to ReST for the API Reference. Please be sure to follow the few guidelines described in *Formatting*. Once it is converted, please include the module in the API documentation and update the status in the table to "converted". Once docstring conversion is complete and all the modules are available in the docs, we can figure out how best to organize the API Reference and continue from there.

Module	Author	Status
backend_agg		needs conversion
backend_cairo		needs conversion
backend_cocoa		needs conversion
backend_emf		needs conversion
backend_fltkagg		needs conversion
backend_gdk		needs conversion
backend_gtk		needs conversion
backend_gtkagg		needs conversion
backend_gtkcairo		needs conversion

Continued on next page

Table 34.2 – continued from previous page

Module	Author	Status
backend_mixed		needs conversion
backend_pdf		needs conversion
backend_ps	Darren	needs conversion
backend_qt	Darren	needs conversion
backend_qtagg	Darren	needs conversion
backend_qt4	Darren	needs conversion
backend_qt4agg	Darren	needs conversion
backend_svg		needs conversion
backend_template		needs conversion
backend_tkagg		needs conversion
backend_wx		needs conversion
backend_wxagg		needs conversion
backends/tkagg		needs conversion
config/checkdep	Darren	needs conversion
config/cutils	Darren	needs conversion
config/mplconfig	Darren	needs conversion
config/mpltraits	Darren	needs conversion
config/rcparams	Darren	needs conversion
config/rcsetup	Darren	needs conversion
config/tconfig	Darren	needs conversion
config/verbose	Darren	needs conversion
projections/__init__	Mike	converted
projections/geo	Mike	converted (not included–experimental)
projections/polar	Mike	converted
afm		converted
artist		converted
axes		converted
axis		converted
backend_bases		converted
cbook		converted
cm		converted
collections		converted
colorbar		converted
colors		converted
contour		needs conversion
dates	Darren	needs conversion
dviread	Darren	needs conversion
figure	Darren	needs conversion
finance	Darren	needs conversion
font_manager	Mike	converted

Continued on next page

Table 34.2 – continued from previous page

Module	Author	Status
fontconfig_pattern	Mike	converted
image		needs conversion
legend		needs conversion
lines	Mike & ???	converted
mathtext	Mike	converted
mlab	John/Mike	converted
mpl		N/A
patches	Mike	converted
path	Mike	converted
pylab		N/A
pyplot		converted
quiver		needs conversion
rcsetup		needs conversion
scale	Mike	converted
table		needs conversion
texmanager	Darren	needs conversion
text	Mike	converted
ticker	John	converted
transforms	Mike	converted
type1font		needs conversion
units		needs conversion
widgets		needs conversion

And we might want to do a similar table for the FAQ, but that may also be overkill...

If you agree to author a unit, remove the question mark by your name (or add your name if there is no candidate), and change the status to "has author". Once you have completed draft and checked it in, you can change the status to "submitted" and try to find a reviewer if you don't have one. The reviewer should read your chapter, test it for correctness (eg try your examples) and change the status to "complete" when done.

You are free to lift and convert as much material from the web site or the existing latex user's guide as you see fit. The more the better.

The UI chapters should give an example or two of using mpl with your GUI and any relevant info, such as version, installation, config, etc... The backend chapters should cover backend specific configuration (eg PS only options), what features are missing, etc...

Please feel free to add units, volunteer to review or author a chapter, etc...

It is probably easiest to be an editor. Once you have signed up to be an editor, if you have an author pester the author for a submission every so often. If you don't have an author, find one, and then pester them! Your only two responsibilities are getting your author to produce and checking their work, so don't be shy. You *do not* need to be an expert in the subject you are editing – you should know something about it and be willing to read, test, give feedback and pester!

34.1 Reviewer notes

If you want to make notes for the authorwhen you have reviewed a submission, you can put them here. As the author cleans them up or addresses them, they should be removed.

34.1.1 mathtext user's guide– reviewed by JDH

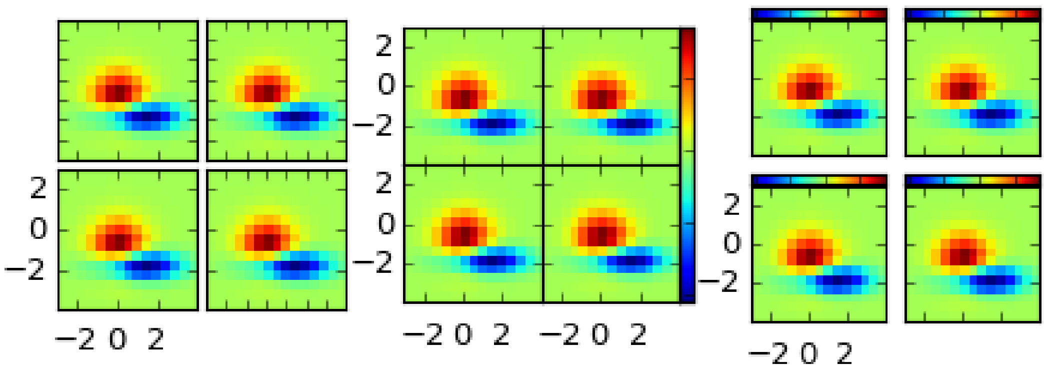
This looks good (see *Writing mathematical expressions*) – there are a few minor things to close the book on this chapter:

1. **The main thing to wrap this up is getting the mathtext module** ported over to rest and included in the API so the links from the user's guide tutorial work.
   - There's nothing in the mathtext module that I really consider a "public" API (i.e. that would be useful to people just doing plots). If mathtext.py were to be documented, I would put it in the developer's docs. Maybe I should just take the link in the user's guide out. - MGD
2. This section might also benefit from a little more detail on the customizations that are possible (eg an example fleshing out the rc options a little bit). Admittedly, this is pretty clear from readin ghte rc file, but it might be helpful to a newbie.
   - The only rcParam that is currently useful is mathtext.fontset, which is documented here. The others only apply when mathtext.fontset == 'custom', which I'd like to declare "unsupported". It's really hard to get a good set of math fonts working that way, though it might be useful in a bind when someone has to use a specific wacky font for mathtext and only needs basics, like sub/superscripts. - MGD
3. There is still a TODO in the file to include a complete list of symbols
   - Done. It's pretty extensive, thanks to STIX... - MGD

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)

CHAPTER THIRTYFIVE

OVERVIEW OF AXESGRID TOOLKIT

35.1 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.

35.2 AXES_GRID1

35.2.1 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.

- The position of each axes is determined at the drawing time (see 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).
- 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,

Name	Default	Description
fig		
rect		
nrows_ncols		number of rows and cols. e.g. (2,2)
ngrids	None	number of grids. nrows x ncols if None
direction	"row"	increasing direction of axes number. [row|column]
axes_pad	0.02	pad between axes in inches
add_all	True	Add axes to figures if True
share_all	False	xaxis & yaxis of all axes are shared if True
aspect	True	aspect of axes
label_mode	"L"	location of tick labels thaw will be displayed. "1" (only the lower left axes), "L" (left most and bottom most axes), or "all".
cbar_mode	None	[None|single|each]
cbar_location	"right"	[right|top]
cbar_pad	None	pad between image axes and colorbar axes
cbar_size	"5%"	size of the colorbar
axes_class	None	

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]
grid[2]	grid[3]

For "column",

grid[0]	grid[2]
grid[1]	grid[3]

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.

35.2.2 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.

35.2.3 colorbar whose height (or width) in sync with the master axes

scatter_hist.py with AxesDivider

The "scatter_hist.py" example in mpl can be rewritten using *make_axes_locatable*.

```
axScatter = subplot(111)
axScatter.scatter(x, y)
axScatter.set_aspect(1.)

# create new axes on the right and on the top of the current axes.
divider = make_axes_locatable(axScatter)
axHistx = divider.append_axes("top", size=1.2, pad=0.1, sharex=axScatter)
axHisty = divider.append_axes("right", size=1.2, pad=0.1, sharey=axScatter)

# the scatter plot:
# histograms
bins = np.arange(-lim, lim + binwidth, binwidth)
axHistx.hist(x, bins=bins)
axHisty.hist(y, bins=bins, orientation='horizontal')
```

See the full source code below.

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.

35.2.4 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_suplot* or *host_axes* command.

Example 1. twinx

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.

```
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.

35.2.5 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.

35.2.6 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,:

```
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,

```
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.

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")
```

35.3 AXISARTIST

35.3.1 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,

```
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

```
ax = AA.Subplot(fig, 111)
fig.add_subplot(ax)
```

For example, you can hide the right, and top axis by

```
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 an horizontal axis at y=0 (in data coordinate).

```
ax.axis["y=0"] = ax.new_floating_axis(nth_coord=0, value=0)
```

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,

```
import mpl_tookits.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.

35.3.2 Curvilinear Grid

The motivation behind the AxisArtist module is to support curvilinear grid and ticks.

See *AXISARTIST namespace* for more details.

35.3.3 Floating Axes

This also support a Floating Axes whose outer axis are defined as floating axis.

CHAPTER

THIRTYSIX

THE MATPLOTLIB AXESGRID TOOLKIT USER'S GUIDE

Release 1.3.x

Date April 01, 2013

36.1 AxesDivider

The axes_divider module provide helper classes to adjust the axes positions of set of images in the drawing time.

- `axes_size` provides a classes 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 calculates 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,:

```
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:

```
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.:

```
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.:

```
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,

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.

36.2 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_tootlkits.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

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.

36.2.1 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)
```

AxisAritst 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.

36.2.2 HowTo

1. Changing tick locations and label.

   Same as the original mpl's axes.:

   ```
   ax.set_xticks([1,2,3])
   ```

2. Changing axis properties like color, etc.

   Change the properties of appropriate artists. For example, to change the color of the ticklabels:

   ```
   ax.axis["left"].major_ticklabels.set_color("r")
   ```

3. To change the attributes of multiple axis:

   ```
   ax.axis["left","bottom"].major_ticklabels.set_color("r")
   ```

   or to change the attributes of all axis:

   ```
   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.

36.2.3 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.

```
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.

   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 (-).

3. same for the *label_direction*

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.

?	?	left	bottom	right	top
axislabel	direction	'-'	'+'	'+'	'-'
axislabel	rotation	180	0	0	180
axislabel	va	center	top	center	bottom
axislabel	ha	right	center	right	center
ticklabel	direction	'-'	'+'	'+'	'-'
ticklabels	rotation	90	0	-90	180
ticklabel	ha	right	center	right	center
ticklabel	va	center	baseline	center	baseline

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.

```
ax1.axis["left"].set_axis_direction("top")
```

changes axis_direction of all the associated artist with the "left" axis, while

```
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.

```
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).

```
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

```
ax.axis["left"].major_ticklabels.set_pad(10)
```

Or ticklabels and axis-label

```
ax.axis["left"].label.set_pad(10)
```

36.2.4 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 defines 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.

```
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.

```
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
# Anthing you draw in ax2 will match the ticks and grids of ax1.
ax1.parasites.append(ax2)
```

36.2.5 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.:

```
# 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.

36.2.6 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

CHAPTER

THIRTYSEVEN

THE MATPLOTLIB AXESGRID TOOLKIT API

Release 1.3.x

Date April 01, 2013

37.1 mpl_toolkits.axes_grid.axes_size

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*)
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*)
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*.

```
>>> s = Fraction(0.3, AxesX(ax))
```

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*.

`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.:

```
>>> a = Size.from_any(1.2) # => Size.Fixed(1.2)
>>> Size.from_any("50%", a) # => Size.Fraction(0.5, a)
```

37.2 `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 to as the axes_locator of the axes.

Parameters

- **fig** – matplotlib figure
- **pos** – position (tuple of 4 floats) of the rectangle that will be divided.
- **horizontal** – list of sizes (`axes_size`) for horizontal division
- **vertical** – list of sizes (`axes_size`) for vertical division
- **aspect** – if True, the overall rectangular area is reduced so that the relative part of the horizontal and vertical scales have same scale.
- **anchor** – Determine how the reduced rectangle is placed when aspect is True.

`add_auto_adjustable_area`(*use_axes, pad=0.1, adjust_dirs=['left', 'right', 'bottom', 'top']*)

`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

- **nx1** (*nx,*) – 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.
- **ny1** (*ny,*) – same as nx and nx1, but for row positions.

new_locator(*nx*, *ny*, *nx1=None*, *ny1=None*)
returns a new locator (mpl_toolkits.axes_grid.axes_divider.AxesLocator) for specified cell.

Parameters

- **nx1** (*nx,*) – 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.
- **ny1** (*ny,*) – same as nx and nx1, but for row positions.

set_anchor(*anchor*)

Parameters anchor – anchor position

value	description
'C'	Center
'SW'	bottom left
'S'	bottom
'SE'	bottom right
'E'	right
'NE'	top right
'N'	top
'NW'	top left
'W'	left

set_aspect(*aspect=False*)

Parameters anchor – True or False

set_horizontal(*h*)

Parameters horizontal – list of sizes (axes_size) for horizontal division

.

set_locator(*_locator*)

set_position(*pos*)

set the position of the rectangle.

Parameters pos – position (tuple of 4 floats) of the rectangle that will be divided.

set_vertical(*v*)

Parameters horizontal – list of sizes (axes_size) for horizontal division

.

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** – An instance of AxesDivider class.
- **nx1** (*nx,*) – 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 is specified.
- **ny1** (*ny,*) – 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.

fig is a matplotlib.figure.Figure instance.

args is the tuple (*numRows*, *numCols*, *plotNum*), where the array of subplots in the figure has dimensions *numRows*, *numCols*, and where *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, eg. from 1,1,1 to 2,2,3

get_geometry()
get the subplot geometry, eg 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

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** – A width of the axes. A axes_size instance or if float or string is given, *from_any* function is used to create one, with *ref_size* set to AxesX instance of the current axes.
- **pad** – pad between the axes. It takes same argument as *size*.
- **pack_start** – 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.

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. Otherwise, the same class of the main axes will be used.

new_vertical(*size*, *pad=None*, *pack_start=False*, ***kwargs*)
Add a new axes on the top (or bottom) side of the main axes.

Parameters

- **size** – A height of the axes. A `axes_size` instance or if float or string is given, *from_any* function is used to create one, with *ref_size* set to AxesX instance of the current axes.
- **pad** – pad between the axes. It takes same argument as *size*.
- **pack_start** – If False, the new axes is appended at the end of the list, i.e., it became the top-most axes. If True, it is inserted at the start of the list, and becomes the bottom-most axes.

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. Otherwise, the same class of the main axes will be used.

37.3 mpl_toolkits.axes_grid.axes_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:

Keyword	Default	Description
direction	“row”	[“row” | “column”]
axes_pad	0.02	float| pad between axes given in inches
add_all	True	[True | False]
share_all	False	[True | False]
share_x	True	[True | False]
share_y	True	[True | False]
label_mode	“L”	[“L” | “1” | “all”]
axes_class	None	a type object which must be a subclass of `Axes`

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:

Keyword	Default	Description
direction	"row"	["row" | "column"]
axes_pad	0.02	float| pad between axes given in inches
add_all	True	[True | False]
share_all	False	[True | False]
aspect	True	[True | False]
label_mode	"L"	["L" | "1" | "all"]
cbar_mode	None	["each" | "single" | "edge"]
cbar_location	"right"	["left" | "right" | "bottom" | "top"]
cbar_pad	None	
cbar_size	"5%"	
cbar_set_cax	True	[True | False]
axes_class	None	a type object which must be a subclass of `Axes`

cbar_set_cax [if True, each axes in the grid has a cax] attribute that is bind to associated cbar_axes.

37.4 `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].

property	left	bottom	right	top
ticklabels location	"-"	"+"	"+"	"-"
axislabel location	"-"	"+"	"+"	"-"
ticklabels angle	90	0	-90	180
ticklabel va	center	baseline	center	baseline
ticklabel ha	right	center	right	center
axislabel angle	180	0	0	180
axislabel va	center	top	center	bottom
axislabel ha	right	center	right	center

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.

ACCEPTS: ["+" | "-"]

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.

ACCEPTS: ["+" | "-"]

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.

property	left	bottom	right	top
axislabel angle	180	0	0	180
axislabel va	center	top	center	bottom
axislabel ha	right	center	right	center

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].

property	left	bottom	right	top
ticklabels angle	90	0	-90	180
ticklabel va	center	baseline	center	baseline
ticklabel ha	right	center	right	center

Note that the text angles are actually relative to (90 + angle of the direction to the ticklabel), which gives 0 for bottom axis.

Part V

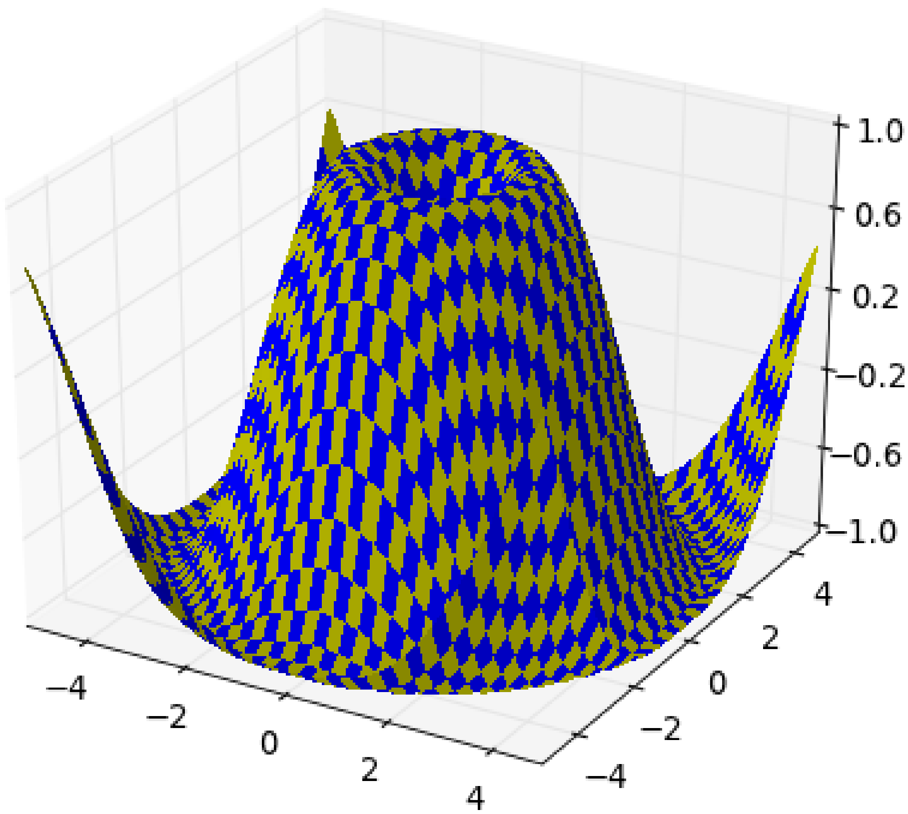
mplot3d

CHAPTER

THIRTYEIGHT

MATPLOTLIB MPLOT3D TOOLKIT

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.

38.1 mplot3d tutorial

Contents

38.1.1 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`:

```
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)`.

38.1.2 Line plots

`Axes3D.`**`plot`**(*xs*, *ys*, **args*, ***kwargs*)

Plot 2D or 3D data.

Argument	Description
xs, *ys*	X, y coordinates of vertices
zs	z value(s), either one for all points or one for each point.
zdir	Which direction to use as z ('x', 'y' or 'z') when plotting a 2D set.

Other arguments are passed on to `plot()`

38.1.3 Scatter plots

`Axes3D.scatter`(*xs*, *ys*, *zs=0*, *zdir='z'*, *s=20*, *c='b'*, **args*, ***kwargs*)

Create a scatter plot.

Argument	Description
xs, *ys*	Positions of data points.
zs	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.
zdir	Which direction to use as z ('x', 'y' or 'z') when plotting a 2D set.
s	size in points^2. It is a scalar or an array of the same length as *x* and *y*.
c	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.

Keyword arguments are passed on to `scatter()`.

Returns a `Patch3DCollection`

38.1.4 Wireframe plots

`Axes3D.plot_wireframe`(*X*, *Y*, *Z*, **args*, ***kwargs*)

Plot a 3D wireframe.

Argument	Description
X, *Y*,	Data values as 2D arrays
Z	
rstride	Array row stride (step size)
cstride	Array column stride (step size)

Keyword arguments are passed on to `LineCollection`.

Returns a `Line3DCollection`

38.1.5 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.

Argument	Description
X, Y, Z	Data values as 2D arrays
rstride	Array row stride (step size)
cstride	Array column stride (step size)
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`

38.1.6 Tri-Surface plots

`Axes3D.plot_trisurf(*args, **kwargs)`

Argument	Description
X, Y, Z	Data values as 1D arrays
color	Color of the surface patches
cmap	A colormap for the surface 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

The (optional) triangulation can be specified in one of two ways; either:

```
plot_trisurf(triangulation, ...)
```

where triangulation is a `Triangulation` object, or:

```
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:

```
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.

38.1.7 Contour plots

`Axes3D.contour`(*X*, *Y*, *Z*, **args*, ***kwargs*)

Create a 3D contour plot.

Argument	Description
X, *Y*,	Data values as numpy.arrays
Z	
extend3d	Whether to extend contour in 3D (default: False)
stride	Stride (step size) for extending contour
zdir	The direction to use: x, y or z (default)
offset	If specified plot a projection of the contour lines on this position in plane normal to zdir

The positional and other keyword arguments are passed on to `contour()`

Returns a `contour`

38.1.8 Filled contour plots

`Axes3D.contourf`(*X*, *Y*, *Z*, **args*, ***kwargs*)

Create a 3D contourf plot.

Argument	Description
X, *Y*,	Data values as numpy.arrays
Z	
zdir	The direction to use: x, y or z (default)
offset	If specified plot a projection of the filled contour on this position in plane normal to zdir

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.

38.1.9 Polygon plots

`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
- LineColleciton
- PatchCollection

38.1.10 Bar plots

`Axes3D.bar`(*left*, *height*, *zs=0*, *zdir='z'*, **args*, ***kwargs*)

Add 2D bar(s).

Argument	Description
left	The x coordinates of the left sides of the bars.
height	The height of the bars.
zs	Z coordinate of bars, if one value is specified they will all be placed at the same z.
zdir	Which direction to use as z ('x', 'y' or 'z') when plotting a 2D set.

Keyword arguments are passed onto `bar()`.

Returns a `Patch3DCollection`

38.1.11 2D plots in 3D

38.1.12 Text

`Axes3D.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.

38.1.13 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.

38.2 mplot3d API

Contents

38.2.1 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*)

Bases: `matplotlib.axes.Axes`

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
- LineColleciton
- 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. Therfore, '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. However, it currently does not do anything.

bar(*left*, *height*, *zs=0*, *zdir='z'*, **args*, ***kwargs*)
Add 2D bar(s).

Argument	Description
left	The x coordinates of the left sides of the bars.
height	The height of the bars.
zs	Z coordinate of bars, if one value is specified they will all be placed at the same z.
zdir	Which direction to use as z ('x', 'y' or 'z') when plotting a 2D set.

Keyword arguments are passed onto `bar()`.

Returns a `Patch3DCollection`

bar3d(*x*, *y*, *z*, *dx*, *dy*, *dz*, *color='b'*, *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 and disable mouse button callbacks.

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.

Argument	Description
X, *Y*,	Data values as numpy.arrays
Z	
extend3d	Whether to extend contour in 3D (default: False)
stride	Stride (step size) for extending contour
zdir	The direction to use: x, y or z (default)
offset	If specified plot a projection of the contour lines on this position in plane normal to zdir

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.

Argument	Description
X, *Y*,	Data values as numpy.arrays
Z	
extend3d	Whether to extend contour in 3D (default: False)
stride	Stride (step size) for extending contour
zdir	The direction to use: x, y or z (default)
offset	If specified plot a projection of the contour lines on this position in plane normal to zdir

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.

Argument	Description
X, *Y*,	Data values as numpy.arrays
Z	
zdir	The direction to use: x, y or z (default)
offset	If specified plot a projection of the filled contour on this position in plane normal to zdir

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.

Argument	Description
X, *Y*,	Data values as numpy.arrays
Z	
zdir	The direction to use: x, y or z (default)
offset	If specified plot a projection of the filled contour on this position in plane normal to zdir

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.

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 [*left*, *right*] Changed in version 1.1.0: This function now correctly refers to the 3D x-limits

`get_xlim3d()`
Get the x-axis range [*left*, *right*] Changed in version 1.1.0: This function now correctly refers to the 3D x-limits

`get_ylim()`
Get the y-axis range [*bottom*, *top*] Changed in version 1.1.0: This function now correctly refers to the 3D y-limits.

`get_ylim3d()`
Get the y-axis range [*bottom*, *top*] 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.

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.

Argument	Description
rotate_btn	The integer or list of integers specifying which mouse button or buttons to use for 3D rotation of the axes. Default = 1.
zoom_btn	The integer or list of integers specifying which mouse button or buttons to use to zoom the 3D axes. Default = 3.

name = '3d'

plot(*xs*, *ys*, **args*, ***kwargs*)

Plot 2D or 3D data.

Argument	Description
xs, *ys*	X, y coordinates of vertices
zs	z value(s), either one for all points or one for each point.
zdir	Which direction to use as z ('x', 'y' or 'z') when plotting a 2D set.

Other arguments are passed on to `plot()`

plot3D(*xs*, *ys*, **args*, ***kwargs*)

Plot 2D or 3D data.

Argument	Description
xs, *ys*	X, y coordinates of vertices
zs	z value(s), either one for all points or one for each point.
zdir	Which direction to use as z ('x', 'y' or 'z') when plotting a 2D set.

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.

Argument	Description
X, *Y*, *Z*	Data values as 2D arrays
rstride	Array row stride (step size)
cstride	Array column stride (step size)
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*)

Argument	Description
X, *Y*, *Z*	Data values as 1D arrays
color	Color of the surface patches
cmap	A colormap for the surface 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

The (optional) triangulation can be specified in one of two ways; either:

```
plot_trisurf(triangulation, ...)
```

where triangulation is a `Triangulation` object, or:

```
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:

```
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.

Argument	Description
X, *Y*,	Data values as 2D arrays
Z	
rstride	Array row stride (step size)
cstride	Array column stride (step size)

Keyword arguments are passed on to `LineCollection`.

Returns a `Line3DCollection`

scatter(*xs*, *ys*, *zs=0*, *zdir='z'*, *s=20*, *c='b'*, **args*, ***kwargs*)

Create a scatter plot.

Argument	Description
xs, ys	Positions of data points.
zs	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.
zdir	Which direction to use as z ('x', 'y' or 'z') when plotting a 2D set.
s	size in points^2. It is a scalar or an array of the same length as *x* and *y*.
c	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.

Keyword arguments are passed on to `scatter()`.

Returns a `Patch3DCollection`

scatter3D(*xs, ys, zs=0, zdir='z', s=20, c='b', *args, **kwargs*)
Create a scatter plot.

Argument	Description
xs, ys	Positions of data points.
zs	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.
zdir	Which direction to use as z ('x', 'y' or 'z') when plotting a 2D set.
s	size in points^2. It is a scalar or an array of the same length as *x* and *y*.
c	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.

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.

set_axis_off()

set_axis_on()

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*

ACCEPTS: [*True* | *False*] New in version 1.1.0: This function was added for completeness.

set_frame_on(*b*)
Set whether the 3D axes panels are drawn

ACCEPTS: [*True* | *False*] New in version 1.1.0.

set_title(*label*, *fontdict=None*, ***kwargs*)

set_top_view()

set_xlim(**args*, ***kwargs*)
Set 3D x limits.

See `matplotlib.axes.Axes.set_xlim()` for full documentation.

set_xlim3d(**args*, ***kwargs*)
Set 3D x limits.

See `matplotlib.axes.Axes.set_xlim()` for full documentation.

set_xscale(*value*, ***kwargs*)
Call signature:

```
set_xscale(value)
```

Set the scaling of the x-axis: 'linear' | 'log' | 'symlog'

ACCEPTS: ['linear' | 'log' | '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.

'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(**args*, ***kwargs*)
Set 3D y limits.

See `matplotlib.axes.Axes.set_ylim()` for full documentation.

set_ylim3d(**args*, ***kwargs*)
Set 3D y limits.

See `matplotlib.axes.Axes.set_ylim()` for full documentation.

set_yscale(*value*, ***kwargs*)
Call signature:

```
set_yscale(value)
```

Set the scaling of the y-axis: 'linear' | 'log' | 'symlog'

ACCEPTS: ['linear' | 'log' | '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.

'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_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.

Note: Currently, *labelpad* does not have an effect on the labels.

---

set_zlim(**args*, ***kwargs*)
Set 3D z limits.

See `matplotlib.axes.Axes.set_ylim()` for full documentation.

set_zlim3d(**args*, ***kwargs*)

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*)

call signature:

```
set_zscale(value)
```

Set the scaling of the z-axis: 'linear' | 'log' | 'symlog'

ACCEPTS: ['linear' | 'log' | '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.

'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.

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.

Call signature:

```
text(x, y, s, fontdict=None, **kwargs)
```

Add text in string *s* to axis at location *x*, *y*, data coordinates.

Keyword arguments:

***fontdict*:** A dictionary to override the default text properties. If *fontdict* is *None*, the defaults are determined by your rc parameters.

***withdash*: [*False* | *True*]** Creates a `TextWithDash` instance instead of a `Text` instance.

Individual keyword arguments can be used to override any given parameter:

```
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:

```
text(0.5, 0.5,'matplotlib',
     horizontalalignment='center',
     verticalalignment='center',
     transform = ax.transAxes)
```

You can put a rectangular box around the text instance (eg. to set a background color) by using the keyword *bbox*. *bbox* is a dictionary of `matplotlib.patches.Rectangle` properties. For example:

```
text(x, y, s, bbox=dict(facecolor='red', alpha=0.5))
```

Valid kwargs are `Text` properties:

Property	Description
`agg_filter`	unknown
`alpha`	float (0.0 transparent through 1.0 opaque)
`animated`	[True | False]
`axes`	an `Axes` instance
`backgroundcolor`	any matplotlib color
`bbox`	rectangle prop dict
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color`	any matplotlib color
`contains`	a callable function
`family` or fontfamily or fontname or name	[FONTNAME | 'serif' | 'sans-serif' | 'cursive' | 'fantasy' | 'mono
`figure`	a `matplotlib.figure.Figure` instance
`fontproperties` or font_properties	a `matplotlib.font_manager.FontProperties` instance
`gid`	an id string
`horizontalalignment` or ha	['center' | 'right' | 'left']
`label`	string or anything printable with '%s' conversion.
`linespacing`	float (multiple of font size)

Table 38.1 – cont

Property	Description
lod	[True | False]
multialignment	['left' | 'right' | 'center']
path_effects	unknown
picker	[None|float|boolean|callable]
position	(x,y)
rasterized	[True | False | None]
rotation	[angle in degrees | 'vertical' | 'horizontal']
rotation_mode	unknown
size or fontsize	[size in points | 'xx-small' | 'x-small' | 'small' | 'medium' | 'large
snap	unknown
stretch or fontstretch	[a numeric value in range 0-1000 | 'ultra-condensed' | 'extra-conc
style or fontstyle	['normal' | 'italic' | 'oblique']
text	string or anything printable with '%s' conversion.
transform	Transform instance
url	a url string
variant or fontvariant	['normal' | 'small-caps']
verticalalignment or va or ma	['center' | 'top' | 'bottom' | 'baseline']
visible	[True | False]
weight or fontweight	[a numeric value in range 0-1000 | 'ultralight' | 'light' | 'normal' |
x	float
y	float
zorder	any number

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 Axed3D 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(*X*, *Y*, *Z*, **args*, ***kwargs*)

Create a 3D contour plot.

Argument	Description
X, *Y*,	Data values as numpy.arrays
Z	
extend3d	Whether to extend contour in 3D (default: False)
stride	Stride (step size) for extending contour
zdir	The direction to use: x, y or z (default)
offset	If specified plot a projection of the contour lines on this position in plane normal to zdir

Other keyword arguments are passed on to `tricontour()`

Returns a `contour` New in version 1.1.0.

tricontourf(*X*, *Y*, *Z*, *offset=None*, *zdir='z'*, **args*, ***kwargs*)

Create a 3D contourf plot.

Argument	Description
X, *Y*,	Data values as numpy.arrays
Z	
zdir	The direction to use: x, y or z (default)
offset	If specified plot a projection of the contour lines on this position in plane normal to zdir

Other keyword arguments are passed on to `tricontour()`

Returns a `contour` New in version 1.1.0.

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**()

38.2.2 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.

---

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*)

class mpl_toolkits.mplot3d.axis3d.**XAxis**(*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.**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.

38.2.3 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.

`do_3d_projection`(*renderer*)

draw(*renderer*)

set_3d_properties(*zs*, *zdir*)

set_sort_zpos(*val*)
Set the position to use for z-sorting.

class mpl_toolkits.mplot3d.art3d.**PathPatch3D**(*path*, ***kwargs*)
Bases: mpl_toolkits.mplot3d.art3d.Patch3D

3D PathPatch object.

do_3d_projection(*renderer*)

set_3d_properties(*path*, *zs=0*, *zdir='z'*)

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*)

get_edgecolor()

get_edgecolors()

get_facecolor()

get_facecolors()

get_vector(*segments3d*)
Optimize points for projection

set_3d_properties()

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_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'*)
Convert a PatchCollection to a Patch3DCollection object.

`mpl_toolkits.mplot3d.art3d.`**`path_to_3d_segment`**(*path*, *zs=0*, *zdir='z'*)
Convert a path to a 3D segment.

`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.`**`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

`mpl_toolkits.mplot3d.art3d.`**`text_2d_to_3d`**(*obj*, *z=0*, *zdir='z'*)
Convert a Text to a Text3D object.

`mpl_toolkits.mplot3d.art3d.`**`zalpha`**(*colors*, *zs*)
Modify the alphas of the color list according to depth

38.2.4 proj3d

Various transforms used for by the 3D code

`mpl_toolkits.mplot3d.proj3d.`**`inv_transform`**(*xs*, *ys*, *zs*, *M*)

`mpl_toolkits.mplot3d.proj3d.`**`line2d`**(*p0*, *p1*)
Return 2D equation of line in the form ax+by+c = 0

`mpl_toolkits.mplot3d.proj3d.`**`line2d_dist`**(*l*, *p*)
Distance from line to point line is a tuple of coefficients a,b,c

`mpl_toolkits.mplot3d.proj3d.`**`line2d_seg_dist`**(*p1*, *p2*, *p0*)
distance(s) from line defined by p1 - p2 to point(s) p0

p0[0] = x(s) p0[1] = y(s)

intersection point p = p1 + u*(p2-p1) and intersection point lies within segment if u is between 0 and 1

`mpl_toolkits.mplot3d.proj3d.`**`mod`**(*v*)
3d vector length

`mpl_toolkits.mplot3d.proj3d.`**`persp_transformation`**(*zfront*, *zback*)

`mpl_toolkits.mplot3d.proj3d.`**`proj_points`**(*points*, *M*)

`mpl_toolkits.mplot3d.proj3d.`**`proj_trans_clip_points`**(*points*, *M*)

`mpl_toolkits.mplot3d.proj3d.`**`proj_trans_points`**(*points*, *M*)

mpl_toolkits.mplot3d.proj3d.**proj_transform**(*xs*, *ys*, *zs*, *M*)
Transform the points by the projection matrix

mpl_toolkits.mplot3d.proj3d.**proj_transform_clip**(*xs*, *ys*, *zs*, *M*)
Transform the points by the projection matrix and return the clipping result returns txs,tys,tzs,tis

mpl_toolkits.mplot3d.proj3d.**proj_transform_vec**(*vec*, *M*)

mpl_toolkits.mplot3d.proj3d.**proj_transform_vec_clip**(*vec*, *M*)

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=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*)

38.3 mplot3d FAQ

38.3.1 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 recomended 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.

38.3.2 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 occurance of this artifact, it is currently an intractable problem, and can not 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) can not 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.

38.3.3 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.

CHAPTER

THIRTYNINE

BASEMAP

Plots data on map projections, with continental and political boundaries, see basemap docs.

CHAPTER

FORTY

GTK TOOLS

mpl_toolkits.gtktools provides some utilities for working with GTK. This toolkit ships with matplotlib, but requires pygtk.

CHAPTER

FORTYONE

EXCEL TOOLS

mpl_toolkits.exceltools provides some utilities for working with Excel. This toolkit ships with matplotlib, but requires xlwt

CHAPTER

FORTYTWO

NATGRID

mpl_toolkits.natgrid is an interface to natgrid C library for gridding irregularly spaced data. This requires a separate installation of the natgrid toolkit from the sourceforge download page.

CHAPTER

FORTYTHREE

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.

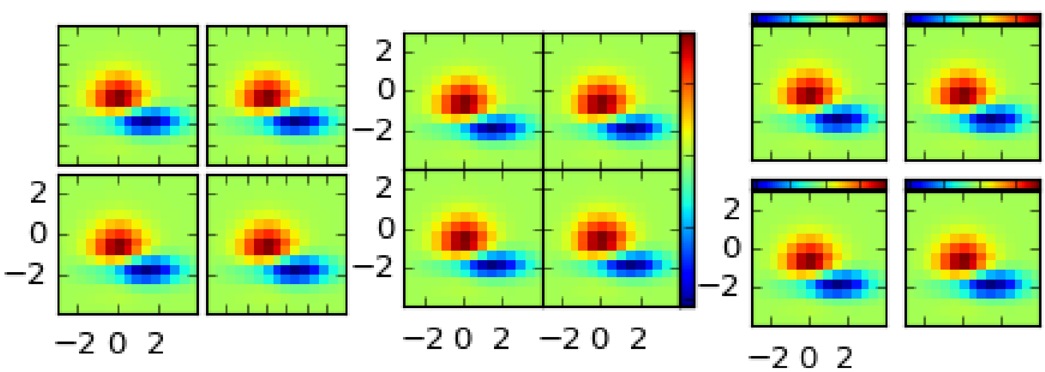
See *mplot3d* for more documentation and examples.

CHAPTER

FORTYFOUR

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.



See *Matplotlib AxesGrid Toolkit* for documentations.

Part VII

The Matplotlib API

CHAPTER

FORTYFIVE

PLOTTING COMMANDS SUMMARY

`matplotlib.pyplot.plotting()`

Function	Description
acorr	Plot the autocorrelation of *x*.
annotate	Create an annotation: a piece of text referring to a data point.
arrow	Add an arrow to the axes.
autoscale	Autoscale the axis view to the data (toggle).
axes	Add an axes to the figure.
axhline	Add a horizontal line across the axis.
axhspan	Add a horizontal span (rectangle) across the axis.
axis	Set or get the axis properties.:: >>> axis() returns the current axes limits `[xmin, xmax`
axvline	Add a vertical line across the axes.
axvspan	Add a vertical span (rectangle) across the axes.
bar	Make a bar plot.
barbs	Plot a 2-D field of barbs.
barh	Make a horizontal bar plot.
box	Turn the axes box on or off.
boxplot	Make a box and whisker plot.
broken_barh	Plot horizontal bars.
cla	Clear the current axes.
clabel	Label a contour plot.
clf	Clear the current figure.
clim	Set the color limits of the current image.
close	Close a figure window.
cohere	Plot the coherence between *x* and *y*.
colorbar	Add a colorbar to a plot.
contour	Plot contours.
contourf	Plot contours.
csd	Plot cross-spectral density.
delaxes	Remove an axes from the current figure.

Function	Description
draw	Redraw the current figure.
errorbar	Plot an errorbar graph.
figimage	Adds a non-resampled image to the figure.
figlegend	Place a legend in the figure.
figtext	Add text to figure.
figure	Create a new figure.
fill	Plot filled polygons.
fill_between	Make filled polygons between two curves.
fill_betweenx	Make filled polygons between two horizontal curves.
findobj	Find artist objects.
gca	Return the current axis instance.
gcf	Return a reference to the current figure.
gci	Get the current colorable artist.
get_figlabels	Return a list of existing figure labels.
get_fignums	Return a list of existing figure numbers.
grid	Turn the axes grids on or off.
hexbin	Make a hexagonal binning plot.
hist	Plot a histogram.
hist2d	Make a 2D histogram plot.
hlines	Plot horizontal lines.
hold	Set the hold state.
imread	Read an image from a file into an array.
imsave	Save an array as in image file.
imshow	Display an image on the axes.
ioff	Turn interactive mode off.
ion	Turn interactive mode on.
ishold	Return the hold status of the current axes.
isinteractive	Return status of interactive mode.
legend	Place a legend on the current axes.
locator_params	Control behavior of tick locators.
loglog	Make a plot with log scaling on both the *x* and *y* axis.
margins	Set or retrieve autoscaling margins.
matshow	Display an array as a matrix in a new figure window.
minorticks_off	Remove minor ticks from the current plot.
minorticks_on	Display minor ticks on the current plot.
over	Call a function with hold(True).
pause	Pause for *interval* seconds.
pcolor	Create a pseudocolor plot of a 2-D array.
pcolormesh	Plot a quadrilateral mesh.
pie	Plot a pie chart.

Function	Description
plot	Plot lines and/or markers to the Axes.
plot_date	Plot with data with dates.
plotfile	Plot the data in in a file.
polar	Make a polar plot.
psd	Plot the power spectral density.
quiver	Plot a 2-D field of arrows.
quiverkey	Add a key to a quiver plot.
rc	Set the current rc params.
rcdefaults	Restore the default rc params.
rgrids	Get or set the radial gridlines on a polar plot.
savefig	Save the current figure.
sca	Set the current Axes instance to *ax*.
scatter	Make a scatter plot.
sci	Set the current image.
semilogx	Make a plot with log scaling on the *x* axis.
semilogy	Make a plot with log scaling on the *y* axis.
set_cmap	Set the default colormap.
setp	Set a property on an artist object.
show	Display a figure.
specgram	Plot a spectrogram.
spy	Plot the sparsity pattern on a 2-D array.
stackplot	Draws a stacked area plot.
stem	Create a stem plot.
step	Make a step plot.
streamplot	Draws streamlines of a vector flow.
subplot	Create a new axes (subplot).
subplot2grid	Create a subplot in a grid.
subplot_tool	Launch a subplot tool window for a figure.
subplots	Create a figure with a set of subplots already made.
subplots_adjust	Tune the subplot layout.
suptitle	Add a centered title to the figure.
switch_backend	Switch the default backend.
table	Add a table to the current axes.
text	Add text to the axes.
thetagrids	Get or set the theta locations of the gridlines in a polar plot.
tick_params	Change the appearance of ticks and tick labels.
ticklabel_format	Change the ScalarFormatter used by default for linear axes.
tight_layout	Automatically adjust subplot parameters to give specified padding.
title	Set the title of the current axis.
tricontour	Draw contours on an unstructured triangular grid.

Function	Description
tricontourf	Draw contours on an unstructured triangular grid.
tripcolor	Create a pseudocolor plot of an unstructured triangular grid.
triplot	Draw a unstructured triangular grid as lines and/or markers.
twinx	Make a second axes that shares the *x*-axis.
twiny	Make a second axes that shares the *y*-axis.
vlines	Plot vertical lines.
xcorr	Plot the cross correlation between *x* and *y*.
xlabel	Set the *x* axis label of the current axis.
xlim	Get or set the *x* limits of the current axes.
xscale	Set the scaling of the *x*-axis.
xticks	Get or set the *x*-limits of the current tick locations and labels.
ylabel	Set the *y* axis label of the current axis.
ylim	Get or set the *y*-limits of the current axes.
yscale	Set the scaling of the *y*-axis.
yticks	Get or set the *y*-limits of the current tick locations and labels.

`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 (with the exception of `spectral`) derived from those of the same name provided with Matlab:

Colormap	Description
autumn	sequential linearly-increasing shades of red-orange-yellow
bone	sequential increasing black-white color map with a tinge of blue, to emulate X-ray film
cool	linearly-decreasing shades of cyan-magenta
copper	sequential increasing shades of black-copper
flag	repetitive red-white-blue-black pattern (not cyclic at endpoints)
gray	sequential linearly-increasing black-to-white grayscale
hot	sequential black-red-yellow-white, to emulate blackbody radiation from an object at increasing temperatures
hsv	cyclic red-yellow-green-cyan-blue-magenta-red, formed by changing the hue component in the HSV color space
jet	a spectral map with dark endpoints, blue-cyan-yellow-red; based on a fluid-jet simulation by NCSA [1]
pink	sequential increasing pastel black-pink-white, meant for sepia tone colorization of photographs
prism	repetitive red-yellow-green-blue-purple-...-green pattern (not cyclic at endpoints)
spring	linearly-increasing shades of magenta-yellow
summer	sequential linearly-increasing shades of green-yellow
winter	linearly-increasing shades of blue-green
spectral	black-purple-blue-green-yellow-red-white spectrum

For the above list only, you can also set the colormap using the corresponding pylab shortcut interface function, similar to Matlab:

```
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

Colormap	Description
gist_earth	mapmaker's colors from dark blue deep ocean to green lowlands to brown highlands to white mountains
gist_heat	sequential increasing black-red-orange-white, to emulate blackbody radiation from an iron bar as it grows hotter
gist_ncar	pseudo-spectral black-blue-green-yellow-red-purple-white colormap from National Center for Atmospheric Research [2]
gist_rainbow	runs through the colors in spectral order from red to violet at full saturation (like *hsv* but not cyclic)
gist_stern	"Stern special" color table from Interactive Data Language software

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):

Colormap	Description
BrBG	brown, white, blue-green
PiYG	pink, white, yellow-green
PRGn	purple, white, green
PuOr	orange, white, purple
RdBu	red, white, blue
RdGy	red, white, gray
RdYlBu	red, yellow, blue
RdYlGn	red, yellow, green
Spectral	red, orange, yellow, green, blue

ColorBrewer Sequential (luminance decreases monotonically):

[2] Resembles "BkBlAqGrYeOrReViWh200" from NCAR Command Language. See Color Table Gallery

Colormap	Description
Blues	white to dark blue
BuGn	white, light blue, dark green
BuPu	white, light blue, dark purple
GnBu	white, light green, dark blue
Greens	white to dark green
Greys	white to black (not linear)
Oranges	white, orange, dark brown
OrRd	white, orange, dark red
PuBu	white, light purple, dark blue
PuBuGn	white, light purple, dark green
PuRd	white, light purple, dark red
Purples	white to dark purple
RdPu	white, pink, dark purple
Reds	white to dark red
YlGn	light yellow, dark green
YlGnBu	light yellow, light green, dark blue
YlOrBr	light yellow, orange, dark brown
YlOrRd	light yellow, orange, dark red

ColorBrewer Qualitative:

(For plotting nominal data, `ListedColormap` should be used, not `LinearSegmentedColormap`. Different sets of colors are recommended for different numbers of categories. These continuous versions of the qualitative schemes may be removed or converted in the future.)

- Accent
- Dark2
- Paired
- Pastel1
- Pastel2
- Set1
- Set2
- Set3

Other miscellaneous schemes:

Colormap	Description
afmhot	sequential black-orange-yellow-white blackbody spectrum, commonly used in atomic force microscopy
brg	blue-red-green
bwr	diverging blue-white-red
coolwarm	diverging blue-gray-red, meant to avoid issues with 3D shading, color blindness, and ordering of colors [3]
CMRmap	"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]
cubehelix	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.
gnuplot	gnuplot's traditional pm3d scheme (black-blue-red-yellow)
gnuplot2	sequential color printable as gray (black-blue-violet-yellow-white)
ocean	green-blue-white
rainbow	spectral purple-blue-green-yellow-orange-red colormap with diverging luminance
seismic	diverging blue-white-red
terrain	mapmaker's colors, blue-green-yellow-brown-white, originally from IGOR Pro

The following colormaps are redundant and may be removed in future versions. It's recommended to use *gray* or *gray_r* instead, which produce identical output:

Colormap	Description
gist_gray	identical to *gray*
gist_yarg	identical to *gray_r*
binary	identical to *gray_r*

[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

CHAPTER

FORTYSIX

API CHANGES

This chapter is a log of changes to matplotlib that affect the outward-facing API. If updating matplotlib breaks your scripts, this list may help describe what changes may be necessary in your code or help figure out possible sources of the changes you are experiencing.

For new features that were added to matplotlib, please see *What's new in matplotlib*.

46.1 Changes in 1.2.x

- 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:

```
key = figure._make_key(*args, **kwargs)
ispolar = kwargs.pop('polar', False)
```

```
projection = kwargs.pop('projection', None)
if ispolar:
    if projection is not None and projection != 'polar':
        raise ValueError('polar and projection args are inconsistent')
    projection = 'polar'
ax = projection_factory(projection, self, rect, **kwargs)
key = self._make_key(*args, **kwargs)

# is now

projection_class, kwargs, key = \
                   process_projection_requirements(self, *args, **kwargs)
ax = projection_class(self, rect, **kwargs)
```

This change means that third party objects can expose themselves as matplotlib axes by providing a `_as_mpl_axes` method. See *Adding new scales and projections to matplotlib* 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`:

```
class MyTransform(mtrans.Transform):
    def transform(self, xy):
        ...
    transform_non_affine = transform
```

This approach will no longer function correctly and should be changed to:

```
class MyTransform(mtrans.Transform):
    def transform_non_affine(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:

```
>>> 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.],\n       [ 90.,  90.]])')
```

- 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:

```
>>> 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.

46.2 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 kwargs.

- 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".

46.3 Changes beyond 0.99.x

- The default behavior of `matplotlib.axes.Axes.set_xlim()`, `matplotlib.axes.Axes.set_ylim()`, and `matplotlib.axes.Axes.axis()`, and their corresponding pyplot functions, 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 :class:'~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:

```
draw_path_collection(self, master_transform, cliprect, clippath,
                     clippath_trans, paths, all_transforms, offsets,
                     offsetTrans, facecolors, edgecolors, linewidths,
                     linestyles, antialiaseds, urls)

# is now

draw_path_collection(self, gc, master_transform, paths, all_transforms,
                     offsets, offsetTrans, facecolors, edgecolors,
                     linewidths, linestyles, antialiaseds, urls)


draw_quad_mesh(self, master_transform, cliprect, clippath,
               clippath_trans, meshWidth, meshHeight, coordinates,
               offsets, offsetTrans, facecolors, antialiased,
               showedges)

# is now

draw_quad_mesh(self, gc, master_transform, meshWidth, meshHeight,
               coordinates, offsets, offsetTrans, facecolors,
               antialiased, showedges)


draw_image(self, x, y, im, bbox, clippath=None, clippath_trans=None)
```

```
# is now

draw_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.

46.4 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.

46.5 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.

Deprecated	New
pad	borderpad
labelsep	labelspacing
handlelen	handlelength
handlestextsep	handletextpad
axespad	borderaxespad

- 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.afm.AFM.get_fullname()` and `matplotlib.afm.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 NotImplementedError) 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`; `matplotlib.axes.Axes.axesPatch` renamed `matplotlib.axes.Axes.patch`; `matplotlib.axes.Axes.axesFrame` renamed `matplotlib.axes.Axes.frame`. `matplotlib.axes.Axes.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.

46.6 Changes for 0.98.1

- Removed broken `matplotlib.axes3d` support and replaced it with a non-implemented error pointing to 0.91.x

46.7 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.
- Rewrote the `matplotlib.cm.ScalarMappable` callback infrastructure to use `matplotlib.cbook.CallbackRegistry` rather than custom callback handling. Any users of `matplotlib.cm.ScalarMappable.add_observer()` of the `ScalarMappable`

should use the matplotlib.cm.ScalarMappable.callbacks CallbackRegistry instead.

- 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)

46.7.1 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

Old method	New method
`Bbox.get_bounds()`	`transforms.Bbox.bounds`
`Bbox.width()`	`transforms.Bbox.width`
`Bbox.height()`	`transforms.Bbox.height`
`Bbox.intervalx().get_bounds()`	`transforms.Bbox.intervalx`
`Bbox.intervalx().set_bounds()`	[`Bbox.intervalx` is now a property.]
`Bbox.intervaly().get_bounds()`	`transforms.Bbox.intervaly`
`Bbox.intervaly().set_bounds()`	[`Bbox.intervaly` is now a property.]
`Bbox.xmin()`	`transforms.Bbox.x0` or `transforms.Bbox.xmin` [1]
`Bbox.ymin()`	`transforms.Bbox.y0` or `transforms.Bbox.ymin` [1]
`Bbox.xmax()`	`transforms.Bbox.x1` or `transforms.Bbox.xmax` [1]
`Bbox.ymax()`	`transforms.Bbox.y1` or `transforms.Bbox.ymax` [1]
`Bbox.overlaps(bboxes)`	`Bbox.count_overlaps(bboxes)`
`bbox_all(bboxes)`	`Bbox.union(bboxes)` [`transforms.Bbox.union()` is a staticmethod.]
`lbwh_to_bbox(l, b, w, h)`	`Bbox.from_bounds(x0, y0, w, h)` [`transforms.Bbox.from_bounds()` is a staticmethod.]
`inverse_transform_bbox(trans, bbox)`	`Bbox.inverse_transformed(trans)`
`Interval.contains_open(v)`	`interval_contains_open(tuple, v)`
`Interval.contains(v)`	`interval_contains(tuple, v)`
`identity_transform()`	`matplotlib.transforms.IdentityTransform`
`blend_xy_sep_transform(xtrans, ytrans)`	`blended_transform_factory(xtrans, ytrans)`
`scale_transform(xs, ys)`	`Affine2D().scale(xs[, ys])`
`get_bbox_transform(boxin, boxout)`	`BboxTransform(boxin, boxout)` or `BboxTransformFrom(boxin)` or `BboxTransformTo(boxout)`
`Transform.seq_xy_tup(points)`	`Transform.transform(points)`
`Transform.inverse_xy_tup(points)`	`Transform.inverted().transform(points)`

[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`.

matplotlib.axes

Old method	New method
Axes.get_position()	matplotlib.axes.Axes.get_position() [2]
Axes.set_position()	matplotlib.axes.Axes.set_position() [3]
Axes.toggle_log_lineary()	matplotlib.axes.Axes.set_yscale() [4]
Subplot class	removed.

The Polar class has moved to matplotlib.projections.polar.

matplotlib.artist

Old method	New method
Artist.set_clip_path(path)	Artist.set_clip_path(path, transform) [5]

matplotlib.collections

Old method	New method
linestyle	linestyles [6]

matplotlib.colors

Old method	New method
ColorConvertor.to_rgba_list(c)	ColorConvertor.to_rgba_array(c) [matplotlib.colors.ColorConvertor.to_rgba_array() returns an Nx4 Numpy array of RGBA color quadruples.]

matplotlib.contour

Old method	New method
Contour._segments	matplotlib.contour.Contour.get_paths'() [Returns a list of matplotlib.path.Path instances.]

[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 recfactoring 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.

matplotlib.figure

Old method	New method
Figure.dpi.get() / Figure.dpi.set()	matplotlib.figure.Figure.dpi *(a property)*

matplotlib.patches

Old method	New method
Patch.get_verts()	matplotlib.patches.Patch.get_path() [Returns a matplotlib.path.Path instance]

matplotlib.backend_bases

Old method	New method
GraphicsContext.set_clip_rectangle(tuple)	GraphicsContext.set_clip_rectangle(bbox)
GraphicsContext.get_clip_path()	GraphicsContext.get_clip_path() [7]
GraphicsContext.set_clip_path()	GraphicsContext.set_clip_path() [8]

RendererBase

New methods:

- draw_path(self, gc, path, transform, rgbFace)
- draw_markers(self, gc, marker_path, marker_trans, path, trans, rgbFace) <matplotlib.backend_bases.RendererBase.draw_markers()
- draw_path_collection(self, master_transform, cliprect, clippath, clippath_trans, paths, all_transforms, offsets, offsetTrans, facecolors, edgecolors, linewidths, 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

[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_line
- draw_lines
- draw_point
- draw_quad_mesh
- draw_poly_collection
- draw_polygon
- draw_rectangle
- draw_regpoly_collection

46.8 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 rec2gtk() to matplotlib.toolkits.gtktools
- Moved rec2excel() to matplotlib.toolkits.exceltools
- Removed, dead/experimental ExampleInfo, Namespace and Importer code from matplotlib.__init__

46.9 Changes for 0.91.1

46.10 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 doesnt 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, eg. `$\mathcal{R}$`.
- 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.

46.11 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 (eg I added a default label
on this iteration) w/o having to change the tuple length and hence
the API of the client code everytime 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 int he vertical and horizontal direction.  As you'll see
in the changes below, these funcs 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
```

46.12 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
```

46.13 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 explicitely 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_figsize_inches 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
 constants.
```

46.14 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 MATPLOTLIB DATA 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.
```

46.15 Changes for 0.85

```
Made xtick and ytick separate props in rc

made pos=None the default for tick formatters 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.
```

46.16 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.
```

46.17 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.
  Eg, the new names are matplotlibrc, tex.cache, and ttffont.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.
```

46.18 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. Thats 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

      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.numerix.mlab import randn, rand, zeros, Float
 from matplotlib.mlab 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)
```

46.19 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].
```

46.20 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
```

46.21 Changes for 0.73

```
- Removed deprecated ColormapJet and friends

- Removed all error handling from the verbose object

- figure num of zero is now allowed
```

46.22 Changes for 0.72

```
- Line2D, Text, and Patch copy_properties renamed update_from and
  moved into artist base class

- LineCollecitons.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
```

46.23 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

  http://sourceforge.net/mailarchive/forum.php?thread_id=6398890&forum_id=36187
  http://sourceforge.net/mailarchive/forum.php?thread_id=6323208&forum_id=36187


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, eg (amin, amax, asum)
```

46.24 Changes for 0.70

```
MplEvent factored into a base class Event and derived classes
MouseEvent and KeyEvent

Removed definct set_measurement in wx toolbar
```

46.25 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 axe.  If you
have code you are porting which called, eg, figmanager.add_axes, you
can now simply do figmanager.canvas.figure.add_axes.
```

46.26 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 (obligue, etc) or the rotation of the
text.  Use style and rotation instead.
```

46.27 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.  Eg

  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
```

46.28 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, evet)
```

46.29 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
```

46.30 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.
```

46.31 Changes for 0.54

46.31.1 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 propertes that are available on rectangles or other patches are also available on PolyCollections, eg 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, 'linewidth', 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, eg 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'$\rm{Histogram\ of\ IQ:}\ \mu=100,\ \sigma=15$')
```

46.31.2 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 = clickBBox = lbwh_to_bbox(left, bottom, width, height)

The Bbox has a different API than the Bound2D. Eg, 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 gemoetry 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

l = Line2D(dpi, bbox, x, y, color = color, transx = transx, transy = transy,
)

now all you need is

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 insstances. 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 implemetations, 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, ie the x and y view limits), the bounding box of the display, and possibily nonlinear transformations of x and y. The 2 most frequently used transformations, data cordinates -> 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:

```
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.

46.32 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

    fig.draw()

  In the new API, you do

    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.  Eg, 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

    # 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 in now intialized 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

   figure.draw()  and
   figure.print_figure(args)

  should now be

   canvas.draw()  and
   canvas.print_figure(args)

  Apologies for the inconvenience.  This refactorization brings
```

```
significant more freedom in developing matplotlib and should bring
better plotting capabilities, so I hope the inconvenience is worth
it.
```

46.33 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 frmo 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, eg 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, eg figManager.window for GTK and
  figManager.frame for wx
```

46.34 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 intance 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

FORTYSEVEN

CONFIGURATION

47.1 matplotlib

This is an object-oriented plotting library.

A procedural interface is provided by the companion pyplot module, which may be imported directly, e.g:

```
from matplotlib.pyplot import *
```

To include numpy functions too, use:

```
from pylab import *
```

or using ipython:

```
ipython -pylab
```

For the most part, direct use of the object-oriented library is encouraged when programming; pyplot is primarily for working interactively. The exceptions are the pyplot commands `figure()`, `subplot()`, `subplots()`, `show()`, and `savefig()`, which can greatly simplify scripting.

Modules include:

matplotlib.axes defines the `Axes` class. Most pylab commands are wrappers for `Axes` methods. The axes module is the highest level of OO access to the library.

matplotlib.figure defines the `Figure` class.

matplotlib.artist defines the `Artist` base class for all classes that draw things.

matplotlib.lines defines the `Line2D` class for drawing lines and markers

matplotlib.patches defines classes for drawing polygons

matplotlib.text defines the `Text`, `TextWithDash`, and `Annotate` classes

matplotlib.image defines the `AxesImage` and `FigureImage` classes

matplotlib.collections classes for efficient drawing of groups of lines or polygons

matplotlib.colors classes for interpreting color specifications and for making colormaps

matplotlib.cm colormaps and the `ScalarMappable` mixin class for providing color mapping functionality to other classes

matplotlib.ticker classes for calculating tick mark locations and for formatting tick labels

matplotlib.backends a subpackage with modules for various gui libraries and output formats

The base matplotlib namespace includes:

rcParams a global dictionary of default configuration settings. It is initialized by code which may be overridded by a matplotlibrc file.

rc() a function for setting groups of rcParams values

use() a function for setting the matplotlib backend. If used, this function must be called immediately after importing matplotlib for the first time. In particular, it must be called **before** importing pylab (if pylab is imported).

matplotlib was initially written by John D. Hunter (jdh2358 at gmail.com) and is now developed and maintained by a host of others.

Occasionally the internal documentation (python docstrings) will refer to MATLAB®, a registered trademark of The MathWorks, Inc.

`matplotlib.rc`(*group*, ***kwargs*)

Set the current rc params. Group is the grouping for the rc, eg. 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, eg. (*xtick*, *ytick*). *kwargs* is a dictionary attribute name/value pairs, eg:

```
rc('lines', linewidth=2, color='r')
```

sets the current rc params and is equivalent to:

```
rcParams['lines.linewidth'] = 2
rcParams['lines.color'] = 'r'
```

The following aliases are available to save typing for interactive users:

Alias	Property
'lw'	'linewidth'
'ls'	'linestyle'
'c'	'color'
'fc'	'facecolor'
'ec'	'edgecolor'
'mew'	'markeredgewidth'
'aa'	'antialiased'

Thus you could abbreviate the above rc command as:

```
rc('lines', lw=2, c='r')
```

Note you can use python's kwargs dictionary facility to store dictionaries of default parameters. Eg, you can customize the font rc as follows:

```
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 `rcdefaults()` to restore the default rc params after changes.

`matplotlib.`**`rcdefaults`**`()`

Restore the default rc params. These are not the params loaded by the rc file, but mpl's internal params. See rc_file_defaults for reloading the default params from the rc file

`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()`.

CHAPTER

FORTYEIGHT

AFM (ADOBE FONT METRICS INTERFACE)

48.1 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 mine) I decided not to go with them because either they were either

1. copyrighted or used a non-BSD compatible license
2. had too many dependencies and I wanted a free standing lib
3. Did more than I needed and it was easier to write my own than figure out how to just get what I needed from theirs

It is pretty easy to use, and requires only built-in python libs:

```
>>> from afm import AFM
>>> fh = open('ptmr8a.afm')
>>> afm = AFM(fh)
>>> afm.string_width_height('What the heck?')
(6220.0, 683)
>>> 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]
>>> afm.get_bbox_font()
[-168, -218, 1000, 898]
```

AUTHOR: John D. Hunter <jdh2358@gmail.com>

class `matplotlib.afm.`**`AFM`**(*fh*)
Bases: `object`

Parse the AFM file in file object *fh*

`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, eg, 'Times'

`get_fontname`()
Return the font name, eg, 'Times-Roman'

`get_fullname`()
Return the font full name, eg, '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, ie, ';' 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, eg, '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 {})

CHAPTER

FORTYNINE

ANIMATION

49.1 matplotlib.animation

class matplotlib.animation.**Animation**(*fig*, *event_source=None*, *blit=False*)

Bases: `object`

This class wraps the creation of an animation using matplotlib. It is only a base class which should be subclassed to provide needed behavior.

fig is the figure object that is used to get draw, resize, and any other needed events.

event_source is 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 is a boolean that controls whether blitting is used to optimize drawing.

new_frame_seq()

Creates a new sequence of frame information.

new_saved_frame_seq()

Creates a new sequence of saved/cached frame information.

save(*filename*, *writer=None*, *fps=None*, *dpi=None*, *codec=None*, *bitrate=None*, *extra_args=None*, *metadata=None*, *extra_anim=None*)

Saves a movie file by drawing every frame.

filename is the output filename, eg `mymovie.mp4`

writer is either an instance of `MovieWriter` or a string key that identifies a class to use, such as 'ffmpeg' or 'mencoder'. If nothing is passed, the value of the rcparam `animation.writer` is used.

fps is the frames per second in the movie. Defaults to None, which will use the animation's specified interval to set the frames per second.

dpi controls the dots per inch for the movie frames. This combined with the figure's size in inches controls the size of the movie.

codec is the video codec to be used. Not all codecs are supported by a given `MovieWriter`. If none is given, this defaults to the value specified by the rcparam `animation.codec`.

bitrate specifies the amount 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 no value is given, this defaults to the value given by the rcparam `animation.bitrate`.

extra_args is a list of extra string arguments to be passed to the underlying movie utiltiy. The default is None, which passes the additional argurments in the 'animation.extra_args' rcParam.

metadata is 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.

extra_anim is a list of 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.

class `matplotlib.animation.`**`ArtistAnimation`**(*fig*, *artists*, **args*, ***kwargs*)

Bases: `matplotlib.animation.TimedAnimation`

Before calling this function, all plotting should have taken place and the relevant artists saved.

frame_info is a list, with each list entry a collection of artists that represent what needs to be enabled on each frame. These will be disabled for other frames.

class `matplotlib.animation.`**`FFMpegBase`**

`args_key` **= 'animation.ffmpeg_args'**

`exec_key` **= 'animation.ffmpeg_path'**

`output_args`

class `matplotlib.animation.`**`FFMpegFileWriter`**(**args*, ***kwargs*)

Bases: `matplotlib.animation.FileMovieWriter`, `matplotlib.animation.FFMpegBase`

`supported_formats` **= ['png', 'jpeg', 'ppm', 'tiff', 'sgi', 'bmp', 'pbm', 'raw', 'rgba']**

class `matplotlib.animation.`**`FFMpegWriter`**(*fps=5*, *codec=None*, *bitrate=None*, *extra_args=None*, *metadata=None*)

Bases: `matplotlib.animation.MovieWriter`, `matplotlib.animation.FFMpegBase`

Construct a new MovieWriter object.

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 utiltiy. The default is None, which passes the additional argurments 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.

class `matplotlib.animation.`**`FileMovieWriter`**(**args*, ***kwargs*)

Bases: `matplotlib.animation.MovieWriter`

`MovieWriter` subclass that handles writing to a file.

`cleanup`()

`finish`()

`frame_format`

Format (png, jpeg, etc.) to use for saving the frames, which can be decided by the individual subclasses.

`setup`(*fig*, *outfile*, *dpi*, *frame_prefix='_tmp'*, *clear_temp=True*)

Perform setup for writing the movie file.

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.

frame_prefix: string, optional The filename prefix to use for the temporary files. Defaults to '_tmp'

clear_temp: bool Specifies whether the temporary files should be deleted after the movie is written. (Useful for debugging.) Defaults to True.

class `matplotlib.animation.`**`FuncAnimation`**(*fig*, *func*, *frames=None*, *init_func=None*, *fargs=None*, *save_count=None*, ***kwargs*)

Bases: `matplotlib.animation.TimedAnimation`

Makes an animation by repeatedly calling a function *func*, passing in (optional) arguments in *fargs*.

frames can be a generator, an iterable, or a number of frames.

init_func is 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.

new_frame_seq()

new_saved_frame_seq()

class matplotlib.animation.**ImageMagickBase**

args_key = 'animation.convert_args'

delay

exec_key = 'animation.convert_path'

output_args

class matplotlib.animation.**ImageMagickFileWriter**(**args*, ***kwargs*)
Bases: matplotlib.animation.FileMovieWriter, matplotlib.animation.ImageMagickBase

supported_formats = ['png', 'jpeg', 'ppm', 'tiff', 'sgi', 'bmp', 'pbm', 'raw', 'rgba']

class matplotlib.animation.**ImageMagickWriter**(*fps=5*, *codec=None*, *bitrate=None*, *extra_args=None*, *metadata=None*)
Bases: matplotlib.animation.MovieWriter, matplotlib.animation.ImageMagickBase

Construct a new MovieWriter object.

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 utiltiy. The default is None, which passes the additional argurments 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.

class matplotlib.animation.**MencoderBase**

allowed_metadata = ['name', 'artist', 'genre', 'subject', 'copyright', 'srcform', 'comment']

args_key = 'animation.mencoder_args'

exec_key = 'animation.mencoder_path'

output_args

class matplotlib.animation.**MencoderFileWriter**(**args*, ***kwargs*)
Bases: matplotlib.animation.FileMovieWriter, matplotlib.animation.MencoderBase

supported_formats = ['png', 'jpeg', 'tga', 'sgi']

class matplotlib.animation.**MencoderWriter**(*fps=5*, *codec=None*, *bitrate=None*, *extra_args=None*, *metadata=None*)
Bases: matplotlib.animation.MovieWriter, matplotlib.animation.MencoderBase

Construct a new MovieWriter object.

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 utiltiy. The default is None, which passes the additional argurments 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.

class matplotlib.animation.**MovieWriter**(*fps=5*, *codec=None*, *bitrate=None*, *extra_args=None*, *metadata=None*)
Bases: `object`

Base class for writing movies. Fundamentally, what a MovieWriter does is provide is a way to grab frames by calling grab_frame(). setup() is called to start the process and finish() is called afterwards. This class is set up to provide for writing movie frame data to a pipe. saving() is provided as a context manager to facilitate this process as:

```
with moviewriter.saving('myfile.mp4'):
    # Iterate over frames
    moviewriter.grab_frame()
```

The use of the context manager ensures that setup and cleanup are performed as necessary.

frame_format: string The format used in writing frame data, defaults to 'rgba'

Construct a new MovieWriter object.

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 utiltiy. The default is None, which passes the additional argurments 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.

classmethod `bin_path`()
Returns the binary path to the commandline tool used by a specific subclass. This is a class method so that the tool can be looked for before making a particular MovieWriter subclass available.

`cleanup`()
Clean-up and collect the process used to write the movie file.

`finish`()
Finish any processing for writing the movie.

`frame_size`
A tuple (width,height) in pixels of a movie frame.

`grab_frame`()
Grab the image information from the figure and save as a movie frame.

classmethod `isAvailable`()
Check to see if a MovieWriter subclass is actually available by running the commandline tool.

`saving`(**args*, ***kwds*)
Context manager to facilitate writing the movie file.

`*args` are any parameters that should be passed to `setup`.

`setup`(*fig*, *outfile*, *dpi*, **args*)
Perform setup for writing the movie file.

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.

class `matplotlib.animation.`**`MovieWriterRegistry`**

Bases: `object`

`list`()

Get a list of available MovieWriters.

`register`(*name*)

class `matplotlib.animation.`**`TimedAnimation`**(*fig*, *interval=200*, *repeat_delay=None*, *repeat=True*, *event_source=None*, **args*, ***kwargs*)

Bases: `matplotlib.animation.Animation`

`Animation` subclass that supports time-based animation, drawing a new frame every *interval* milliseconds.

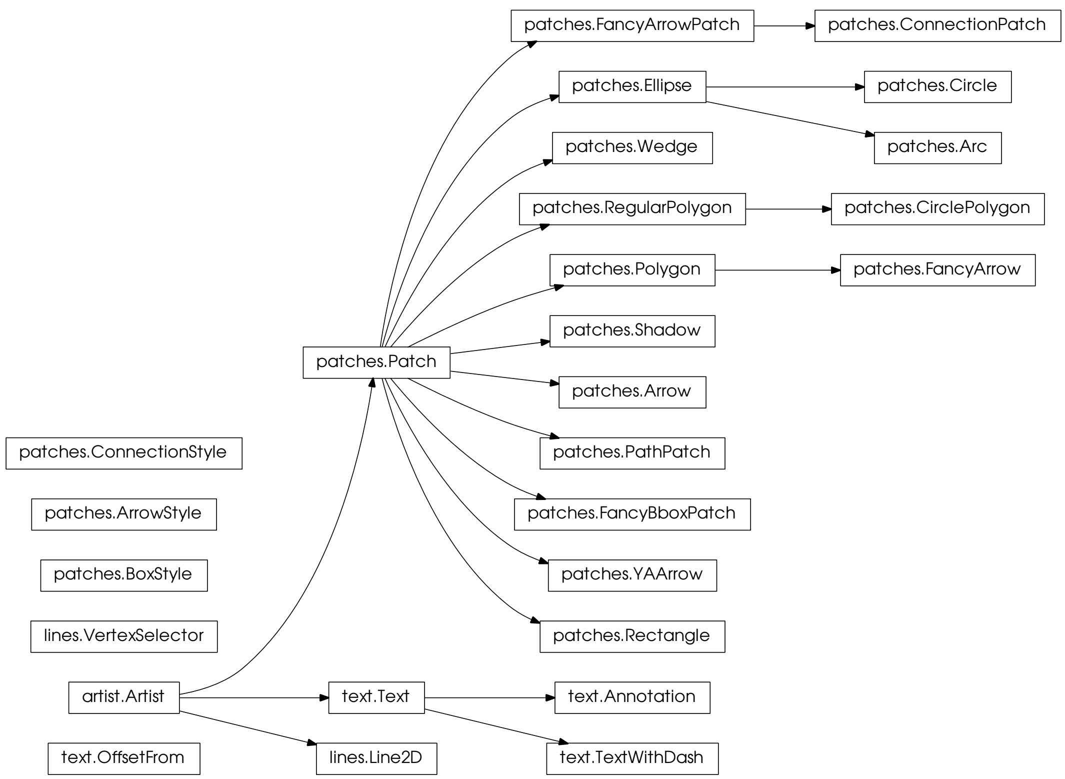
repeat controls whether the animation should repeat when the sequence of frames is completed.

repeat_delay optionally adds a delay in milliseconds before repeating the animation.

CHAPTER

FIFTY

ARTISTS



50.1 matplotlib.artist

class `matplotlib.artist.`**`Artist`**

Bases: `object`

Abstract base class for someone who renders into a `FigureCanvas`.

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.

aname = '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.

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(*renderer*, **args*, ***kwargs*)
Derived classes drawing method

findobj(*match=None*, *include_self=True*)
Find artist objects.

pyplot signature: findobj(o=gcf(), match=None, include_self=True)

Recursively find all :class:matplotlib.artist.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: eg Line2D. Only return artists of class type.

If *include_self* is True (default), include self in the list to be checked for a match.

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_axes()
Return the Axes instance the artist resides in, or *None*

get_children()
Return a list of the child `Artist's this :class:'Artist` contains.

get_clip_box()
Return artist clipbox

get_clip_on()
Return whether artist uses clipping

get_clip_path()
Return artist clip path

get_contains()
Return the _contains test used by the artist, or *None* for default.

get_figure()
Return the `Figure` instance the artist belongs to.

get_gid()
Returns the group id

get_label()
Get the label used for this artist in the legend.

get_picker()
Return the picker object used by this artist

get_rasterized()
return True if the artist is to be rasterized

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_url()
Returns the url

get_visible()
Return the artist's visiblity

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.

pchanged()
Fire an event when property changed, calling all of the registered callbacks.

pick(*mouseevent*)
call signature:

```
pick(mouseevent)
```

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 tkstyle set command, pass *kwargs* to set properties

set_agg_filter(*filter_func*)
set agg_filter fuction.

set_alpha(*alpha*)
Set the alpha value used for blending - not supported on all backends.

ACCEPTS: float (0.0 transparent through 1.0 opaque)

set_animated(*b*)
Set the artist's animation state.

ACCEPTS: [True | False]

set_axes(*axes*)
Set the `Axes` instance in which the artist resides, if any.

ACCEPTS: an `Axes` instance

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.

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_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_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_label(*s*)
Set the label to *s* for auto legend.

ACCEPTS: string or anything printable with '%s' conversion.

set_lod(*on*)
Set Level of Detail on or off. If on, the artists may examine things like the pixel width of the axes and draw a subset of their contents accordingly

ACCEPTS: [True | False]

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_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_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_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

update(*props*)
Update the properties of this `Artist` from the dictionary *prop*.

update_from(*other*)
Copy properties from *other* to *self*.

zorder = 0

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.

aliased_name(*s*)
return 'PROPNAME or alias' if *s* has an alias, else return PROPNAME.

E.g. for the line markerfacecolor property, which has an alias, return 'markerfacecolor or mfc' and for the transform property, which does not, return 'transform'

aliased_name_rest(*s*, *target*)
return 'PROPNAME or alias' if *s* has an alias, else return PROPNAME formatted for ReST

E.g. for the line markerfacecolor property, which has an alias, return 'markerfacecolor or mfc' and for the transform property, which does not, return 'transform'

findobj(*match=None*)
Recursively find all `matplotlib.artist.Artist` instances contained in *self*.

If *match* is not None, it can be

- function with signature `boolean = match(artist)`
- class instance: eg `Line2D`

used to filter matches.

get_aliases()
Get a dict mapping *fullname* -> *alias* for each *alias* in the `ArtistInspector`.

Eg., for lines:

```
{'markerfacecolor': 'mfc',
 'linewidth'      : 'lw',
}
```

get_setters()
Get the attribute strings with setters for object. Eg., for a line, return `['markerfacecolor', 'linewidth', ....]`.

get_valid_values(*attr*)
Get the legal arguments for the setter associated with *attr*.

This is done by querying the docstring of the function *set_attr* for a line that begins with ACCEPTS:

Eg., for a line linestyle, return "[`'-'` | `'--'` | `'-.'` | `':'` | `'steps'` | `'None'`]"

is_alias(*o*)
Return *True* if method object *o* is an alias for another function.

pprint_getters()
Return the getters and actual values as list of strings.

pprint_setters(*prop=None*, *leadingspace=2*)
If *prop* is *None*, return a list of strings of all settable properies and their valid values.

If *prop* is not *None*, it is a valid property name and that property will be returned as a string of property : valid values.

pprint_setters_rest(*prop=None*, *leadingspace=2*)
If *prop* is *None*, return a list of strings of all settable properies and their valid values. Format the output for ReST

If *prop* is not *None*, it is a valid property name and that property will be returned as a string of property : valid values.

properties()
return a dictionary mapping property name -> value

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.

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, eg 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

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:

```
getp(obj)  # get all the object properties
getp(obj, 'linestyle')  # get the linestyle property
```

obj is a Artist instance, eg 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

`matplotlib.artist.kwdoc(a)`

`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:

```
>>> 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:

```
>>> 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:

```
>>> 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:

```
>>> 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:

```
>>> setp(lines, 'linewidth', 2, 'color', r')  # MATLAB style

>>> setp(lines, linewidth=2, color='r')       # python style
```

50.2 matplotlib.lines

This module contains all the 2D line class which can draw with a variety of line styles, markers and colors.

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='full'*, *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, eg 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:

Property	Description
`agg_filter`	unknown
`alpha`	float (0.0 transparent through 1.0 opaque)
`animated`	[True | False]
`antialiased` or aa	[True | False]
`axes`	an `Axes` instance
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color` or c	any matplotlib color
`contains`	a callable function
`dash_capstyle`	['butt' | 'round' | 'projecting']
`dash_joinstyle`	['miter' | 'round' | 'bevel']
`dashes`	sequence of on/off ink in points
`data`	2D array (rows are x, y) or two 1D arrays
`drawstyle`	['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post']
`figure`	a `matplotlib.figure.Figure` instance
`fillstyle`	['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']
`gid`	an id string
`label`	string or anything printable with '%s' conversion.
`linestyle` or ls	[`'-'` | `'--'` | `'-.'` | `':'` | `'None'` | `' '` | ''] and any drawstyle in combina
`linewidth` or lw	float value in points
`lod`	[True | False]
`marker`	[`7` | `4` | `5` | `6` | `'o'` | `'D'` | `'h'` | `'H'` | `'_'` | '' | `'None'` | `None` | `' '` | `'8'` | `'p`
`markeredgecolor` or mec	any matplotlib color
`markeredgewidth` or mew	float value in points
`markerfacecolor` or mfc	any matplotlib color

Table 50.1

Property	Description
markerfacecoloralt or mfcalt	any matplotlib color
markersize or ms	float
markevery	None | integer | (startind, stride)
picker	float distance in points or callable pick function fn(artist, event)
pickradius	float distance in points
rasterized	[True | False | None]
snap	unknown
solid_capstyle	['butt' | 'round' | 'projecting']
solid_joinstyle	['miter' | 'round' | 'bevel']
transform	a matplotlib.transforms.Transform instance
url	a url string
visible	[True | False]
xdata	1D array
ydata	1D array
zorder	any number

See set_linestyle() for a decription of the line styles, set_marker() for a description of the markers, and set_drawstyle() for a description of the draw styles.

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*)

drawStyleKeys = ['default', 'steps-mid', 'steps-pre', 'steps-post', 'steps']

drawStyles = {'default': '_draw_lines', 'steps-mid': '_draw_steps_mid', 'steps': '_draw_steps_pre'

fillStyles = ('full', 'left', 'right', 'bottom', 'top', 'none')

filled_markers = ('o', 'v', '^', '<', '>', '8', 's', 'p', '*', 'h', 'H', 'D', 'd')

get_aa()
alias for get_antialiased

get_antialiased()

get_c()
alias for get_color

get_color()

get_dash_capstyle()
Get the cap style for dashed linestyles

get_dash_joinstyle()
Get the join style for dashed linestyles

get_data(*orig=True*)
Return the xdata, ydata.

If *orig* is *True*, return the original data

get_drawstyle()

get_fillstyle()
return the marker fillstyle

get_linestyle()

get_linewidth()

get_ls()
alias for get_linestyle

get_lw()
alias for get_linewidth

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(*orig=True*)
Return the ydata.

If *orig* is *True*, return the original data, else the processed data.

is_dashed()
return True if line is dashstyle

lineStyles = {'': '_draw_nothing', ' ': '_draw_nothing', 'None': '_draw_nothing', '–': '_draw_das

markers = {0: 'tickleft', 1: 'tickright', 2: 'tickup', 3: 'tickdown', 4: 'caretleft', 'D': 'diamond', 6: 'ca

recache(*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_axes(*ax*)
Set the Axes instance in which the artist resides, if any.

ACCEPTS: an Axes instance

`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 = (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(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(linestyle)`
Set the linestyle of the line (also accepts drawstyles)

linestyle	description
`'-'`	solid
`'--'`	dashed
`'-.'`	dash_dot
`':'`	dotted
`'None'`	draw nothing
`' '`	draw nothing
”	draw nothing

'steps' is equivalent to 'steps-pre' and is maintained for backward-compatibility.

See Also:

`set_drawstyle()` To set the drawing style (stepping) of the plot.

ACCEPTS: [`'-'` | `'--'` | `'-.'` | `':'` | `'None'` | `' '` | `''`] and any drawstyle in combination with a linestyle, e.g. `'steps--'`.

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

marker	description
`7`	caretdown
`4`	caretleft
`5`	caretright
`6`	caretup
`'o'`	circle
`'D'`	diamond
`'h'`	hexagon1
`'H'`	hexagon2
`'_'`	hline
`''`	nothing
`'None'`	nothing
`None`	nothing
`' '`	nothing
`'8'`	octagon
`'p'`	pentagon
`','`	pixel
`'+'`	plus
`'.'`	point
`'s'`	square
`'*'`	star
`'d'`	thin_diamond
`3`	tickdown

Continued on next page

Table 50.2 – continued from previous page

marker	description
`0`	tickleft
`1`	tickright
`2`	tickup
`'1'`	tri_down
`'3'`	tri_left
`'4'`	tri_right
`'2'`	tri_up
`'v'`	triangle_down
`'<'`	triangle_left
`'>'`	triangle_right
`'^'`	triangle_up
`'|'`	vline
`'x'`	x
`'$...$'`	render the string using mathtext.
verts	a list of (x, y) pairs used for Path vertices.
path	a `Path` instance.
(*numsides*, *style*, *angle*)	see below

The marker can also be a tuple (*numsides*, *style*, *angle*), which will create a custom, regular symbol.

***numsides*:** the number of sides

***style*:** the style of the regular symbol:

Value	Description
0	a regular polygon
1	a star-like symbol
2	an asterisk
3	a circle (*numsides* and *angle* is ignored)

***angle*:** the angle of rotation of the symbol, in degrees

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.

ACCEPTS: [`7` | `4` | `5` | `6` | `'o'` | `'D'` | `'h'` | `'H'` | `'_'` | ” | `'None'` | `None` | `' '` | `'8'` | `'p'` | `','` | `'+'` | `'.'` | `'s'` | `'*'` | `'d'` | `3` | `0` | `1` | `2` | `'1'` | `'3'` | `'4'` | `'2'` | `'v'` | `'<'` | `'>'` | `'^'` | `'|'` | `'x'` | `'$...$'` | *tuple* | *Nx2 array*]

`set_markeredgecolor`(*ec*)

Set the marker edge color

ACCEPTS: any matplotlib color

`set_markeredgewidth`(*ew*)
Set the marker edge width in points

ACCEPTS: float value in points

`set_markerfacecolor`(*fc*)
Set the marker face color.

ACCEPTS: any matplotlib color

`set_markerfacecoloralt`(*fc*)
Set the alternate marker face color.

ACCEPTS: any matplotlib color

`set_markersize`(*sz*)
Set the marker size in points

ACCEPTS: float

`set_markevery`(*every*)
Set the markevery property to subsample the plot when using markers. Eg if `markevery=5`, every 5-th marker will be plotted. *every* can be

None Every point will be plotted

an integer N Every N-th marker will be plotted starting with marker 0

A length-2 tuple of integers every=(start, N) will start at point start and plot every N-th marker

ACCEPTS: None | integer | (startind, stride)

`set_mec`(*val*)
alias for set_markeredgecolor

`set_mew`(*val*)
alias for set_markeredgewidth

`set_mfc`(*val*)
alias for set_markerfacecolor

`set_mfcalt`(*val*)
alias for set_markerfacecoloralt

`set_ms`(*val*)
alias for set_markersize

`set_picker`(*p*)
Sets the event picker details for the line.

ACCEPTS: float distance in points or callable pick function `fn(artist, event)`

`set_pickradius`(*d*)
Sets the pick radius used for containment tests

ACCEPTS: float distance in points

set_solid_capstyle(*s*)
Set the cap style for solid linestyles

ACCEPTS: ['butt' | 'round' | 'projecting']

set_solid_joinstyle(*s*)
Set the join style for solid linestyles ACCEPTS: ['miter' | 'round' | 'bevel']

set_transform(*t*)
set the Transformation instance used by this artist

ACCEPTS: a `matplotlib.transforms.Transform` instance

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*)
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:

```
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.

50.3 matplotlib.patches

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:

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
clip_box	a matplotlib.transforms.Bbox instance
clip_on	[True | False]
clip_path	[(Path, Transform) | Patch | None]
color	matplotlib color spec
contains	a callable function
edgecolor or ec	mpl color spec, or None for default, or 'none' for no color
facecolor or fc	mpl color spec, or None for default, or 'none' for no color
figure	a matplotlib.figure.Figure instance
fill	[True | False]
gid	an id string
hatch	['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
label	string or anything printable with '%s' conversion.
linestyle or ls	['solid' | 'dashed' | 'dashdot' | 'dotted']
linewidth or lw	float or None for default
lod	[True | False]
path_effects	unknown
picker	[None|float|boolean|callable]
rasterized	[True | False | None]
snap	unknown
transform	Transform instance
url	a url string
visible	[True | False]
zorder	any number

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.
>
> http://www.tinaja.com/glib/ellipse4.pdf

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 be 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:

   > Vince, John. Geometry for Computer Graphics: Formulae, Examples & Proofs. London: Springer-Verlag, 2005.

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()`.

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:

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
clip_box	a matplotlib.transforms.Bbox instance
clip_on	[True | False]
clip_path	[(Path, Transform) | Patch | None]
color	matplotlib color spec
contains	a callable function
edgecolor or ec	mpl color spec, or None for default, or 'none' for no color
facecolor or fc	mpl color spec, or None for default, or 'none' for no color
figure	a matplotlib.figure.Figure instance
fill	[True | False]
gid	an id string
hatch	['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
label	string or anything printable with '%s' conversion.
linestyle or ls	['solid' | 'dashed' | 'dashdot' | 'dotted']
linewidth or lw	float or None for default
lod	[True | False]
path_effects	unknown
picker	[None|float|boolean|callable]
rasterized	[True | False | None]
snap	unknown
transform	Transform instance
url	a url string
visible	[True | False]
zorder	any number

get_patch_transform()

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:

```
ArrowStyle.Fancy(head_length=.4, head_width=.4, tail_width=.4)
```

or:

```
ArrowStyle("Fancy", head_length=.4, head_width=.4, tail_width=.4)
```

or:

```
ArrowStyle("Fancy, head_length=.4, head_width=.4, tail_width=.4")
```

The following classes are defined

<table>
<tr><th>Class</th><th>Name</th><th>Attrs</th></tr>
<tr><td>Curve</td><td>-</td><td>None</td></tr>
<tr><td>CurveB</td><td>-></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>CurveFilledB</td><td>-|></td><td>head_length=0.4,head_width=0.2</td></tr>
<tr><td>CurveA</td><td><-</td><td>head_length=0.4,head_width=0.2</td></tr>
<tr><td>CurveAB</td><td><-></td><td>head_length=0.4,head_width=0.2</td></tr>
<tr><td>CurveFilledA</td><td><|-</td><td>head_length=0.4,head_width=0.2</td></tr>
<tr><td>CurveFilledAB</td><td><|-|></td><td>head_length=0.4,head_width=0.2</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=No</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>widthA=1.0,angleA=None,widthB=1.0,angleB=None</td></tr>
</table>

An instance of any arrow style class is an 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 witch the arrow will be drawn. *mutation_size* and *aspect_ratio* has a 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.

class BarAB(*widthA=1.0*, *angleA=None*, *widthB=1.0*, *angleB=None*)

Bases: `matplotlib.patches._Bracket`

An arrow with a bar(|) at both ends.

widthA width of the bracket

lengthA length of the bracket

angleA angle between the bracket and the line

widthB width of the bracket

lengthB length of the bracket

angleB angle between the bracket and the line

class `ArrowStyle.`**`BracketA`**(*widthA=1.0*, *lengthA=0.2*, *angleA=None*)
Bases: `matplotlib.patches._Bracket`

An arrow with a bracket(]) at its end.

widthA width of the bracket

lengthA length of the bracket

angleA angle between the bracket and the line

class `ArrowStyle.`**`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.

widthA width of the bracket

lengthA length of the bracket

angleA angle between the bracket and the line

widthB width of the bracket

lengthB length of the bracket

angleB angle between the bracket and the line

class `ArrowStyle.`**`BracketB`**(*widthB=1.0*, *lengthB=0.2*, *angleB=None*)
Bases: `matplotlib.patches._Bracket`

An arrow with a bracket([) at its end.

widthB width of the bracket

lengthB length of the bracket

angleB angle between the bracket and the line

class `ArrowStyle.`**`Curve`**
Bases: `matplotlib.patches._Curve`

A simple curve without any arrow head.

class `ArrowStyle.`**`CurveA`**(*head_length=0.4*, *head_width=0.2*)
Bases: `matplotlib.patches._Curve`

An arrow with a head at its begin point.

head_length length of the arrow head

head_width width of the arrow head

class `ArrowStyle.`**`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.

head_length length of the arrow head

head_width width of the arrow head

class `ArrowStyle.`**`CurveB`**(*head_length=0.4*, *head_width=0.2*)
Bases: `matplotlib.patches._Curve`

An arrow with a head at its end point.

head_length length of the arrow head

head_width width of the arrow head

class `ArrowStyle.`**`CurveFilledA`**(*head_length=0.4*, *head_width=0.2*)
Bases: `matplotlib.patches._Curve`

An arrow with filled triangle head at the begin.

head_length length of the arrow head

head_width width of the arrow head

class `ArrowStyle.`**`CurveFilledAB`**(*head_length=0.4*, *head_width=0.2*)
Bases: `matplotlib.patches._Curve`

An arrow with filled triangle heads both at the begin and the end point.

head_length length of the arrow head

head_width width of the arrow head

class `ArrowStyle.`**`CurveFilledB`**(*head_length=0.4*, *head_width=0.2*)
Bases: `matplotlib.patches._Curve`

An arrow with filled triangle head at the end.

head_length length of the arrow head

head_width width of the arrow head

class `ArrowStyle.`**`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.

head_length length of the arrow head

head_with width of the arrow head

tail_width width of the arrow tail

transmute(*path*, *mutation_size*, *linewidth*)

class ArrowStyle.**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.

head_length length of the arrow head

head_with width of the arrow head

tail_width width of the arrow tail

transmute(*path*, *mutation_size*, *linewidth*)

class ArrowStyle.**Wedge**(*tail_width=0.3*, *shrink_factor=0.5*)
Bases: `matplotlib.patches._Base`

Wedge(?) shape. Only wokrs 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.

tail_width width of the tail

shrink_factor 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.

Class	Name	Attrs
LArrow	`larrow`	pad=0.3
RArrow	`rarrow`	pad=0.3
Round	`round`	pad=0.3,rounding_size=None
Round4	`round4`	pad=0.3,rounding_size=None
Roundtooth	`roundtooth`	pad=0.3,tooth_size=None
Sawtooth	`sawtooth`	pad=0.3,tooth_size=None
Square	`square`	pad=0.3

An instance of any boxstyle class is an callable object, whose call signature is:

```
__call__(self, x0, y0, width, height, mutation_size, aspect_ratio=1.)
```

and returns a `Path` instance. *x0*, *y0*, *width* and *height* specify the location and size of the box to be drawn. *mutation_scale* determines the overall size of the mutation (by which I mean the transformation of the rectangle to the fancy box). *mutation_aspect* determines the aspect-ratio of the mutation.

class `LArrow`(*pad=0.3*)
Bases: `matplotlib.patches._Base`

(left) Arrow Box

`transmute`(*x0*, *y0*, *width*, *height*, *mutation_size*)

class `BoxStyle.`**`RArrow`**(*pad=0.3*)
Bases: `matplotlib.patches.LArrow`

(right) Arrow Box

`transmute`(*x0*, *y0*, *width*, *height*, *mutation_size*)

class `BoxStyle.`**`Round`**(*pad=0.3*, *rounding_size=None*)
Bases: `matplotlib.patches._Base`

A box with round corners.

pad amount of padding

rounding_size rounding radius of corners. *pad* if None

`transmute`(*x0*, *y0*, *width*, *height*, *mutation_size*)

class `BoxStyle.`**`Round4`**(*pad=0.3*, *rounding_size=None*)
Bases: `matplotlib.patches._Base`

Another box with round edges.

pad amount of padding

rounding_size rounding size of edges. *pad* if None

`transmute`(*x0*, *y0*, *width*, *height*, *mutation_size*)

class `BoxStyle.`**`Roundtooth`**(*pad=0.3*, *tooth_size=None*)

Bases: `matplotlib.patches.Sawtooth`

A roundtooth(?) box.

pad amount of padding

tooth_size size of the sawtooth. pad* if None

`transmute`(*x0*, *y0*, *width*, *height*, *mutation_size*)

class `BoxStyle.`**`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 `BoxStyle.`**`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
clip_box	a matplotlib.transforms.Bbox instance
clip_on	[True | False]
clip_path	[(Path, Transform) | Patch | None]
color	matplotlib color spec
contains	a callable function
edgecolor or ec	mpl color spec, or None for default, or 'none' for no color
facecolor or fc	mpl color spec, or None for default, or 'none' for no color
figure	a matplotlib.figure.Figure instance
fill	[True | False]
gid	an id string
hatch	['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
label	string or anything printable with '%s' conversion.
linestyle or ls	['solid' | 'dashed' | 'dashdot' | 'dotted']
linewidth or lw	float or None for default
lod	[True | False]
path_effects	unknown
picker	[None|float|boolean|callable]
rasterized	[True | False | None]
snap	unknown
transform	Transform instance
url	a url string
visible	[True | False]
zorder	any number

get_radius()

return the radius of the circle

radius

return the radius of the circle

set_radius(*radius*)

Set the radius of the circle

ACCEPTS: float

class matplotlib.patches.**CirclePolygon**(*xy*, *radius=5*, *resolution=20*, ***kwargs*)

Bases: matplotlib.patches.RegularPolygon

A polygon-approximation of a circle patch.

Create a circle at *xy* = (*x*, *y*) with given *radius*. This circle is approximated by a regular polygon with *resolution* sides. For a smoother circle drawn with splines, see `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
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color`	matplotlib color spec
`contains`	a callable function
`edgecolor` or ec	mpl color spec, or None for default, or 'none' for no color
`facecolor` or fc	mpl color spec, or None for default, or 'none' for no color
`figure`	a `matplotlib.figure.Figure` instance
`fill`	[True | False]
`gid`	an id string
`hatch`	['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
`label`	string or anything printable with '%s' conversion.
`linestyle` or ls	['solid' | 'dashed' | 'dashdot' | 'dotted']
`linewidth` or lw	float or None for default
`lod`	[True | False]
`path_effects`	unknown
`picker`	[None|float|boolean|callable]
`rasterized`	[True | False | None]
`snap`	unknown
`transform`	`Transform` instance
`url`	a url string
`visible`	[True | False]
`zorder`	any number

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	Description
arrowstyle	the arrow style
connectionstyle	the connection style
relpos	default is (0.5, 0.5)
patchA	default is bounding box of the text
patchB	default is None
shrinkA	default is 2 points
shrinkB	default is 2 points
mutation_scale	default is text size (in points)
mutation_aspect	default is 1.
?	any key for `matplotlib.patches.PathPatch`

coordsA and *coordsB* are strings that indicate the coordinates of *xyA* and *xyB*.

Property	Description
'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'	0,1 is lower left of axes and 1,1 is upper right
'data'	use the coordinate system of the object being annotated (default)
'offset points'	Specify an offset (in points) from the *xy* value
'polar'	you can specify *theta*, *r* 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.

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

Class	Name	Attrs
Angle	angle	angleA=90,angleB=0,rad=0.0
Angle3	angle3	angleA=90,angleB=0
Arc	arc	angleA=0,angleB=0,armA=None,armB=None,rad=0.0
Arc3	arc3	rad=0.0
Bar	bar	armA=0.0,armB=0.0,fraction=0.3,angle=None

An instance of any connection style class is an callable object, whose call signature is:

```
__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.

class `Angle`(*angleA=90*, *angleB=0*, *rad=0.0*)
Bases: `matplotlib.patches._Base`

Creates a picewise 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 `ConnectionStyle.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 `ConnectionStyle.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 `ConnectionStyle.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 contol 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 `ConnectionStyle.`**`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 arm is extend so that they are connected in a right angle. The length of armA is determined by (*armA* + *fraction* x AB distance). Same for armB.

armA : minimum length of armA *armB* : minimum length of armB *fraction* : a fraction of the distance between two points that

will be added to armA and armB.

angle [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:

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
clip_box	a matplotlib.transforms.Bbox instance
clip_on	[True | False]
clip_path	[(Path, Transform) | Patch | None]
color	matplotlib color spec
contains	a callable function
edgecolor or ec	mpl color spec, or None for default, or 'none' for no color
facecolor or fc	mpl color spec, or None for default, or 'none' for no color
figure	a matplotlib.figure.Figure instance
fill	[True | False]
gid	an id string
hatch	['/' | '\\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
label	string or anything printable with '%s' conversion.
linestyle or ls	['solid' | 'dashed' | 'dashdot' | 'dotted']
linewidth or lw	float or None for default
lod	[True | False]
path_effects	unknown
picker	[None|float|boolean|callable]
rasterized	[True | False | None]
snap	unknown
transform	Transform instance
url	a url string
visible	[True | False]
zorder	any number

contains(*ev*)

get_patch_transform()

get_path()

Return the vertices of the rectangle

class matplotlib.patches.**FancyArrow**(*x*, *y*, *dx*, *dy*, *width=0.001*, *length_includes_head=False*, *head_width=None*, *head_length=None*, *shape='full'*, *overhang=0*, *head_starts_at_zero=False*, ***kwargs*)

Bases: 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:

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
clip_box	a matplotlib.transforms.Bbox instance
clip_on	[True | False]
clip_path	[(Path, Transform) | Patch | None]
color	matplotlib color spec
contains	a callable function
edgecolor or ec	mpl color spec, or None for default, or 'none' for no color
facecolor or fc	mpl color spec, or None for default, or 'none' for no color
figure	a matplotlib.figure.Figure instance
fill	[True | False]
gid	an id string
hatch	['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
label	string or anything printable with '%s' conversion.
linestyle or ls	['solid' | 'dashed' | 'dashdot' | 'dotted']
linewidth or lw	float or None for default
lod	[True | False]
path_effects	unknown
picker	[None|float|boolean|callable]
rasterized	[True | False | None]
snap	unknown
transform	Transform instance
url	a url string
visible	[True | False]
zorder	any number

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 shirnked 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 *connectionstyle* describes how *posA* and *posB* are connected. It can be an instance of the ConnectionStyle class (matplotlib.patches.ConnectionStlye) or a string of the connectionstyle name, with optional comma-separated attributes. The following connection styles are available.

Class	Name	Attrs
Angle	`angle`	angleA=90,angleB=0,rad=0.0
Angle3	`angle3`	angleA=90,angleB=0
Arc	`arc`	angleA=0,angleB=0,armA=None,armB=None,rad=0.0
Arc3	`arc3`	rad=0.0
Bar	`bar`	armA=0.0,armB=0.0,fraction=0.3,angle=None

The *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 *mutation_scale*. The following arrow styles are available.

<table>
<tr><th>Class</th><th>Name</th><th>Attrs</th></tr>
<tr><td>Curve</td><td><code>-</code></td><td>None</td></tr>
<tr><td>CurveB</td><td><code>-></code></td><td>head_length=0.4,head_width=0.2</td></tr>
<tr><td>BracketB</td><td><code>-[</code></td><td>widthB=1.0,lengthB=0.2,angleB=None</td></tr>
<tr><td>CurveFilledB</td><td><code>-|></code></td><td>head_length=0.4,head_width=0.2</td></tr>
<tr><td>CurveA</td><td><code><-</code></td><td>head_length=0.4,head_width=0.2</td></tr>
<tr><td>CurveAB</td><td><code><-></code></td><td>head_length=0.4,head_width=0.2</td></tr>
<tr><td>CurveFilledA</td><td><code><|-</code></td><td>head_length=0.4,head_width=0.2</td></tr>
<tr><td>CurveFilledAB</td><td><code><|-|></code></td><td>head_length=0.4,head_width=0.2</td></tr>
<tr><td>BracketA</td><td><code>]-</code></td><td>widthA=1.0,lengthA=0.2,angleA=None</td></tr>
<tr><td>BracketAB</td><td><code>]-[</code></td><td>widthA=1.0,lengthA=0.2,angleA=None,widthB=1.0,lengthB=0.2,angleB=No</td></tr>
<tr><td>Fancy</td><td><code>fancy</code></td><td>head_length=0.4,head_width=0.4,tail_width=0.4</td></tr>
<tr><td>Simple</td><td><code>simple</code></td><td>head_length=0.5,head_width=0.5,tail_width=0.2</td></tr>
<tr><td>Wedge</td><td><code>wedge</code></td><td>tail_width=0.3,shrink_factor=0.5</td></tr>
<tr><td>BarAB</td><td><code>|-|</code></td><td>widthA=1.0,angleA=None,widthB=1.0,angleB=None</td></tr>
</table>

mutation_scale [a value with which attributes of arrowstyle] (e.g., head_length) 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:

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
clip_box	a matplotlib.transforms.Bbox instance
clip_on	[True | False]
clip_path	[(Path, Transform) | Patch | None]
color	matplotlib color spec
contains	a callable function
edgecolor or ec	mpl color spec, or None for default, or 'none' for no color
facecolor or fc	mpl color spec, or None for default, or 'none' for no color
figure	a matplotlib.figure.Figure instance
fill	[True | False]
gid	an id string
hatch	['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
label	string or anything printable with '%s' conversion.
linestyle or ls	['solid' | 'dashed' | 'dashdot' | 'dotted']
linewidth or lw	float or None for default
lod	[True | False]
path_effects	unknown
picker	[None|float|boolean|callable]
rasterized	[True | False | None]
snap	unknown
transform	Transform instance
url	a url string
visible	[True | False]
zorder	any number

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 shink factor. Mutation scale is not 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 probided 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 shink factor. Mutation scale is not 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 end end positions of the connecting path. Use current vlaue 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.

Class	Name	Attrs
LArrow	`larrow`	pad=0.3
RArrow	`rarrow`	pad=0.3
Round	`round`	pad=0.3,rounding_size=None
Round4	`round4`	pad=0.3,rounding_size=None
Roundtooth	`roundtooth`	pad=0.3,tooth_size=None
Sawtooth	`sawtooth`	pad=0.3,tooth_size=None
Square	`square`	pad=0.3

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:

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
clip_box	a matplotlib.transforms.Bbox instance
clip_on	[True | False]
clip_path	[(Path, Transform) | Patch | None]
color	matplotlib color spec
contains	a callable function
edgecolor or ec	mpl color spec, or None for default, or 'none' for no color
facecolor or fc	mpl color spec, or None for default, or 'none' for no color
figure	a matplotlib.figure.Figure instance
fill	[True | False]
gid	an id string
hatch	['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
label	string or anything printable with '%s' conversion.
linestyle or ls	['solid' | 'dashed' | 'dashdot' | 'dotted']
linewidth or lw	float or None for default
lod	[True | False]
path_effects	unknown
picker	[None|float|boolean|callable]
rasterized	[True | False | None]
snap	unknown
transform	Transform instance
url	a url string
visible	[True | False]
zorder	any number

get_bbox()

get_boxstyle()
Return the boxstyle object

get_height()
Return the height of the rectangle

get_mutation_aspect()
Return the aspect ratio of the bbox mutation.

get_mutation_scale()
Return the mutation scale.

get_path()
Return the mutated path of the rectangle

get_width()
Return the width of the rectangle

get_x()
Return the left coord 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_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:

```
set_boxstyle("round,pad=0.2")
set_boxstyle("round", pad=0.2)
```

Old attrs simply are forgotten.

Without argument (or with *boxstyle* = None), it returns available box styles.

ACCEPTS:

Class	Name	Attrs
LArrow	`larrow`	pad=0.3
RArrow	`rarrow`	pad=0.3
Round	`round`	pad=0.3,rounding_size=None
Round4	`round4`	pad=0.3,rounding_size=None
Roundtooth	`roundtooth`	pad=0.3,tooth_size=None
Sawtooth	`sawtooth`	pad=0.3,tooth_size=None
Square	`square`	pad=0.3

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=None*, *facecolor=None*, *color=None*, *linewidth=None*, *linestyle=None*, *antialiased=None*, *hatch=None*, *fill=True*, *path_effects=None*, ***kwargs*)
Bases: `matplotlib.artist.Artist`

A patch is a 2D artist with a face color and an edge color.

If any of *edgecolor*, *facecolor*, *linewidth*, or *antialiased* are *None*, they default to their rc params setting.

The following kwarg properties are supported

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
clip_box	a matplotlib.transforms.Bbox instance
clip_on	[True | False]
clip_path	[(Path, Transform) | Patch | None]
color	matplotlib color spec
contains	a callable function
edgecolor or ec	mpl color spec, or None for default, or 'none' for no color
facecolor or fc	mpl color spec, or None for default, or 'none' for no color
figure	a matplotlib.figure.Figure instance
fill	[True | False]
gid	an id string
hatch	['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
label	string or anything printable with '%s' conversion.
linestyle or ls	['solid' | 'dashed' | 'dashdot' | 'dotted']
linewidth or lw	float or None for default
lod	[True | False]
path_effects	unknown
picker	[None|float|boolean|callable]
rasterized	[True | False | None]
snap	unknown
transform	Transform instance
url	a url string
visible	[True | False]
zorder	any number

contains(*mouseevent*, *radius=None*)

Test whether the mouse event occurred in the patch.

Returns T/F, {}

contains_point(*point*, *radius=None*)

Returns *True* if the given point is inside the path (transformed with its transform attribute).

draw(*artist*, *renderer*, **args*, ***kwargs*)

Draw the Patch to the given *renderer*.

fill

return whether fill is set

get_aa()

Returns True if the Patch is to be drawn with antialiasing.

get_antialiased()
Returns True if the Patch is to be drawn with antialiasing.

get_data_transform()
Return the Transform instance which maps data coordinates to physical coordinates.

get_ec()
Return the edge color of the Patch.

get_edgecolor()
Return the edge color of the Patch.

get_extents()
Return a Bbox object defining the axis-aligned extents of the Patch.

get_facecolor()
Return the face color of the Patch.

get_fc()
Return the face color of the Patch.

get_fill()
return whether fill is set

get_hatch()
Return the current hatching pattern

get_linestyle()
Return the linestyle. Will be one of ['solid' | 'dashed' | 'dashdot' | 'dotted']

get_linewidth()
Return the line width in points.

get_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_path_effects()

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 tranparency of the patch.

ACCEPTS: float or None

set_antialiased(*aa*)
Set whether to use antialiased rendering

ACCEPTS: [True | False] or None for default

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

ACCEPTS: mpl color spec, or None for default, or 'none' for no color

set_facecolor(*color*)
Set the patch face color

ACCEPTS: 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

ACCEPTS: [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.

ACCEPTS: ['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']

set_linestyle(*ls*)
Set the patch linestyle

ACCEPTS: ['solid' | 'dashed' | 'dashdot' | 'dotted']

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

set_path_effects(*path_effects*)
set path_effects, which should be a list of instances of matplotlib.patheffect._Base class or its derivatives.

update_from(*other*)
Updates this `Patch` from the properties of *other*.

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:

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
clip_box	a matplotlib.transforms.Bbox instance
clip_on	[True | False]
clip_path	[(Path, Transform) | Patch | None]
color	matplotlib color spec
contains	a callable function
edgecolor or ec	mpl color spec, or None for default, or 'none' for no color
facecolor or fc	mpl color spec, or None for default, or 'none' for no color
figure	a matplotlib.figure.Figure instance
fill	[True | False]
gid	an id string
hatch	['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
label	string or anything printable with '%s' conversion.
linestyle or ls	['solid' | 'dashed' | 'dashdot' | 'dotted']
linewidth or lw	float or None for default
lod	[True | False]
path_effects	unknown
picker	[None|float|boolean|callable]
rasterized	[True | False | None]
snap	unknown
transform	Transform instance
url	a url string
visible	[True | False]
zorder	any number

See Also:

Patch For additional kwargs

get_path()

class matplotlib.patches.**Polygon**(*xy*, *closed=True*, ***kwargs*)

Bases: matplotlib.patches.Patch

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 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
clip_box	a matplotlib.transforms.Bbox instance
clip_on	[True | False]
clip_path	[(Path, Transform) | Patch | None]
color	matplotlib color spec
contains	a callable function
edgecolor or ec	mpl color spec, or None for default, or 'none' for no color
facecolor or fc	mpl color spec, or None for default, or 'none' for no color
figure	a matplotlib.figure.Figure instance
fill	[True | False]
gid	an id string
hatch	['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
label	string or anything printable with '%s' conversion.
linestyle or ls	['solid' | 'dashed' | 'dashdot' | 'dotted']
linewidth or lw	float or None for default
lod	[True | False]
path_effects	unknown
picker	[None|float|boolean|callable]
rasterized	[True | False | None]
snap	unknown
transform	Transform instance
url	a url string
visible	[True | False]
zorder	any number

See Also:

Patch For additional kwargs

get_closed()

get_path()

get_xy()

set_closed(*closed*)

set_xy(*xy*)

xy

Set/get the vertices of the polygon. This property is provided for backward compatibility with matplotlib 0.91.x only. New code should use get_xy() and set_xy()

instead.

class `matplotlib.patches.Rectangle`(*xy*, *width*, *height*, ***kwargs*)

Bases: `matplotlib.patches.Patch`

Draw a rectangle with lower left at *xy* = (*x*, *y*) with specified *width* and *height*.

fill is a boolean indicating whether to fill the rectangle

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
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color`	matplotlib color spec
`contains`	a callable function
`edgecolor` or ec	mpl color spec, or None for default, or 'none' for no color
`facecolor` or fc	mpl color spec, or None for default, or 'none' for no color
`figure`	a `matplotlib.figure.Figure` instance
`fill`	[True | False]
`gid`	an id string
`hatch`	['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
`label`	string or anything printable with '%s' conversion.
`linestyle` or ls	['solid' | 'dashed' | 'dashdot' | 'dotted']
`linewidth` or lw	float or None for default
`lod`	[True | False]
`path_effects`	unknown
`picker`	[None|float|boolean|callable]
`rasterized`	[True | False | None]
`snap`	unknown
`transform`	`Transform` instance
`url`	a url string
`visible`	[True | False]
`zorder`	any number

`contains`(*mouseevent*)

`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:

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
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color`	matplotlib color spec
`contains`	a callable function
`edgecolor` or ec	mpl color spec, or None for default, or 'none' for no color
`facecolor` or fc	mpl color spec, or None for default, or 'none' for no color
`figure`	a `matplotlib.figure.Figure` instance
`fill`	[True | False]
`gid`	an id string
`hatch`	['/' | '\\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
`label`	string or anything printable with '%s' conversion.
`linestyle` or ls	['solid' | 'dashed' | 'dashdot' | 'dotted']
`linewidth` or lw	float or None for default
`lod`	[True | False]
`path_effects`	unknown
`picker`	[None|float|boolean|callable]
`rasterized`	[True | False | None]
`snap`	unknown
`transform`	`Transform` instance
`url`	a url string
`visible`	[True | False]
`zorder`	any number

get_patch_transform()

get_path()

numvertices

orientation

radius

xy

class `matplotlib.patches.`**`Shadow`**(*patch*, *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 have the same color as the face, but darkened.

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
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color`	matplotlib color spec
`contains`	a callable function
`edgecolor` or ec	mpl color spec, or None for default, or 'none' for no color
`facecolor` or fc	mpl color spec, or None for default, or 'none' for no color
`figure`	a `matplotlib.figure.Figure` instance
`fill`	[True | False]
`gid`	an id string
`hatch`	['/' | '\\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
`label`	string or anything printable with '%s' conversion.
`linestyle` or ls	['solid' | 'dashed' | 'dashdot' | 'dotted']
`linewidth` or lw	float or None for default
`lod`	[True | False]
`path_effects`	unknown
`picker`	[None|float|boolean|callable]
`rasterized`	[True | False | None]
`snap`	unknown
`transform`	`Transform` instance
`url`	a url string
`visible`	[True | False]
`zorder`	any number

`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:

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
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color`	matplotlib color spec
`contains`	a callable function
`edgecolor` or ec	mpl color spec, or None for default, or 'none' for no color
`facecolor` or fc	mpl color spec, or None for default, or 'none' for no color
`figure`	a `matplotlib.figure.Figure` instance
`fill`	[True | False]
`gid`	an id string
`hatch`	['/' | '\\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
`label`	string or anything printable with '%s' conversion.
`linestyle` or ls	['solid' | 'dashed' | 'dashdot' | 'dotted']
`linewidth` or lw	float or None for default
`lod`	[True | False]
`path_effects`	unknown
`picker`	[None|float|boolean|callable]
`rasterized`	[True | False | None]
`snap`	unknown
`transform`	`Transform` instance
`url`	a url string
`visible`	[True | False]
`zorder`	any number

get_path()

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:

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
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color`	matplotlib color spec
`contains`	a callable function
`edgecolor` or ec	mpl color spec, or None for default, or 'none' for no color
`facecolor` or fc	mpl color spec, or None for default, or 'none' for no color
`figure`	a `matplotlib.figure.Figure` instance
`fill`	[True | False]
`gid`	an id string
`hatch`	['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
`label`	string or anything printable with '%s' conversion.
`linestyle` or ls	['solid' | 'dashed' | 'dashdot' | 'dotted']
`linewidth` or lw	float or None for default
`lod`	[True | False]
`path_effects`	unknown
`picker`	[None|float|boolean|callable]
`rasterized`	[True | False | None]
`snap`	unknown
`transform`	`Transform` instance
`url`	a url string
`visible`	[True | False]
`zorder`	any number

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.

50.4 matplotlib.text

Classes for including text in a figure.

class matplotlib.text.**Annotation**(*s*, *xy*, *xytext=None*, *xycoords='data'*, *textcoords=None*, *arrowprops=None*, *annotation_clip=None*, ***kwargs*)
Bases: `matplotlib.text.Text`, `matplotlib.text._AnnotationBase`

A `Text` class to make annotating things in the figure, such as `Figure`, `Axes`, `Rectangle`, etc., easier.

Annotate the *x*, *y* point *xy* with text *s* at *x*, *y* location *xytext*. (If *xytext* = *None*, defaults to *xy*, and if *textcoords* = *None*, defaults to *xycoords*).

arrowprops, if not *None*, is a dictionary of line properties (see `matplotlib.lines.Line2D`) for the arrow that connects annotation to the point.

If the dictionary has a key *arrowstyle*, a FancyArrowPatch instance is created with the given dictionary and is drawn. Otherwise, a YAArow patch instance is created and drawn. Valid keys for YAArow are

Key	Description
width	the width of the arrow in points
frac	the fraction of the arrow length occupied by the head
head-width	the width of the base of the arrow head in points
shrink	oftentimes it is convenient to have the arrowtip and base a bit away from the text and point being annotated. If *d* is the distance between the text and annotated point, shrink will shorten the arrow so the tip and base are shink percent of the distance *d* away from the endpoints. ie, `shrink=0.05 is 5%`
?	any key for `matplotlib.patches.polygon`

Valid keys for FancyArrowPatch are

Key	Description
arrowstyle	the arrow style
connectionstyle	the connection style
relpos	default is (0.5, 0.5)
patchA	default is bounding box of the text
patchB	default is None
shrinkA	default is 2 points
shrinkB	default is 2 points
mutation_scale	default is text size (in points)
mutation_aspect	default is 1.
?	any key for `matplotlib.patches.PathPatch`

xycoords and *textcoords* are strings that indicate the coordinates of *xy* and *xytext*.

Property	Description
'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'	0,1 is lower left of axes and 1,1 is upper right
'data'	use the coordinate system of the object being annotated (default)
'offset points'	Specify an offset (in points) from the *xy* value
'polar'	you can specify *theta*, *r* 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.

If a 'points' or 'pixels' option is specified, values will be added to the bottom-left and if negative, values will be subtracted from the top-right. Eg:

```
# 10 points to the right of the left border of the axes and
# 5 points below the top border
xy=(10,-5), xycoords='axes points'
```

You may use an instance of `Transform` or `Artist`. See *Annotating Axes* for more details.

The *annotation_clip* attribute contols 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".

Additional kwargs are Text properties:

Property	Description
`agg_filter`	unknown
`alpha`	float (0.0 transparent through 1.0 opaque)
`animated`	[True | False]
`axes`	an `Axes` instance
`backgroundcolor`	any matplotlib color
`bbox`	rectangle prop dict

Table 50.3 – cont

Property	Description
clip_box	a matplotlib.transforms.Bbox instance
clip_on	[True | False]
clip_path	[(Path, Transform) | Patch | None]
color	any matplotlib color
contains	a callable function
family or fontfamily or fontname or name	[FONTNAME | 'serif' | 'sans-serif' | 'cursive' | 'fantasy' | 'mono
figure	a matplotlib.figure.Figure instance
fontproperties or font_properties	a matplotlib.font_manager.FontProperties instance
gid	an id string
horizontalalignment or ha	['center' | 'right' | 'left']
label	string or anything printable with '%s' conversion.
linespacing	float (multiple of font size)
lod	[True | False]
multialignment	['left' | 'right' | 'center']
path_effects	unknown
picker	[None|float|boolean|callable]
position	(x,y)
rasterized	[True | False | None]
rotation	[angle in degrees | 'vertical' | 'horizontal']
rotation_mode	unknown
size or fontsize	[size in points | 'xx-small' | 'x-small' | 'small' | 'medium' | 'large
snap	unknown
stretch or fontstretch	[a numeric value in range 0-1000 | 'ultra-condensed' | 'extra-conc
style or fontstyle	['normal' | 'italic' | 'oblique']
text	string or anything printable with '%s' conversion.
transform	Transform instance
url	a url string
variant or fontvariant	['normal' | 'small-caps']
verticalalignment or va or ma	['center' | 'top' | 'bottom' | 'baseline']
visible	[True | False]
weight or fontweight	[a numeric value in range 0-1000 | 'ultralight' | 'light' | 'normal' |
x	float
y	float
zorder	any number

contains(*event*)

draw(*artist*, *renderer*, **args*, ***kwargs*)
Draw the Annotation object to the given *renderer*.

set_figure(*fig*)

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_positions(*renderer*)
Update the pixel positions of the annotated point and the text.

class `matplotlib.text.`**OffsetFrom**(*artist*, *ref_coord*, *unit='points'*)
Bases: `object`

get_unit()

set_unit(*unit*)

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*, *path_effects=None*, ***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

Property	Description
`agg_filter`	unknown
`alpha`	float (0.0 transparent through 1.0 opaque)
`animated`	[True | False]
`axes`	an `Axes` instance
`backgroundcolor`	any matplotlib color
`bbox`	rectangle prop dict
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color`	any matplotlib color
`contains`	a callable function
`family` or fontfamily or fontname or name	[FONTNAME | 'serif' | 'sans-serif' | 'cursive' | 'fantasy' | 'mono
`figure`	a `matplotlib.figure.Figure` instance
`fontproperties` or font_properties	a `matplotlib.font_manager.FontProperties` instance
`gid`	an id string
`horizontalalignment` or ha	['center' | 'right' | 'left']
`label`	string or anything printable with '%s' conversion.
`linespacing`	float (multiple of font size)
`lod`	[True | False]

Table 50.4 – cont

Property	Description
multialignment	['left' | 'right' | 'center']
path_effects	unknown
picker	[None|float|boolean|callable]
position	(x,y)
rasterized	[True | False | None]
rotation	[angle in degrees | 'vertical' | 'horizontal']
rotation_mode	unknown
size or fontsize	[size in points | 'xx-small' | 'x-small' | 'small' | 'medium' | 'large
snap	unknown
stretch or fontstretch	[a numeric value in range 0-1000 | 'ultra-condensed' | 'extra-cond
style or fontstyle	['normal' | 'italic' | 'oblique']
text	string or anything printable with '%s' conversion.
transform	Transform instance
url	a url string
variant or fontvariant	['normal' | 'small-caps']
verticalalignment or va or ma	['center' | 'top' | 'bottom' | 'baseline']
visible	[True | False]
weight or fontweight	[a numeric value in range 0-1000 | 'ultralight' | 'light' | 'normal' |
x	float
y	float
zorder	any number

cached = {}

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 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_path_effects()

get_position()
Return the position of the text as a tuple (x, y)

get_prop_tup()
Return a hashable tuple of properties.

Not intended to be human readable, but useful for backends who want to cache derived information about text (eg 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_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.

static **is_math_text**(*s*)
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".

set_backgroundcolor(*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 rectangle, eg facecolor='red', alpha=0.5.

t.set_bbox(dict(facecolor='red', alpha=0.5))

If rectprops has "boxstyle" key. A FancyBboxPatch is initialized with rectprops and will be drawn. The mutation scale of the FancyBboxPath is set to the fontsize.

ACCEPTS: rectangle prop dict

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 bu 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_path_effects(*path_effects*)

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 alignement 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).

ACCEPTS: [a numeric value in range 0-1000 | 'ultra-condensed' | 'extra-condensed' | 'condensed' | 'semi-condensed' | 'normal' | 'semi-expanded' | 'expanded' | 'extra-expanded' | 'ultra-expanded']

`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_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.

ACCEPTS: [a numeric value in range 0-1000 | 'ultralight' | 'light' | 'normal' | 'regular' | 'book' | 'medium' | 'roman' | 'semibold' | 'demibold' | 'demi' | 'bold' | 'heavy' | 'extra bold' | 'black']

`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_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()
Return a hashable tuple of properties.

Not intended to be human readable, but useful for backends who want to cache derived information about text (eg layouts) and need to know if the text has changed.

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.

rotation may be 'horizontal', 'vertical', or a numeric value in degrees.

CHAPTER

FIFTYONE

AXES

51.1 matplotlib.axes

class `matplotlib.axes.`**`Axes`**(*fig*, *rect*, *axisbg=None*, *frameon=True*, *sharex=None*, *sharey=None*, *label=''*, *xscale=None*, *yscale=None*, ***kwargs*)

Bases: `matplotlib.artist.Artist`

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.

`acorr`(*x*, ***kwargs*)

Plot the autocorrelation of *x*.

Call signature:

```
acorr(x, normed=True, detrend=mlab.detrend_none, usevlines=True,
      maxlags=10, **kwargs)
```

If *normed* = *True*, normalize the data by the autocorrelation at 0-th lag. *x* is detrended by the *detrend* callable (default no normalization).

Data are plotted as `plot(lags, c, **kwargs)`

Return value is a tuple (*lags*, *c*, *line*) 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()`

The default *linestyle* is None and the default *marker* is `'o'`, though these can be overridden with keyword args. The cross correlation is performed with `numpy.correlate()` with *mode* = 2.

If *usevlines* is *True*, `vlines()` rather than `plot()` is used to draw vertical lines from the origin to the acorr. Otherwise, the plot style is determined by the kwargs, which are `Line2D` properties.

maxlags is a positive integer detailing the number of lags to show. The default value of *None* will return all `(2*len(x)-1)` lags.

The return value is a tuple (*lags*, *c*, *linecol*, *b*) where

- *linecol* is the `LineCollection`
- *b* is the *x*-axis.

See Also:

`plot()` or `vlines()` For documentation on valid kwargs.

Example:

`xcorr()` is top graph, and `acorr()` is bottom graph.

add_artist(*a*)
Add any `Artist` to the axes.

Returns the artist.

add_collection(*collection*, *autolim=True*)
Add a `Collection` instance to the axes.

Returns the collection.

add_container(*container*)
Add a `Container` instance to the axes.

Returns the collection.

add_line(*line*)
Add a `Line2D` to the list of plot lines

Returns the line.

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.

add_table(*tab*)
Add a `Table` instance to the list of axes tables

Returns the table.

annotate(**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)
```

Keyword arguments:

Annotate the *x*, *y* point *xy* with text *s* at *x*, *y* location *xytext*. (If *xytext* = *None*, defaults to *xy*, and if *textcoords* = *None*, defaults to *xycoords*).

arrowprops, if not *None*, is a dictionary of line properties (see `matplotlib.lines.Line2D`) for the arrow that connects annotation to the point.

If the dictionary has a key *arrowstyle*, a FancyArrowPatch instance is created with the given dictionary and is drawn. Otherwise, a YAArow patch instance is created and drawn. Valid keys for YAArow are

Key	Description
width	the width of the arrow in points
frac	the fraction of the arrow length occupied by the head
head-width	the width of the base of the arrow head in points
shrink	oftentimes it is convenient to have the arrowtip and base a bit away from the text and point being annotated. If *d* is the distance between the text and annotated point, shrink will shorten the arrow so the tip and base are shink percent of the distance *d* away from the endpoints. ie, `shrink=0.05 is 5%`
?	any key for `matplotlib.patches.polygon`

Valid keys for FancyArrowPatch are

Key	Description
arrowstyle	the arrow style
connectionstyle	the connection style
relpos	default is (0.5, 0.5)
patchA	default is bounding box of the text
patchB	default is None
shrinkA	default is 2 points
shrinkB	default is 2 points
mutation_scale	default is text size (in points)
mutation_aspect	default is 1.
?	any key for `matplotlib.patches.PathPatch`

xycoords and *textcoords* are strings that indicate the coordinates of *xy* and *xytext*.

Property	Description
'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'	0,1 is lower left of axes and 1,1 is upper right
'data'	use the coordinate system of the object being annotated (default)
'offset points'	Specify an offset (in points) from the *xy* value
'polar'	you can specify *theta*, *r* 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.

If a 'points' or 'pixels' option is specified, values will be added to the bottom-left and if negative, values will be subtracted from the top-right. Eg:

```
# 10 points to the right of the left border of the axes and
# 5 points below the top border
xy=(10,-5), xycoords='axes points'
```

You may use an instance of `Transform` or `Artist`. See *Annotating Axes* for more details.

The *annotation_clip* attribute contols 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".

Additional kwargs are Text properties:

Property	Description
`agg_filter`	unknown
`alpha`	float (0.0 transparent through 1.0 opaque)
`animated`	[True | False]
`axes`	an `Axes` instance
`backgroundcolor`	any matplotlib color

Table 51.1 – cont

Property	Description
bbox	rectangle prop dict
clip_box	a matplotlib.transforms.Bbox instance
clip_on	[True | False]
clip_path	[(Path, Transform) | Patch | None]
color	any matplotlib color
contains	a callable function
family or fontfamily or fontname or name	[FONTNAME | 'serif' | 'sans-serif' | 'cursive' | 'fantasy' | 'mono
figure	a matplotlib.figure.Figure instance
fontproperties or font_properties	a matplotlib.font_manager.FontProperties instance
gid	an id string
horizontalalignment or ha	['center' | 'right' | 'left']
label	string or anything printable with '%s' conversion.
linespacing	float (multiple of font size)
lod	[True | False]
multialignment	['left' | 'right' | 'center']
path_effects	unknown
picker	[None|float|boolean|callable]
position	(x,y)
rasterized	[True | False | None]
rotation	[angle in degrees | 'vertical' | 'horizontal']
rotation_mode	unknown
size or fontsize	[size in points | 'xx-small' | 'x-small' | 'small' | 'medium' | 'large
snap	unknown
stretch or fontstretch	[a numeric value in range 0-1000 | 'ultra-condensed' | 'extra-conc
style or fontstyle	['normal' | 'italic' | 'oblique']
text	string or anything printable with '%s' conversion.
transform	Transform instance
url	a url string
variant or fontvariant	['normal' | 'small-caps']
verticalalignment or va or ma	['center' | 'top' | 'bottom' | 'baseline']
visible	[True | False]
weight or fontweight	[a numeric value in range 0-1000 | 'ultralight' | 'light' | 'normal' |
x	float
y	float
zorder	any number

apply_aspect(*position=None*)

Use `_aspect()` and `_adjustable()` to modify the axes box or the view limits.

arrow(*x*, *y*, *dx*, *dy*, ***kwargs*)

Add an arrow to the axes.

Call signature:

```
arrow(x, y, dx, dy, **kwargs)
```

Draws arrow on specified axis from (x, y) to (x + dx, y + dy). Uses FancyArrow patch to construct the arrow.

Optional kwargs 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 `Patch`) 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
clip_box	a matplotlib.transforms.Bbox instance
clip_on	[True | False]
clip_path	[(Path, Transform) | Patch | None]
color	matplotlib color spec
contains	a callable function
edgecolor or ec	mpl color spec, or None for default, or 'none' for no color
facecolor or fc	mpl color spec, or None for default, or 'none' for no color
figure	a matplotlib.figure.Figure instance
fill	[True | False]
gid	an id string
hatch	['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
label	string or anything printable with '%s' conversion.
linestyle or ls	['solid' | 'dashed' | 'dashdot' | 'dotted']
linewidth or lw	float or None for default
lod	[True | False]
path_effects	unknown
picker	[None|float|boolean|callable]
rasterized	[True | False | None]
snap	unknown
transform	Transform instance
url	a url string
visible	[True | False]
zorder	any number

Example:

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.

autoscale_view(*tight=None*, *scalex=True*, *scaley=True*)

Autoscale the view limits using the data limits. You can selectively autoscale only a single axis, eg, the xaxis by setting *scaley* to *False*. The autoscaling preserves any axis direction reversal that has already been done.

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.

axhline(*y=0*, *xmin=0*, *xmax=1*, ***kwargs*)

Add a horizontal line across the axis.

Call signature:

```
axhline(y=0, xmin=0, xmax=1, **kwargs)
```

Draw a horizontal line at *y* from *xmin* to *xmax*. With the default values of *xmin* = 0 and *xmax* = 1, this line will always span the horizontal extent of the axes, regardless of the xlim settings, even if you change them, eg. 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.

Return value is the `Line2D` instance. kwargs are the same as kwargs to plot, and can be used to control the line properties. Eg.,

- 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:

```
>>> axhline(y=1)
```

- draw a default hline at $y = .5$ that spans the the middle half of the xrange:

```
>>> axhline(y=.5, xmin=0.25, xmax=0.75)
```

Valid kwargs are `Line2D` properties, with the exception of 'transform':

Property	Description
`agg_filter`	unknown

Table 51.2

<table>
<tr><th>Property</th><th>Description</th></tr>
<tr><td>alpha</td><td>float (0.0 transparent through 1.0 opaque)</td></tr>
<tr><td>animated</td><td>[True | False]</td></tr>
<tr><td>antialiased or aa</td><td>[True | False]</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 | False]</td></tr>
<tr><td>clip_path</td><td>[(Path, Transform) | Patch | None]</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' | 'round' | 'projecting']</td></tr>
<tr><td>dash_joinstyle</td><td>['miter' | 'round' | 'bevel']</td></tr>
<tr><td>dashes</td><td>sequence of on/off ink in points</td></tr>
<tr><td>data</td><td>2D array (rows are x, y) or two 1D arrays</td></tr>
<tr><td>drawstyle</td><td>['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post']</td></tr>
<tr><td>figure</td><td>a matplotlib.figure.Figure instance</td></tr>
<tr><td>fillstyle</td><td>['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']</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>['-' | '--' | '-.' | ':' | 'None' | ' ' | ''] and any drawstyle in combina</td></tr>
<tr><td>linewidth or lw</td><td>float value in points</td></tr>
<tr><td>lod</td><td>[True | False]</td></tr>
<tr><td>marker</td><td>[7 | 4 | 5 | 6 | 'o' | 'D' | 'h' | 'H' | '_' | '' | 'None' | None | ' ' | '8' | 'p</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 | integer | (startind, stride)</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 | False | None]</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>
</table>

See Also:

`axhspan()` for example plot and source code

axhspan(*ymin*, *ymax*, *xmin=0*, *xmax=1*, ***kwargs*)

Add a horizontal span (rectangle) across the axis.

Call signature:

```
axhspan(ymin, ymax, xmin=0, xmax=1, **kwargs)
```

y coords are in data units and *x* coords are in axes (relative 0-1) units.

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, eg. 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.

Return value is a `matplotlib.patches.Polygon` instance.

Examples:

- draw a gray rectangle from *y* = 0.25-0.75 that spans the horizontal extent of the axes:

```
>>> axhspan(0.25, 0.75, facecolor='0.5', alpha=0.5)
```

Valid kwargs are `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
clip_box	a matplotlib.transforms.Bbox instance
clip_on	[True | False]
clip_path	[(Path, Transform) | Patch | None]
color	matplotlib color spec
contains	a callable function
edgecolor or ec	mpl color spec, or None for default, or 'none' for no color
facecolor or fc	mpl color spec, or None for default, or 'none' for no color
figure	a matplotlib.figure.Figure instance
fill	[True | False]
gid	an id string
hatch	['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
label	string or anything printable with '%s' conversion.
linestyle or ls	['solid' | 'dashed' | 'dashdot' | 'dotted']
linewidth or lw	float or None for default
lod	[True | False]
path_effects	unknown
picker	[None|float|boolean|callable]
rasterized	[True | False | None]
snap	unknown
transform	Transform instance
url	a url string
visible	[True | False]
zorder	any number

Example:

axis(**v*, ***kwargs*)

Convenience method for manipulating the x and y view limits and the aspect ratio of the plot. For details, see `axis()`.

kwargs are passed on to `set_xlim()` and `set_ylim()`

axvline(*x=0*, *ymin=0*, *ymax=1*, ***kwargs*)

Add a vertical line across the axes.

Call signature:

```
axvline(x=0, ymin=0, ymax=1, **kwargs)
```

Draw a vertical line at *x* from *ymin* to *ymax*. With the default values of *ymin* = 0 and *ymax* = 1, this line will always span the vertical extent of the axes, regardless of the ylim settings, even if you change them, eg. with the `set_ylim()` command. That is, the vertical extent is in axes coords: 0=bottom, 0.5=middle, 1.0=top but the *x* location is in data coordinates.

Return value is the `Line2D` instance. kwargs are the same as kwargs to plot, and can be used to control the line properties. Eg.,

- 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 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':

Property	Description
`agg_filter`	unknown
`alpha`	float (0.0 transparent through 1.0 opaque)
`animated`	[True | False]
`antialiased` or aa	[True | False]
`axes`	an `Axes` instance
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color` or c	any matplotlib color
`contains`	a callable function
`dash_capstyle`	['butt' | 'round' | 'projecting']
`dash_joinstyle`	['miter' | 'round' | 'bevel']
`dashes`	sequence of on/off ink in points
`data`	2D array (rows are x, y) or two 1D arrays
`drawstyle`	['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post']
`figure`	a `matplotlib.figure.Figure` instance
`fillstyle`	['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']
`gid`	an id string

Table 51.3

Property	Description
label	string or anything printable with '%s' conversion.
linestyle or ls	['-' | '--' | '-.' | ':' | 'None' | ' ' | "] and any drawstyle in combina
linewidth or lw	float value in points
lod	[True | False]
marker	[7 | 4 | 5 | 6 | 'o' | 'D' | 'h' | 'H' | '_' | " | 'None' | None | ' ' | '8' | 'p
markeredgecolor or mec	any matplotlib color
markeredgewidth or mew	float value in points
markerfacecolor or mfc	any matplotlib color
markerfacecoloralt or mfcalt	any matplotlib color
markersize or ms	float
markevery	None | integer | (startind, stride)
picker	float distance in points or callable pick function `fn(artist, event)`
pickradius	float distance in points
rasterized	[True | False | None]
snap	unknown
solid_capstyle	['butt' | 'round' | 'projecting']
solid_joinstyle	['miter' | 'round' | 'bevel']
transform	a `matplotlib.transforms.Transform` instance
url	a url string
visible	[True | False]
xdata	1D array
ydata	1D array
zorder	any number

See Also:

axhspan() for example plot and source code

axvspan(*xmin*, *xmax*, *ymin=0*, *ymax=1*, ***kwargs*)
Add a vertical span (rectangle) across the axes.

Call signature:

```
axvspan(xmin, xmax, ymin=0, ymax=1, **kwargs)
```

x coords are in data units and *y* coords are in axes (relative 0-1) units.

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, eg. 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.

Return value is the `matplotlib.patches.Polygon` instance.

Examples:

- draw a vertical green translucent rectangle from x=1.25 to 1.55 that spans the yrange of the axes:

```
>>> axvspan(1.25, 1.55, facecolor='g', alpha=0.5)
```

Valid kwargs are `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
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color`	matplotlib color spec
`contains`	a callable function
`edgecolor` or ec	mpl color spec, or None for default, or 'none' for no color
`facecolor` or fc	mpl color spec, or None for default, or 'none' for no color
`figure`	a `matplotlib.figure.Figure` instance
`fill`	[True | False]
`gid`	an id string
`hatch`	['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
`label`	string or anything printable with '%s' conversion.
`linestyle` or ls	['solid' | 'dashed' | 'dashdot' | 'dotted']
`linewidth` or lw	float or None for default
`lod`	[True | False]
`path_effects`	unknown
`picker`	[None|float|boolean|callable]
`rasterized`	[True | False | None]
`snap`	unknown
`transform`	`Transform` instance
`url`	a url string
`visible`	[True | False]
`zorder`	any number

See Also:

`axhspan()` for example plot and source code

bar(*left*, *height*, *width=0.8*, *bottom=None*, ***kwargs*)

Make a bar plot.

Call signature:

```
bar(left, height, width=0.8, bottom=0, **kwargs)
```

Make a bar plot with rectangles bounded by:

> ***left, left + width, bottom, bottom + height*** (left, right, bottom and top edges)

left, *height*, *width*, and *bottom* can be either scalars or sequences

Return value is a list of `matplotlib.patches.Rectangle` instances.

Required arguments:

Argument	Description
left	the x coordinates of the left sides of the bars
height	the heights of the bars

Optional keyword arguments:

Keyword	Description
width	the widths of the bars
bottom	the y coordinates of the bottom edges of the bars
color	the colors of the bars
edgecolor	the colors of the bar edges
linewidth	width of bar edges; None means use default linewidth; 0 means don't draw edges.
xerr	if not None, will be used to generate errorbars on the bar chart
yerr	if not None, will be used to generate errorbars on the bar chart
ecolor	specifies the color of any errorbar
capsize	(default 3) determines the length in points of the error bar caps
error_kw	dictionary of kwargs to be passed to errorbar method. *ecolor* and *capsize* may be specified here rather than as independent kwargs.
align	'edge' (default) | 'center'
orientation	'vertical' | 'horizontal'
log	[False|True] False (default) leaves the orientation axis as-is; True sets it to log scale

For vertical bars, *align* = 'edge' aligns bars by their left edges in left, while *align* = 'center' interprets these values as the *x* coordinates of the bar centers. For horizontal bars, *align* = 'edge' aligns bars by their bottom edges in bottom, while *align* = 'center' interprets these values as the *y* coordinates of the bar centers.

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:

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
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color`	matplotlib color spec
`contains`	a callable function
`edgecolor` or ec	mpl color spec, or None for default, or 'none' for no color
`facecolor` or fc	mpl color spec, or None for default, or 'none' for no color
`figure`	a `matplotlib.figure.Figure` instance
`fill`	[True | False]
`gid`	an id string
`hatch`	['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
`label`	string or anything printable with '%s' conversion.
`linestyle` or ls	['solid' | 'dashed' | 'dashdot' | 'dotted']
`linewidth` or lw	float or None for default
`lod`	[True | False]
`path_effects`	unknown
`picker`	[None|float|boolean|callable]
`rasterized`	[True | False | None]
`snap`	unknown
`transform`	`Transform` instance
`url`	a url string
`visible`	[True | False]
`zorder`	any number

Example: A stacked bar chart.

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 analagous 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 analagous 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 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.

linewidths and edgecolors can be used to customize the barb. Additional PolyCollection keyword arguments:

Property	Description
agg_filter	unknown
alpha	float or None
animated	[True | False]
antialiased or antialiaseds	Boolean or sequence of booleans
array	unknown
axes	an Axes instance
clim	a length 2 sequence of floats
clip_box	a matplotlib.transforms.Bbox instance
clip_on	[True | False]
clip_path	[(Path, Transform) | Patch | None]
cmap	a colormap or registered colormap name
color	matplotlib color arg or sequence of rgba tuples
colorbar	unknown
contains	a callable function
edgecolor or edgecolors	matplotlib color arg or sequence of rgba tuples
facecolor or facecolors	matplotlib color arg or sequence of rgba tuples
figure	a matplotlib.figure.Figure instance
gid	an id string
hatch	['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
label	string or anything printable with '%s' conversion.
linestyle or linestyles or dashes	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
linewidth or lw or linewidths	float or sequence of floats
lod	[True | False]
norm	unknown
offset_position	unknown
offsets	float or sequence of floats
paths	unknown
picker	[None|float|boolean|callable]
pickradius	unknown
rasterized	[True | False | None]
snap	unknown
transform	Transform instance
url	a url string
urls	unknown
visible	[True | False]

Continued on next page

Table 51.4 – continued from previous page

Property	Description
zorder	any number

Example:

barh(*bottom*, *width*, *height=0.8*, *left=None*, ***kwargs*)

Make a horizontal bar plot.

Call signature:

```
barh(bottom, width, height=0.8, left=0, **kwargs)
```

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

Return value is a list of `matplotlib.patches.Rectangle` instances.

Required arguments:

Argument	Description
bottom	the vertical positions of the bottom edges of the bars
width	the lengths of the bars

Optional keyword arguments:

Keyword	Description
height	the heights (thicknesses) of the bars
left	the x coordinates of the left edges of the bars
color	the colors of the bars
edgecolor	the colors of the bar edges
linewidth	width of bar edges; None means use default linewidth; 0 means don't draw edges.
xerr	if not None, will be used to generate errorbars on the bar chart
yerr	if not None, will be used to generate errorbars on the bar chart
ecolor	specifies the color of any errorbar
capsize	(default 3) determines the length in points of the error bar caps
align	'edge' (default) | 'center'
log	[False|True] False (default) leaves the horizontal axis as-is; True sets it to log scale

Setting *align* = 'edge' aligns bars by their bottom edges in bottom, while *align* = 'center' interprets these values as the *y* coordinates of the bar centers.

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 barh as the basis for stacked bar charts, or candlestick plots.

other optional kwargs:

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
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color`	matplotlib color spec
`contains`	a callable function
`edgecolor` or ec	mpl color spec, or None for default, or 'none' for no color
`facecolor` or fc	mpl color spec, or None for default, or 'none' for no color
`figure`	a `matplotlib.figure.Figure` instance
`fill`	[True | False]
`gid`	an id string
`hatch`	['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
`label`	string or anything printable with '%s' conversion.
`linestyle` or ls	['solid' | 'dashed' | 'dashdot' | 'dotted']
`linewidth` or lw	float or None for default
`lod`	[True | False]
`path_effects`	unknown
`picker`	[None|float|boolean|callable]
`rasterized`	[True | False | None]
`snap`	unknown
`transform`	`Transform` instance
`url`	a url string
`visible`	[True | False]
`zorder`	any number

boxplot(*x*, *notch=False*, *sym='b+'*, *vert=True*, *whis=1.5*, *positions=None*, *widths=None*, *patch_artist=False*, *bootstrap=None*, *usermedians=None*, *conf_intervals=None*)

Make a box and whisker plot.

Call signature:

```
boxplot(x, notch=False, sym='+', vert=True, whis=1.5,
        positions=None, widths=None, patch_artist=False,
        bootstrap=None, usermedians=None, conf_intervals=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.

Function Arguments:

x **:** Array or a sequence of vectors.

notch [[False (default) | True]] If False (default), produces a rectangular box plot. If True, will produce a notched box plot

sym [[default 'b+']] The default symbol for flier points. Enter an empty string ('') if you don't want to show fliers.

vert [[False | True (default)]] If True (default), makes the boxes vertical. If False, makes horizontal boxes.

whis [[default 1.5]] Defines the length of the whiskers as a function of the inner quartile range. They extend to the most extreme data point within (`whis*(75%-25%)`) data range.

bootstrap [[*None* (default) | integer]] Specifies whether to bootstrap the confidence intervals around the median for notched boxplots. If bootstrap==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 it's 95% confidence intervals. Values between 1000 and 10000 are recommended.

usermedians [[default None]] 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* == None, the median will be computed directly as normal.

conf_intervals [[default None]] Array or sequence whose first dimension (or length) is compatible with *x* and whose second dimension is 2. When the current element of *conf_intervals* is not None, the notch locations computed by matplotlib are overridden (assuming notch is True). When an element of *conf_intervals* is None, boxplot compute notches the method specified by the other kwargs (e.g. *bootstrap*).

positions [[default 1,2,...,n]] Sets the horizontal positions of the boxes. The ticks and limits are automatically set to match the positions.

widths [[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.

patch_artist [[False (default) | True]] If False produces boxes with the Line2D artist If True produces boxes with the Patch artist

Returns 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: horizonal lines at the median of each box.
- whiskers: the vertical lines extending to the most extreme, n-outlier data points.
- caps: the horizontal lines at the ends of the whiskers.
- fliers: points representing data that extend beyone the whiskers (outliers).

Example:

broken_barh(*xranges*, *yrange*, ***kwargs*)

Plot horizontal bars.

Call signature:

```
broken_barh(self, xranges, yrange, **kwargs)
```

A collection of horizontal bars spanning *yrange* with a sequence of *xranges*.

Required arguments:

Argument	Description
xranges	sequence of (*xmin*, *xwidth*)
yrange	sequence of (*ymin*, *ywidth*)

kwargs are `matplotlib.collections.BrokenBarHCollection` properties:

Property	Description
`agg_filter`	unknown
`alpha`	float or None
`animated`	[True | False]
`antialiased` or antialiaseds	Boolean or sequence of booleans
`array`	unknown
`axes`	an `Axes` instance
`clim`	a length 2 sequence of floats

Continued on next page

Table 51.5 – continued from previous page

Property	Description
clip_box	a matplotlib.transforms.Bbox instance
clip_on	[True | False]
clip_path	[(Path, Transform) | Patch | None]
cmap	a colormap or registered colormap name
color	matplotlib color arg or sequence of rgba tuples
colorbar	unknown
contains	a callable function
edgecolor or edgecolors	matplotlib color arg or sequence of rgba tuples
facecolor or facecolors	matplotlib color arg or sequence of rgba tuples
figure	a matplotlib.figure.Figure instance
gid	an id string
hatch	['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
label	string or anything printable with '%s' conversion.
linestyle or linestyles or dashes	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
linewidth or lw or linewidths	float or sequence of floats
lod	[True | False]
norm	unknown
offset_position	unknown
offsets	float or sequence of floats
paths	unknown
picker	[None|float|boolean|callable]
pickradius	unknown
rasterized	[True | False | None]
snap	unknown
transform	Transform instance
url	a url string
urls	unknown
visible	[True | False]
zorder	any number

these can either be a single argument, ie:

```
facecolors = 'black'
```

or a sequence of arguments for the various bars, ie:

```
facecolors = ('black', 'red', 'green')
```

Example:

can_pan()

Return *True* if this axes supports any pan/zoom button functionality.

can_zoom()
Return *True* if this axes supports the zoom box button functionality.

cla()
Clear the current axes.

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 eg '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.

clear()

clear the axes

cohere(*x*, *y*, *NFFT=256*, *Fs=2*, *Fc=0*, *detrend=<function detrend_none at 0x9bc0a3c>*, *window=<function window_hanning at 0x9bc0924>*, *noverlap=0*, *pad_to=None*, *sides='default'*, *scale_by_freq=None*, ***kwargs*)

Plot the coherence between *x* and *y*.

Call signature:

```
cohere(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, **kwargs)
```

Plot the coherence between *x* and *y*. Coherence is the normalized cross spectral density:

$$C_{xy} = \frac{|P_{xy}|^2}{P_{xx}P_{yy}} \tag{51.1}$$

Keyword arguments:

***NFFT*: integer** The number of data points used in each block for the FFT. Must be even; 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.

***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.

***detrend*: 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.

***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.

***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 psd (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*

***sides*: ['default' | 'onesided' | 'twosided']** Specifies which sides of the PSD 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 PSD, while 'twosided' forces two-sided.

***scale_by_freq*: boolean** 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.

The return value is a tuple (Cxy, f), where f are the frequencies of the coherence vector.

kwargs are applied to the lines.

References:

- Bendat & Piersol – Random Data: Analysis and Measurement Procedures, John Wiley & Sons (1986)

kwargs control the `Line2D` properties of the coherence plot:

Property	Description
`agg_filter`	unknown
`alpha`	float (0.0 transparent through 1.0 opaque)
`animated`	[True | False]

Table 51.6

<table>
<tr><th>Property</th><th>Description</th></tr>
<tr><td>antialiased or aa</td><td>[True | False]</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 | False]</td></tr>
<tr><td>clip_path</td><td>[(Path, Transform) | Patch | None]</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' | 'round' | 'projecting']</td></tr>
<tr><td>dash_joinstyle</td><td>['miter' | 'round' | 'bevel']</td></tr>
<tr><td>dashes</td><td>sequence of on/off ink in points</td></tr>
<tr><td>data</td><td>2D array (rows are x, y) or two 1D arrays</td></tr>
<tr><td>drawstyle</td><td>['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post']</td></tr>
<tr><td>figure</td><td>a matplotlib.figure.Figure instance</td></tr>
<tr><td>fillstyle</td><td>['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']</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>['-' | '--' | '-.' | ':' | 'None' | ' ' | ''] and any drawstyle in combina</td></tr>
<tr><td>linewidth or lw</td><td>float value in points</td></tr>
<tr><td>lod</td><td>[True | False]</td></tr>
<tr><td>marker</td><td>[7 | 4 | 5 | 6 | 'o' | 'D' | 'h' | 'H' | '_' | '' | 'None' | None | ' ' | '8' | 'p</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 | integer | (startind, stride)</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 | False | None]</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>
</table>

Example:

connect(*s*, *func*)
Register observers to be notified when certain events occur. Register with callback functions with the following signatures. The function has the following signature:

```
func(ax)  # where ax is the instance making the callback.
```

The following events can be connected to:

'xlim_changed','ylim_changed'

The connection id is is returned - you can use this with disconnect to disconnect from the axes event

contains(*mouseevent*)
Test whether the mouse event occured in the axes.

Returns *True* / *False*, {}

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.

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 *N* automatically-chosen levels.

```
contour(Z,V)
contour(X,Y,Z,V)
```

draw contour lines at the values specified in sequence *V* contourf(..., V)

fill the len(V)-1 regions between the values in *V*

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:

***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; eg 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 rcParams['lines.antialiased'].

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:

***nchunk*: [0 | integer]** If 0, no subdivision of the domain. Specify a positive integer to divide the domain into subdomains of roughly *nchunk* by *nchunk* points. This may never actually be advantageous, so this option may be removed. Chunking introduces artifacts at the chunk boundaries unless *antialiased* is *False*.

***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 *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:

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 *N* automatically-chosen levels.

```
contour(Z,V)
contour(X,Y,Z,V)
```

draw contour lines at the values specified in sequence *V*

```
contourf(..., V)
```

fill the `len(V)-1` regions between the values in *V*

```
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:

***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; eg 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 rcParams['lines.antialiased'].

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:

***nchunk*: [0 | integer]** If 0, no subdivision of the domain. Specify a positive integer to divide the domain into subdomains of roughly *nchunk* by *nchunk* points. This may never actually be advantageous, so this option may be removed. Chunking introduces artifacts at the chunk boundaries unless *antialiased* is *False*.

***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 *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:

csd(*x*, *y*, *NFFT=256*, *Fs=2*, *Fc=0*, *detrend=<function detrend_none at 0x9bc0a3c>*, *window=<function window_hanning at 0x9bc0924>*, *noverlap=0*, *pad_to=None*, *sides='default'*, *scale_by_freq=None*, ***kwargs*)

Plot 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, **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*. 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.

Returns the tuple (*Pxy*, *freqs*). *P* is the cross spectrum (complex valued), and $10\log_{10}|P_{xy}|$ is plotted.

Keyword arguments:

***NFFT*: integer** The number of data points used in each block for the FFT. Must be even; 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.

***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.

***detrend*: 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.

***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.

***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 psd (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*

***sides*: ['default' | 'onesided' | 'twosided']** Specifies which sides of the PSD 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 PSD, while 'twosided' forces two-sided.

***scale_by_freq*: boolean** 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.

References: Bendat & Piersol – Random Data: Analysis and Measurement Procedures, John Wiley & Sons (1986)

kwargs control the Line2D properties:

Property	Description
`agg_filter`	unknown
`alpha`	float (0.0 transparent through 1.0 opaque)
`animated`	[True | False]
`antialiased` or aa	[True | False]
`axes`	an `Axes` instance
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color` or c	any matplotlib color
`contains`	a callable function
`dash_capstyle`	['butt' | 'round' | 'projecting']
`dash_joinstyle`	['miter' | 'round' | 'bevel']
`dashes`	sequence of on/off ink in points
`data`	2D array (rows are x, y) or two 1D arrays

Table 51.7

Property	Description
drawstyle	['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post']
figure	a matplotlib.figure.Figure instance
fillstyle	['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']
gid	an id string
label	string or anything printable with '%s' conversion.
linestyle or ls	['-' | '--' | '-.' | ':' | 'None' | ' ' | "] and any drawstyle in combina
linewidth or lw	float value in points
lod	[True | False]
marker	[7 | 4 | 5 | 6 | 'o' | 'D' | 'h' | 'H' | '_' | " | 'None' | None | ' ' | '8' | 'p
markeredgecolor or mec	any matplotlib color
markeredgewidth or mew	float value in points
markerfacecolor or mfc	any matplotlib color
markerfacecoloralt or mfcalt	any matplotlib color
markersize or ms	float
markevery	None | integer | (startind, stride)
picker	float distance in points or callable pick function fn(artist, event)
pickradius	float distance in points
rasterized	[True | False | None]
snap	unknown
solid_capstyle	['butt' | 'round' | 'projecting']
solid_joinstyle	['miter' | 'round' | 'bevel']
transform	a matplotlib.transforms.Transform instance
url	a url string
visible	[True | False]
xdata	1D array
ydata	1D array
zorder	any number

Example:

disconnect(*cid*)

disconnect from the Axes event.

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.

---

draw(*artist*, *renderer*, **args*, ***kwargs*)
Draw everything (plot lines, axes, labels)

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)

end_pan()
Called when a pan operation completes (when the mouse button is up.)

Note: Intended to be overridden by new projection types.

---

errorbar(*x*, *y*, *yerr=None*, *xerr=None*, *fmt='-'*, *ecolor=None*, *elinewidth=None*, *capsize=3*, *barsabove=False*, *lolims=False*, *uplims=False*, *xlolims=False*, *xuplims=False*, *errorevery=1*, *capthick=None*, ***kwargs*)
Plot an errorbar graph.

Call signature:

```
errorbar(x, y, yerr=None, xerr=None,
         fmt='-', ecolor=None, elinewidth=None, capsize=3,
         barsabove=False, lolims=False, uplims=False,
         xlolims=False, xuplims=False, errorevery=1,
         capthick=None)
```

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*.

Optional keyword arguments:

***xerr/yerr*: [scalar | N, Nx1, or 2xN array-like]** If a scalar number, len(N) array-like object, or an Nx1 array-like object, errorbars are drawn +/- value.

If a sequence of shape 2xN, errorbars are drawn at -row1 and +row2

***fmt*: '-'** The plot format symbol. If *fmt* is *None*, only the errorbars are plotted. This is used for adding errorbars to a bar plot, for example.

***ecolor*: [*None* | mpl color]** A matplotlib color arg which gives the color the errorbar lines; if *None*, use the marker color.

***elinewidth*: scalar** The linewidth of the errorbar lines. If *None*, use the linewidth.

***capsize*: scalar** The length of the error bar caps in points

***capthick*: scalar** 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*: [*True* | *False*]** if *True*, will plot the errorbars above the plot symbols. Default is below.

***lolims* / *uplims* / *xlolims* / *xuplims*: [*False* | *True*]** 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*.

***errorevery*: positive integer** subsamples the errorbars. Eg if everyerror=5, errorbars for every 5-th datapoint will be plotted. The data plot itself still shows all data points.

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:

```
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 *markeredgewith*.

valid kwargs for the marker properties are

Property	Description
agg_filter	unknown
alpha	float (0.0 transparent through 1.0 opaque)
animated	[True | False]
antialiased or aa	[True | False]
axes	an Axes instance
clip_box	a matplotlib.transforms.Bbox instance
clip_on	[True | False]
clip_path	[(Path, Transform) | Patch | None]
color or c	any matplotlib color
contains	a callable function
dash_capstyle	['butt' | 'round' | 'projecting']

Table 51.8

Property	Description
`dash_joinstyle`	['miter' | 'round' | 'bevel']
`dashes`	sequence of on/off ink in points
`data`	2D array (rows are x, y) or two 1D arrays
`drawstyle`	['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post']
`figure`	a `matplotlib.figure.Figure` instance
`fillstyle`	['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']
`gid`	an id string
`label`	string or anything printable with '%s' conversion.
`linestyle` or ls	[`'-'` | `'--'` | `'-.'` | `':'` | `'None'` | `' '` | "] and any drawstyle in combina
`linewidth` or lw	float value in points
`lod`	[True | False]
`marker`	[7 | 4 | 5 | 6 | `'o'` | `'D'` | `'h'` | `'H'` | `'_'` | " | `'None'` | None | `' '` | `'8'` | `'p`
`markeredgecolor` or mec	any matplotlib color
`markeredgewidth` or mew	float value in points
`markerfacecolor` or mfc	any matplotlib color
`markerfacecoloralt` or mfcalt	any matplotlib color
`markersize` or ms	float
`markevery`	None | integer | (startind, stride)
`picker`	float distance in points or callable pick function `fn(artist, event)`
`pickradius`	float distance in points
`rasterized`	[True | False | None]
`snap`	unknown
`solid_capstyle`	['butt' | 'round' | 'projecting']
`solid_joinstyle`	['miter' | 'round' | 'bevel']
`transform`	a `matplotlib.transforms.Transform` instance
`url`	a url string
`visible`	[True | False]
`xdata`	1D array
`ydata`	1D array
`zorder`	any number

Returns (*plotline*, *caplines*, *barlinecols*):

***plotline*: `Line2D` instance** *x*, *y* plot markers and/or line

***caplines*: list of error bar cap** `Line2D` instances

***barlinecols*: list of** `LineCollection` instances for the horizontal and vertical error ranges.

Example:

fill(**args*, ***kwargs*)

Plot filled polygons.

Call signature:

```
fill(*args, **kwargs)
```

args is a variable length argument, allowing for multiple *x*, *y* pairs with an optional color format string; see `plot()` for details on the argument parsing. For example, to plot a polygon with vertices at *x*, *y* in blue.:

```
ax.fill(x,y, 'b' )
```

An arbitrary number of *x*, *y*, *color* groups can be specified:

```
ax.fill(x1, y1, 'g', x2, y2, 'r')
```

Return value is a list of `Patch` instances that were added.

The same color strings that `plot()` supports are supported by the fill format string.

If you would like to fill below a curve, eg. shade a region between 0 and *y* along *x*, use `fill_between()`

The *closed* kwarg will close the polygon when *True* (default).

kwargs control the `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
clip_box	a matplotlib.transforms.Bbox instance
clip_on	[True | False]
clip_path	[(Path, Transform) | Patch | None]
color	matplotlib color spec
contains	a callable function
edgecolor or ec	mpl color spec, or None for default, or 'none' for no color
facecolor or fc	mpl color spec, or None for default, or 'none' for no color
figure	a matplotlib.figure.Figure instance
fill	[True | False]
gid	an id string
hatch	['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
label	string or anything printable with '%s' conversion.
linestyle or ls	['solid' | 'dashed' | 'dashdot' | 'dotted']
linewidth or lw	float or None for default
lod	[True | False]
path_effects	unknown
picker	[None|float|boolean|callable]
rasterized	[True | False | None]
snap	unknown
transform	Transform instance
url	a url string
visible	[True | False]
zorder	any number

Example:

fill_between(*x*, *y1*, *y2=0*, *where=None*, *interpolate=False*, ***kwargs*)

Make filled polygons between two curves.

Call signature:

```
fill_between(x, y1, y2=0, where=None, **kwargs)
```

Create a PolyCollection filling the regions between *y1* and *y2* where `where==True`

***x* :** An N-length array of the x data

***y1* :** An N-length array (or scalar) of the y data

y2 **:** An N-length array (or scalar) of the y data

where **:** 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 **:** 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.

kwargs **:** Keyword args passed on to the `PolyCollection`.

kwargs control the `Polygon` properties:

Property	Description
`agg_filter`	unknown
`alpha`	float or None
`animated`	[True | False]
`antialiased` or antialiaseds	Boolean or sequence of booleans
`array`	unknown
`axes`	an `Axes` instance
`clim`	a length 2 sequence of floats
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`cmap`	a colormap or registered colormap name
`color`	matplotlib color arg or sequence of rgba tuples
`colorbar`	unknown
`contains`	a callable function
`edgecolor` or edgecolors	matplotlib color arg or sequence of rgba tuples
`facecolor` or facecolors	matplotlib color arg or sequence of rgba tuples
`figure`	a `matplotlib.figure.Figure` instance
`gid`	an id string
`hatch`	[‘/’ | ‘\’ | ‘|’ | ‘-‘ | ‘+’ | ‘x’ | ‘o’ | ‘O’ | ‘.’ | ‘*’]
`label`	string or anything printable with ‘%s’ conversion.
`linestyle` or linestyles or dashes	[‘solid’ | ‘dashed’, ‘dashdot’, ‘dotted’ | (offset, on-off-dash-seq)]
`linewidth` or lw or linewidths	float or sequence of floats
`lod`	[True | False]
`norm`	unknown
`offset_position`	unknown
`offsets`	float or sequence of floats
`paths`	unknown
`picker`	[None|float|boolean|callable]
`pickradius`	unknown
`rasterized`	[True | False | None]

Continued on next page

Table 51.9 – continued from previous page

Property	Description
snap	unknown
transform	Transform instance
url	a url string
urls	unknown
visible	[True | False]
zorder	any number

See Also:

fill_betweenx() for filling between two sets of x-values

fill_betweenx(*y*, *x1*, *x2=0*, *where=None*, ***kwargs*)

Make filled polygons between two horizontal curves.

Call signature:

```
fill_between(y, x1, x2=0, where=None, **kwargs)
```

Create a PolyCollection filling the regions between *x1* and *x2* where where==True

y **:** An N-length array of the y data

x1 **:** An N-length array (or scalar) of the x data

x2 **:** An N-length array (or scalar) of the x data

where **:** 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

kwargs **:** keyword args passed on to the PolyCollection

kwargs control the Polygon properties:

Property	Description
agg_filter	unknown
alpha	float or None
animated	[True | False]
antialiased or antialiaseds	Boolean or sequence of booleans
array	unknown
axes	an Axes instance
clim	a length 2 sequence of floats
clip_box	a matplotlib.transforms.Bbox instance

Continued on next page

Table 51.10 – continued from previous page

Property	Description
clip_on	[True | False]
clip_path	[(Path, Transform) | Patch | None]
cmap	a colormap or registered colormap name
color	matplotlib color arg or sequence of rgba tuples
colorbar	unknown
contains	a callable function
edgecolor or edgecolors	matplotlib color arg or sequence of rgba tuples
facecolor or facecolors	matplotlib color arg or sequence of rgba tuples
figure	a matplotlib.figure.Figure instance
gid	an id string
hatch	['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
label	string or anything printable with '%s' conversion.
linestyle or linestyles or dashes	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
linewidth or lw or linewidths	float or sequence of floats
lod	[True | False]
norm	unknown
offset_position	unknown
offsets	float or sequence of floats
paths	unknown
picker	[None|float|boolean|callable]
pickradius	unknown
rasterized	[True | False | None]
snap	unknown
transform	Transform instance
url	a url string
urls	unknown
visible	[True | False]
zorder	any number

See Also:

fill_between() for filling between two sets of y-values

format_coord(*x*, *y*)
Return a format string formatting the *x*, *y* coord

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

`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

`frame`

`get_adjustable()`

`get_anchor()`

`get_aspect()`

`get_autoscale_on()`
Get whether autoscaling is applied for both axes on plot commands

`get_autoscalex_on()`
Get whether autoscaling for the x-axis is applied on plot commands

`get_autoscaley_on()`
Get whether autoscaling for the y-axis is applied on plot commands

`get_axes_locator()`
return axes_locator

`get_axis_bgcolor()`
Return the axis background color

`get_axisbelow()`
Get whether axis below is true or not

`get_child_artists()`
Return a list of artists the axes contains. Deprecated since version 0.98.

`get_children()`
return a list of child artists

`get_cursor_props()`
Return the cursor propertiess as a (*linewidth*, *color*) tuple, where *linewidth* is a float and *color* is an RGBA tuple

`get_data_ratio()`
Returns the aspect ratio of the raw data.

This method is intended to be overridden by new projection types.

`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.

`get_default_bbox_extra_artists()`

`get_frame()`
Return the axes Rectangle frame

get_frame_on()
Get whether the axes rectangle patch is drawn

get_images()
return a list of Axes images contained by the Axes

get_legend()
Return the legend.Legend instance, or None if no legend is defined

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)
```

get_lines()
Return a list of lines contained by the Axes

get_navigate()
Get whether the axes responds to navigation commands

get_navigate_mode()
Get the navigation toolbar button status: 'PAN', 'ZOOM', or None

get_position(*original=False*)
Return the a copy of the axes rectangle as a Bbox

get_rasterization_zorder()
Get zorder value below which artists will be rasterized

get_renderer_cache()

get_shared_x_axes()
Return a copy of the shared axes Grouper object for x axes

get_shared_y_axes()
Return a copy of the shared axes Grouper object for y 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 intereted in the relative size of the tightbbox compared to the axes bbox.

get_title()
Get the title text string.

get_window_extent(**args*, ***kwargs*)
get the axes bounding box in display space; *args* and *kwargs* are empty

get_xaxis()
Return the XAxis instance

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.

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.

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.

get_xbound()
Returns the x-axis numerical bounds where:

lowerBound < upperBound

get_xgridlines()
Get the x grid lines as a list of Line2D instances

get_xlabel()
Get the xlabel text string.

get_xlim()
Get the x-axis range [*left*, *right*]

get_xmajorticklabels()
Get the xtick labels as a list of `Text` instances.

get_xminorticklabels()
Get the x minor tick labels as a list of `matplotlib.text.Text` instances.

get_xscale()
Return the xaxis scale string: linear, log, symlog

get_xticklabels(*minor=False*)
Get the x tick labels as a list of `Text` instances.

get_xticklines()
Get the xtick lines as a list of Line2D instances

get_xticks(*minor=False*)
Return the x ticks as a list of locations

get_yaxis()
Return the YAxis instance

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.

---

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.

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.

get_ybound()
Return y-axis numerical bounds in the form of lowerBound < upperBound

get_ygridlines()
Get the y grid lines as a list of Line2D instances

get_ylabel()
Get the ylabel text string.

get_ylim()
Get the y-axis range [*bottom*, *top*]

get_ymajorticklabels()
Get the major y tick labels as a list of `Text` instances.

get_yminorticklabels()
Get the minor y tick labels as a list of `Text` instances.

get_yscale()
Return the yaxis scale string: linear, log, symlog

get_yticklabels(*minor=False*)
Get the y tick labels as a list of `Text` instances

get_yticklines()
Get the ytick lines as a list of Line2D instances

get_yticks(*minor=False*)
Return the y ticks as a list of locations

grid(*b=None*, *which='major'*, *axis='both'*, ***kwargs*)
Turn the axes grids on or off.

Call signature:

```
grid(self, b=None, which='major', axis='both', **kwargs)
```

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, eg:

```
ax.grid(color='r', linestyle='-', linewidth=2)
```

Valid `Line2D` kwargs are

Property	Description
`agg_filter`	unknown
`alpha`	float (0.0 transparent through 1.0 opaque)
`animated`	[True | False]
`antialiased` or aa	[True | False]
`axes`	an `Axes` instance
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color` or c	any matplotlib color
`contains`	a callable function
`dash_capstyle`	['butt' | 'round' | 'projecting']
`dash_joinstyle`	['miter' | 'round' | 'bevel']
`dashes`	sequence of on/off ink in points
`data`	2D array (rows are x, y) or two 1D arrays
`drawstyle`	['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post']
`figure`	a `matplotlib.figure.Figure` instance
`fillstyle`	['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']
`gid`	an id string
`label`	string or anything printable with '%s' conversion.
`linestyle` or ls	[`'-'` | `'--'` | `'-.'` | `':'` | `'None'` | `' '` | `''`] and any drawstyle in combina
`linewidth` or lw	float value in points
`lod`	[True | False]
`marker`	[`7` | `4` | `5` | `6` | `'o'` | `'D'` | `'h'` | `'H'` | `'_'` | `''` | `'None'` | `None` | `' '` | `'8'` | `'p`
`markeredgecolor` or mec	any matplotlib color

Table 51.1

Property	Description
markeredgewidth or mew	float value in points
markerfacecolor or mfc	any matplotlib color
markerfacecoloralt or mfcalt	any matplotlib color
markersize or ms	float
markevery	None | integer | (startind, stride)
picker	float distance in points or callable pick function fn(artist, event)
pickradius	float distance in points
rasterized	[True | False | None]
snap	unknown
solid_capstyle	['butt' | 'round' | 'projecting']
solid_joinstyle	['miter' | 'round' | 'bevel']
transform	a matplotlib.transforms.Transform instance
url	a url string
visible	[True | False]
xdata	1D array
ydata	1D array
zorder	any number

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.

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 at 0x8761f44>, mincnt=None, marginals=False, **kwargs*)

Make a hexagonal binning plot.

Call signature:

```
hexbin(x, y, C = None, gridsize = 100, bins = None,
       xscale = 'linear', yscale = 'linear',
       cmap=None, norm=None, vmin=None, vmax=None,
       alpha=None, linewidths=None, edgecolors='none'
       reduce_C_function = np.mean, mincnt=None, marginals=True
       **kwargs)
```

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 occurences 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 *reduce_C_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*.)

x, *y* and/or *C* may be masked arrays, in which case only unmasked points will be plotted.

Optional keyword arguments:

***gridsize*: [100 | integer]** 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, gridsize can be a tuple with two elements specifying the number of hexagons in the *x*-direction and the *y*-direction.

***bins*: [*None* | 'log' | integer | sequence]** 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.

***xscale*: ['linear' | 'log']** Use a linear or log10 scale on the horizontal axis.

***scale*: ['linear' | 'log']** Use a linear or log10 scale on the vertical axis.

***mincnt*: [*None* | a positive integer]** If not *None*, only display cells with more than *mincnt* number of points in the cell

***marginals*: [*True* | *False*]** if marginals is *True*, plot the marginal density as colormapped rectagles along the bottom of the x-axis and left of the y-axis

***extent*: [*None* | scalars (left, right, bottom, top)]** The limits of the bins. The default assigns the limits based on gridsize, x, y, xscale and yscale.

Other keyword arguments controlling color mapping and normalization arguments:

***cmap*: [*None* | Colormap]** a `matplotlib.colors.Colormap` instance. If *None*, defaults to rc `image.cmap`.

***norm*: [*None* | Normalize]** `matplotlib.colors.Normalize` instance is used to scale luminance data to 0,1.

***vmin* / *vmax*: scalar** *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 *C* is used. Note if you pass a norm instance, your settings for *vmin* and *vmax* will be ignored.

alpha*: scalar between 0 and 1, or *None the alpha value for the patches

***linewidths*: [*None* | scalar]** If *None*, defaults to rc lines.linewidth. Note that this is a tuple, and if you set the linewidths argument you must set it as a sequence of floats, as required by `RegularPolyCollection`.

Other keyword arguments controlling the Collection properties:

***edgecolors*: [*None* | 'none' | mpl color | color sequence]** 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 the outlines in the default color.

If a matplotlib color arg or sequence of rgba tuples, draws the outlines in the specified color.

Here are the standard descriptions of all the `Collection` kwargs:

Property	Description
`agg_filter`	unknown
`alpha`	float or None
`animated`	[True | False]
`antialiased` or antialiaseds	Boolean or sequence of booleans
`array`	unknown
`axes`	an `Axes` instance
`clim`	a length 2 sequence of floats
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`cmap`	a colormap or registered colormap name
`color`	matplotlib color arg or sequence of rgba tuples
`colorbar`	unknown
`contains`	a callable function
`edgecolor` or edgecolors	matplotlib color arg or sequence of rgba tuples
`facecolor` or facecolors	matplotlib color arg or sequence of rgba tuples
`figure`	a `matplotlib.figure.Figure` instance
`gid`	an id string
`hatch`	['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
`label`	string or anything printable with '%s' conversion.
`linestyle` or linestyles or dashes	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
`linewidth` or lw or linewidths	float or sequence of floats
`lod`	[True | False]
`norm`	unknown
`offset_position`	unknown
`offsets`	float or sequence of floats
`paths`	unknown

Continued on next page

Table 51.12 – continued from previous page

Property	Description
picker	[None|float|boolean|callable]
pickradius	unknown
rasterized	[True | False | None]
snap	unknown
transform	Transform instance
url	a url string
urls	unknown
visible	[True | False]
zorder	any number

The return value is 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*.

Example:

hist(*x*, *bins=10*, *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.

Call signature:

```
hist(x, bins=10, 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)
```

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.

Keyword arguments:

bins: Either an integer number of bins or a sequence giving the bins. If *bins* is an integer, *bins* + 1 bin edges will be returned, consistent with numpy.histogram() for numpy version >= 1.3, and with the *new* =

True argument in earlier versions. Unequally spaced bins are supported if *bins* is a sequence.

***range*:** 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.

***normed*:** If *True*, the first element of the return tuple will be the counts normalized to form a probability density, i.e., `n/(len(x)*dbin)`. In a probability density, the integral of the histogram should be 1; you can verify that with a trapezoidal integration of the probability density function:

```
pdf, bins, patches = ax.hist(...)
print np.sum(pdf * np.diff(bins))
```

Note: 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

***weights*:** 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.

***cumulative*:** 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.

***histtype*:** **['bar' | 'barstacked' | 'step' | 'stepfilled']** The type of histogram to draw.

- 'bar' is a traditional bar-type histogram. If multiple data are given the bars are aranged 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.

***align*: ['left' | 'mid' | 'right']** 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.

***orientation*: ['horizontal' | 'vertical']** If 'horizontal', `barh()` will be used for bar-type histograms and the *bottom* kwarg will be the left edges.

***rwidth*:** The relative width of the bars as a fraction of the bin width. If *None*, automatically compute the width. Ignored if *histtype* = 'step' or 'stepfilled'.

***log*:** 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.

***color*:** Color spec or sequence of color specs, one per dataset. Default (*None*) uses the standard line color sequence.

***label*:** 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:

```
ax.hist(10+2*np.random.randn(1000), label='men')
ax.hist(12+3*np.random.randn(1000), label='women', alpha=0.5)
ax.legend()
```

***stacked*:** If *True*, multiple data are stacked on top of each other If *False* multiple data are aranged side by side if histtype is 'bar' or on top of each other if histtype is 'step'

.

kwargs are used to update the properties of the `Patch` instances returned by *hist*:

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
clip_box	a matplotlib.transforms.Bbox instance
clip_on	[True | False]
clip_path	[(Path, Transform) | Patch | None]
color	matplotlib color spec
contains	a callable function
edgecolor or ec	mpl color spec, or None for default, or 'none' for no color
facecolor or fc	mpl color spec, or None for default, or 'none' for no color
figure	a matplotlib.figure.Figure instance
fill	[True | False]
gid	an id string
hatch	['/' | '\\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
label	string or anything printable with '%s' conversion.
linestyle or ls	['solid' | 'dashed' | 'dashdot' | 'dotted']
linewidth or lw	float or None for default
lod	[True | False]
path_effects	unknown
picker	[None|float|boolean|callable]
rasterized	[True | False | None]
snap	unknown
transform	Transform instance
url	a url string
visible	[True | False]
zorder	any number

Example:

hist2d(*x*, *y*, *bins=10*, *range=None*, *normed=False*, *weights=None*, *cmin=None*, *cmax=None*, ***kwargs*)

Make a 2D histogram plot.

Call signature:

```
hist2d(x, y, bins = None, range=None, weights=None, cmin=None, cmax=None **kwargs)
```

Make a 2d histogram plot of *x* versus *y*, where *x*, *y* are 1-D sequences of the same length.

The return value is (counts, xedges, yedges, Image).

Optional keyword arguments: *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*: [*None* | array_like shape(2,2)]** 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*:[True|False]** Normalize histogram. The default value is False

***weights*: [*None* | array]** An array of values w_i weighing each sample (x_i, y_i).

cmin [[None| scalar]] 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 [[None| scalar]] 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

Remaining keyword arguments are passed directly to `pcolorfast()`.

Rendering the histogram with a logarithmic color scale is accomplished by passing a `colors.LogNorm` instance to the *norm* keyword argument.

Example:

hlines(*y*, *xmin*, *xmax*, *colors='k'*, *linestyles='solid'*, *label=''*, ***kwargs*)

Plot horizontal lines.

call signature:

```
hlines(y, xmin, xmax, colors='k', linestyles='solid', **kwargs)
```

Plot horizontal lines at each *y* from *xmin* to *xmax*.

Returns the `LineCollection` that was added.

Required arguments:

***y*:** a 1-D numpy array or iterable.

***xmin* and *xmax*:** can be scalars or `len(x)` numpy arrays. If they are scalars, then the respective values are constant, else the widths of the lines are determined by *xmin* and *xmax*.

Optional keyword arguments:

***colors*:** a line collections color argument, either a single color or a `len(y)` list of colors

***linestyles*:** ['solid' | 'dashed' | 'dashdot' | 'dotted']

Example:

hold(*b=None*)

Call signature:

```
hold(b=None)
```

Set the hold state. If *hold* is *None* (default), toggle the *hold* state. Else set the *hold* state to boolean value *b*.

Examples:

```
# 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

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*, ***kwargs*)

Display an image on the axes.

Call signature:

```
imshow(X, cmap=None, norm=None, aspect=None, interpolation=None,
       alpha=None, vmin=None, vmax=None, origin=None, extent=None,
       **kwargs)
```

Display the image in *X* to current axes. *X* may be a float array, a uint8 array or a PIL image. If *X* is an array, *X* can have the following shapes:

- MxN – luminance (grayscale, float array only)

- MxNx3 – RGB (float or uint8 array)
- MxNx4 – RGBA (float or uint8 array)

The value for each component of MxNx3 and MxNx4 float arrays should be in the range 0.0 to 1.0; MxN float arrays may be normalised.

An `matplotlib.image.AxesImage` instance is returned.

Keyword arguments:

***cmap*: [*None* | Colormap]** A `matplotlib.colors.Colormap` instance, eg. cm.jet. If *None*, default to rc `image.cmap` value.

cmap is ignored when *X* has RGB(A) information

***aspect*: [*None* | 'auto' | 'equal' | scalar]** 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:

Acceptable values are *None*, 'none', 'nearest', 'bilinear', 'bicubic', 'spline16', 'spline36', 'hanning', 'hamming', 'hermite', 'kaiser', 'quadric', 'catrom', 'gaussian', 'bessel', 'mitchell', 'sinc', 'lanczos'

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*: [*None* | Normalize]** An `matplotlib.colors.Normalize` instance; if *None*, default is `normalization()`. This scales luminance -> 0-1

norm is only used for an MxN float array.

***vmin/vmax*: [*None* | scalar]** 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 *norm* is not *None*, the settings for *vmin* and *vmax* will be ignored.

***alpha*: scalar** The alpha blending value, between 0 (transparent) and 1 (opaque) or *None*

***origin*: [*None* | 'upper' | 'lower']** 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*: [*None* | scalars (left, right, bottom, top)]** Data limits for the axes. The default assigns zero-based row, column indices to the x, y centers of the pixels.

***shape*: [*None* | scalars (columns, rows)]** For raw buffer images

***filternorm*:** 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*:** The filter radius for filters that have a radius parameter, i.e. when interpolation is one of: 'sinc', 'lanczos' or 'blackman'

Additional kwargs are `Artist` properties:

Property	Description
`agg_filter`	unknown
`alpha`	float (0.0 transparent through 1.0 opaque)
`animated`	[True | False]
`axes`	an `Axes` instance
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`contains`	a callable function
`figure`	a `matplotlib.figure.Figure` instance
`gid`	an id string
`label`	string or anything printable with '%s' conversion.
`lod`	[True | False]
`picker`	[None|float|boolean|callable]
`rasterized`	[True | False | None]
`snap`	unknown
`transform`	`Transform` instance
`url`	a url string
`visible`	[True | False]
`zorder`	any number

Example:

in_axes(*mouseevent*)
Return *True* if the given *mouseevent* (in display coords) is in the Axes

invert_xaxis()
Invert the x-axis.

invert_yaxis()

Invert the y-axis.

ishold()

return the HOLD status of the axes

legend(**args*, ***kwargs*)

Place a legend on the current axes.

Call signature:

```
legend(*args, **kwargs)
```

Places legend at location *loc*. Labels are a sequence of strings and *loc* can be a string or an integer specifying the legend location.

To make a legend with existing lines:

```
legend()
```

legend() by itself will try and build a legend using the label property of the lines/patches/collections. You can set the label of a line by doing:

```
plot(x, y, label='my data')
```

or:

```
line.set_label('my data').
```

If label is set to '_nolegend_', the item will not be shown in legend.

To automatically generate the legend from labels:

```
legend( ('label1', 'label2', 'label3') )
```

To make a legend for a list of lines and labels:

```
legend( (line1, line2, line3), ('label1', 'label2', 'label3') )
```

To make a legend at a given location, using a location argument:

```
legend( ('label1', 'label2', 'label3'), loc='upper left')
```

or:

```
legend( (line1, line2, line3),  ('label1', 'label2', 'label3'), loc=2)
```

The location codes are

Location String	Location Code
'best'	0
'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

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. For example,

loc = 'upper right', bbox_to_anchor = (0.5, 0.5)

will place the legend so that the upper right corner of the legend at the center of the axes.

The legend location can be specified in other coordinate, by using the *bbox_transform* keyword.

The loc itslef can be a 2-tuple giving x,y of the lower-left corner of the legend in axes coords (*bbox_to_anchor* is ignored).

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.

fontsize: [size in points | 'xx-small' | 'x-small' | 'small' | 'medium' | 'large' | 'x-large' | 'xx-large']

Set the font size. May be either a size string, relative to the default font size, or an absolute font size in points. This argument is only used if prop is not specified.

***numpoints*: integer** The number of points in the legend for line

***scatterpoints*: integer** The number of points in the legend for scatter plot

***scatteroffsets*: 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.

***frameon*: [*True* | *False*]** if *True*, draw a frame around the legend. The default is set by the rcParam 'legend.frameon'

***fancybox*: [*None* | *False* | *True*]** if *True*, draw a frame with a round fancybox. If *None*, use rc settings

***shadow*: [*None* | *False* | *True*]** If *True*, draw a shadow behind legend. If *None*, use rc settings.

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*)

bbox_to_anchor [an instance of BboxBase or a tuple of 2 or 4 floats] the bbox that the legend will be anchored.

bbox_transform [[an instance of Transform | *None*]] the transform for the bbox. transAxes if *None*.

title [string] the legend title

Padding and spacing between various elements use following keywords parameters. These values are measure 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.

Keyword	Description
borderpad	the fractional whitespace inside the legend border
labelspacing	the vertical space between the legend entries
handlelength	the length of the legend handles
handletextpad	the pad between the legend handle and text
borderaxespad	the pad between the axes and legend border
columnspacing	the spacing between columns

Note: Not all kinds of artist are supported by the legend command. See LINK (FIXME) for details.

Example:

See Also:

Legend guide.

`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.

loglog(**args*, ***kwargs*)

Make a plot with log scaling on both the *x* and *y* axis.

Call signature:

```
loglog(*args, **kwargs)
```

`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 kwargs are `Line2D` properties:

Property	Description
`agg_filter`	unknown
`alpha`	float (0.0 transparent through 1.0 opaque)
`animated`	[True | False]
`antialiased` or aa	[True | False]
`axes`	an `Axes` instance
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]

Table 51.1

Property	Description
color or c	any matplotlib color
contains	a callable function
dash_capstyle	['butt' | 'round' | 'projecting']
dash_joinstyle	['miter' | 'round' | 'bevel']
dashes	sequence of on/off ink in points
data	2D array (rows are x, y) or two 1D arrays
drawstyle	['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post']
figure	a matplotlib.figure.Figure instance
fillstyle	['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']
gid	an id string
label	string or anything printable with '%s' conversion.
linestyle or ls	['-' | '--' | '-.' | ':' | 'None' | ' ' | "] and any drawstyle in combina
linewidth or lw	float value in points
lod	[True | False]
marker	[7 | 4 | 5 | 6 | 'o' | 'D' | 'h' | 'H' | '_' | " | 'None' | None | ' ' | '8' | 'p
markeredgecolor or mec	any matplotlib color
markeredgewidth or mew	float value in points
markerfacecolor or mfc	any matplotlib color
markerfacecoloralt or mfcalt	any matplotlib color
markersize or ms	float
markevery	None | integer | (startind, stride)
picker	float distance in points or callable pick function fn(artist, event)
pickradius	float distance in points
rasterized	[True | False | None]
snap	unknown
solid_capstyle	['butt' | 'round' | 'projecting']
solid_joinstyle	['miter' | 'round' | 'bevel']
transform	a matplotlib.transforms.Transform instance
url	a url string
visible	[True | False]
xdata	1D array
ydata	1D array
zorder	any number

Example:

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.

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.

Argument: *Z* anything that can be interpreted as a 2-D array

kwargs all are passed to `imshow()`. `matshow()` sets defaults for *origin*, *interpolation*, and *aspect*; if you want row zero to be at the bottom instead of the top, you can set the *origin* kwarg to "lower".

Returns: an `matplotlib.image.AxesImage` instance.

minorticks_off()

Remove minor ticks from the axes.

minorticks_on()

Add autoscaling minor ticks to the axes.

name = 'rectilinear'

pcolor(**args*, ***kwargs*)

Create a pseudocolor plot of a 2-D array.

Call signatures:

```
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 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

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*).

Similarly for `meshgrid()`:

```
x = np.arange(5)
y = np.arange(3)
X, Y = meshgrid(x,y)
```

is equivalent to:

```
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:

```
C = rand( len(x), len(y))
```

then you need:

```
pcolor(X, Y, C.T)
```

or:

```
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:

Property	Description
`agg_filter`	unknown
`alpha`	float or None
`animated`	[True | False]

Continued on next page

Table 51.14 – continued from previous page

<table>
<tr><th>Property</th><th>Description</th></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 | False]</td></tr>
<tr><td>clip_path</td><td>[(Path, Transform) | Patch | None]</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>colorbar</td><td>unknown</td></tr>
<tr><td>contains</td><td>a callable function</td></tr>
<tr><td>edgecolor or edgecolors</td><td>matplotlib color arg or sequence of rgba tuples</td></tr>
<tr><td>facecolor or facecolors</td><td>matplotlib color arg or sequence of rgba tuples</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>[‘/’ | ‘\’ | ‘|’ | ‘-‘ | ‘+’ | ‘x’ | ‘o’ | ‘O’ | ‘.’ | ‘*’]</td></tr>
<tr><td>label</td><td>string or anything printable with ‘%s’ conversion.</td></tr>
<tr><td>linestyle or linestyles or dashes</td><td>[’solid’ | ‘dashed’, ‘dashdot’, ‘dotted’ | (offset, on-off-dash-seq)]</td></tr>
<tr><td>linewidth or lw or linewidths</td><td>float or sequence of floats</td></tr>
<tr><td>lod</td><td>[True | False]</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>paths</td><td>unknown</td></tr>
<tr><td>picker</td><td>[None|float|boolean|callable]</td></tr>
<tr><td>pickradius</td><td>unknown</td></tr>
<tr><td>rasterized</td><td>[True | False | None]</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 | False]</td></tr>
<tr><td>zorder</td><td>any number</td></tr>
</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.

pcolorfast(**args*, ***kwargs*)

pseudocolor plot of a 2-D array

Experimental; this is a version of pcolor that does not draw lines, that provides the fastest possible rendering with the Agg backend, and that can handle any quadrilateral grid.

Call signatures:

```
pcolor(C, **kwargs)
pcolor(xr, yr, C, **kwargs)
pcolor(x, y, C, **kwargs)
pcolor(X, Y, C, **kwargs)
```

C is the 2D array of color values corresponding to quadrilateral cells. Let (nr, nc) be its shape. C may be a masked array.

`pcolor(C, **kwargs)` is equivalent to `pcolor([0,nc], [0,nr], C, **kwargs)`

xr, *yr* specify the ranges of *x* and *y* corresponding to the rectangular region bounding *C*. If:

```
xr = [x0, x1]
```

and:

```
yr = [y0,y1]
```

then *x* goes from *x0* to *x1* as the second index of *C* goes from 0 to *nc*, etc. (*x0*, *y0*) is the outermost corner of cell (0,0), and (*x1*, *y1*) 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 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 the column index corresponds to the x-coordinate, and the row index corresponds to y; for details, see the “Grid Orientation” section below.

Optional keyword arguments:

***cmap*: [*None* | Colormap]** A `matplotlib.colors.Colormap` instance from cm. 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 are *None*, the min and max of the color array *C* is used. If you pass a norm instance, *vmin* and *vmax* will be *None*.

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 `QuadMesh` collection in the general quadrilateral case.

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>
<tr><th>Property</th><th>Description</th></tr>
<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 | False]</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 | False]</td></tr>
<tr><td>clip_path</td><td>[(Path, Transform) | Patch | None]</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>colorbar</td><td>unknown</td></tr>
<tr><td>contains</td><td>a callable function</td></tr>
<tr><td>edgecolor or edgecolors</td><td>matplotlib color arg or sequence of rgba tuples</td></tr>
<tr><td>facecolor or facecolors</td><td>matplotlib color arg or sequence of rgba tuples</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>['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']</td></tr>
<tr><td>label</td><td>string or anything printable with '%s' conversion.</td></tr>
<tr><td>linestyle or linestyles or dashes</td><td>['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]</td></tr>
<tr><td>linewidth or lw or linewidths</td><td>float or sequence of floats</td></tr>
<tr><td>lod</td><td>[True | False]</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>paths</td><td>unknown</td></tr>
<tr><td>picker</td><td>[None|float|boolean|callable]</td></tr>
<tr><td>pickradius</td><td>unknown</td></tr>
<tr><td>rasterized</td><td>[True | False | None]</td></tr>
<tr><td>snap</td><td>unknown</td></tr>
<tr><td>transform</td><td>Transform instance</td></tr>
</table>

Continued on next page

Table 51.15 – continued from previous page

Property	Description
url	a url string
urls	unknown
visible	[True | False]
zorder	any number

See Also:

`pcolor()` For an explanation of the grid orientation and the expansion of 1-D *X* and/or *Y* to 2-D arrays.

pick(**args*)

Call signature:

```
pick(mouseevent)
```

each child artist will fire a pick event if mouseevent is over the artist and the artist has picker set

pie(*x*, *explode=None*, *labels=None*, *colors=None*, *autopct=None*, *pctdistance=0.6*, *shadow=False*, *labeldistance=1.1*, *startangle=None*, *radius=None*)

Plot a pie chart.

Call signature:

```
pie(x, explode=None, labels=None,
    colors=('b', 'g', 'r', 'c', 'm', 'y', 'k', 'w'),
    autopct=None, pctdistance=0.6, shadow=False,
    labeldistance=1.1, startangle=None, radius=None)
```

Make 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.

Keyword arguments:

***explode*: [*None* | len(x) sequence]** If not *None*, is a `len(x)` array which specifies the fraction of the radius with which to offset each wedge.

***colors*: [*None* | color sequence]** A sequence of matplotlib color args through which the pie chart will cycle.

***labels*: [*None* | len(x) sequence of strings]** A sequence of strings providing the labels for each wedge

***autopct*: [*None* | format string | format function]** 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*: scalar** The ratio between the center of each pie slice and the start of the text generated by *autopct*. Ignored if *autopct* is *None*; default is 0.6.

***labeldistance*: scalar** The radial distance at which the pie labels are drawn

***shadow*: [*False* | *True*]** Draw a shadow beneath the pie.

***startangle*: [*None* | Offset angle]** If not *None*, rotates the start of the pie chart by *angle* degrees counterclockwise from the x-axis.

radius: [*None* | scalar] The radius of the pie, if *radius* is *None* it will be set to 1.

The pie chart will probably look best if the figure and axes are square. Eg.:

```
figure(figsize=(8,8))
ax = axes([0.1, 0.1, 0.8, 0.8])
```

Return value: If *autopct* is *None*, return the tuple (*patches*, *texts*):

- *patches* is a sequence of `matplotlib.patches.Wedge` instances
- *texts* is a list of the label `matplotlib.text.Text` instances.

If *autopct* is not *None*, return the tuple (*patches*, *texts*, *autotexts*), where *patches* and *texts* are as above, and *autotexts* is a list of `Text` instances for the numeric labels.

`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.

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 color specified by a 'color cycle'. To change this behavior, you can edit the axes.color_cycle rcParam. Alternatively, you can use `set_default_color_cycle()`.

The following format string characters are accepted to control the line style or marker:

character	description
`'-'`	solid line style
`'--'`	dashed line style
`'-.'`	dash-dot line style
`':'`	dotted line style
`'.'`	point marker
`','`	pixel marker
`'o'`	circle marker
`'v'`	triangle_down marker
`'^'`	triangle_up marker
`'<'`	triangle_left marker
`'>'`	triangle_right marker
`'1'`	tri_down marker
`'2'`	tri_up marker
`'3'`	tri_left marker
`'4'`	tri_right marker
`'s'`	square marker
`'p'`	pentagon marker
`'*'`	star marker
`'h'`	hexagon1 marker
`'H'`	hexagon2 marker
`'+'`	plus marker
`'x'`	x marker
`'D'`	diamond marker
`'d'`	thin_diamond marker
`'|'`	vline marker
`'_'`	hline marker

The following color abbreviations are supported:

character	color
'b'	blue
'g'	green
'r'	red
'c'	cyan
'm'	magenta
'y'	yellow
'k'	black
'w'	white

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:

```
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.:

```
plot(x1, y1, x2, y2, antialised=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 markercolor with:

```
plot(x, y, color='green', linestyle='dashed', marker='o',
     markerfacecolor='blue', markersize=12).
```

See `Line2D` for details.

The kwargs are `Line2D` properties:

Property	Description
`agg_filter`	unknown
`alpha`	float (0.0 transparent through 1.0 opaque)
`animated`	[True | False]
`antialiased` or aa	[True | False]
`axes`	an `Axes` instance
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color` or c	any matplotlib color
`contains`	a callable function
`dash_capstyle`	['butt' | 'round' | 'projecting']
`dash_joinstyle`	['miter' | 'round' | 'bevel']

Table 51.1

<table>
<tr><th>Property</th><th>Description</th></tr>
<tr><td>dashes</td><td>sequence of on/off ink in points</td></tr>
<tr><td>data</td><td>2D array (rows are x, y) or two 1D arrays</td></tr>
<tr><td>drawstyle</td><td>[‘default’ | ‘steps’ | ‘steps-pre’ | ‘steps-mid’ | ‘steps-post’]</td></tr>
<tr><td>figure</td><td>a matplotlib.figure.Figure instance</td></tr>
<tr><td>fillstyle</td><td>[’full’ | ‘left’ | ‘right’ | ‘bottom’ | ‘top’ | ‘none’]</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>['-' | '--' | '-.' | ':' | 'None' | ' ' | ''] and any drawstyle in combina</td></tr>
<tr><td>linewidth or lw</td><td>float value in points</td></tr>
<tr><td>lod</td><td>[True | False]</td></tr>
<tr><td>marker</td><td>[7 | 4 | 5 | 6 | 'o' | 'D' | 'h' | 'H' | '_' | '' | 'None' | None | ' ' | '8' | 'p</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 | integer | (startind, stride)</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 | False | None]</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>
</table>

kwargs *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*.

plot_date(*x*, *y*, *fmt='bo'*, *tz=None*, *xdate=True*, *ydate=False*, ***kwargs*)

Plot with data with dates.

Call signature:

```
plot_date(x, y, fmt='bo', tz=None, xdate=True, ydate=False, **kwargs)
```

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.

Keyword arguments:

***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*: [*True* | *False*]** If *True*, the *x*-axis will be labeled with dates.

***ydate*: [*False* | *True*]** If *True*, the *y*-axis will be labeled with dates.

Note if you are using custom date tickers and formatters, it may be necessary to set the formatters/locators after the call to `plot_date()` since `plot_date()` will set the default tick locator to `matplotlib.dates.AutoDateLocator` (if the tick locator is not already set to a `matplotlib.dates.DateLocator` instance) and the default tick formatter to `matplotlib.dates.AutoDateFormatter` (if the tick formatter is not already set to a `matplotlib.dates.DateFormatter` instance).

Valid kwargs are `Line2D` properties:

Property	Description
`agg_filter`	unknown
`alpha`	float (0.0 transparent through 1.0 opaque)
`animated`	[True | False]
`antialiased` or aa	[True | False]
`axes`	an `Axes` instance
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color` or c	any matplotlib color
`contains`	a callable function
`dash_capstyle`	['butt' | 'round' | 'projecting']
`dash_joinstyle`	['miter' | 'round' | 'bevel']
`dashes`	sequence of on/off ink in points
`data`	2D array (rows are x, y) or two 1D arrays
`drawstyle`	['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post']
`figure`	a `matplotlib.figure.Figure` instance
`fillstyle`	['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']
`gid`	an id string
`label`	string or anything printable with '%s' conversion.
`linestyle` or ls	[`'-'` | `'--'` | `'-.'` | `':'` | `'None'` | `' '` | "] and any drawstyle in combina
`linewidth` or lw	float value in points
`lod`	[True | False]
`marker`	[7 | 4 | 5 | 6 | `'o'` | `'D'` | `'h'` | `'H'` | `'_'` | " | `'None'` | None | `' '` | `'8'` | `'p`

Table 51.1

Property	Description
markeredgecolor or mec	any matplotlib color
markeredgewidth or mew	float value in points
markerfacecolor or mfc	any matplotlib color
markerfacecoloralt or mfcalt	any matplotlib color
markersize or ms	float
markevery	None | integer | (startind, stride)
picker	float distance in points or callable pick function fn(artist, event)
pickradius	float distance in points
rasterized	[True | False | None]
snap	unknown
solid_capstyle	['butt' | 'round' | 'projecting']
solid_joinstyle	['miter' | 'round' | 'bevel']
transform	a matplotlib.transforms.Transform instance
url	a url string
visible	[True | False]
xdata	1D array
ydata	1D array
zorder	any number

See Also:

dates for helper functions

date2num(), num2date() and drange() for help on creating the required floating point dates.

psd(*x*, *NFFT=256*, *Fs=2*, *Fc=0*, *detrend=<function detrend_none at 0x9bc0a3c>*, *window=<function window_hanning at 0x9bc0924>*, *noverlap=0*, *pad_to=None*, *sides='default'*, *scale_by_freq=None*, ***kwargs*)

Plot the power spectral density.

Call signature:

```
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, **kwargs)
```

The power spectral density 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 $|\mathrm{fft}(i)|^2$ of each segment *i* are averaged to compute *Pxx*, with a scaling to correct for power loss due to windowing. *Fs* is the sampling frequency.

Keyword arguments:

***NFFT*: integer** The number of data points used in each block for the FFT. Must be even; 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.

***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.

***detrend*: 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.

***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.

***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 psd (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*

***sides*: ['default' | 'onesided' | 'twosided']** Specifies which sides of the PSD 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 PSD, while 'twosided' forces two-sided.

***scale_by_freq*: boolean** 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.

Returns the tuple (*Pxx*, *freqs*).

For plotting, the power is plotted as $10\log_{10}(P_{xx})$ for decibels, though Pxx itself is returned.

References: Bendat & Piersol – Random Data: Analysis and Measurement Procedures, John Wiley & Sons (1986)

kwargs control the `Line2D` properties:

Property	Description
`agg_filter`	unknown
`alpha`	float (0.0 transparent through 1.0 opaque)
`animated`	[True | False]
`antialiased` or aa	[True | False]
`axes`	an `Axes` instance
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color` or c	any matplotlib color
`contains`	a callable function
`dash_capstyle`	['butt' | 'round' | 'projecting']
`dash_joinstyle`	['miter' | 'round' | 'bevel']
`dashes`	sequence of on/off ink in points
`data`	2D array (rows are x, y) or two 1D arrays
`drawstyle`	['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post']
`figure`	a `matplotlib.figure.Figure` instance
`fillstyle`	['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']
`gid`	an id string
`label`	string or anything printable with '%s' conversion.
`linestyle` or ls	[`'-'` | `'--'` | `'-.'` | `':'` | `'None'` | `' '` | "] and any drawstyle in combina
`linewidth` or lw	float value in points
`lod`	[True | False]
`marker`	[`7` | `4` | `5` | `6` | `'o'` | `'D'` | `'h'` | `'H'` | `'_'` | " | `'None'` | `None` | `' '` | `'8'` | `'p`
`markeredgecolor` or mec	any matplotlib color
`markeredgewidth` or mew	float value in points
`markerfacecolor` or mfc	any matplotlib color
`markerfacecoloralt` or mfcalt	any matplotlib color
`markersize` or ms	float
`markevery`	None | integer | (startind, stride)
`picker`	float distance in points or callable pick function `fn(artist, event)`
`pickradius`	float distance in points
`rasterized`	[True | False | None]
`snap`	unknown
`solid_capstyle`	['butt' | 'round' | 'projecting']
`solid_joinstyle`	['miter' | 'round' | 'bevel']

Table 51.1

Property	Description
transform	a matplotlib.transforms.Transform instance
url	a url string
visible	[True | False]
xdata	1D array
ydata	1D array
zorder	any number

Example:

quiver(**args*, ***kw*)

Plot a 2-D field of arrows.

call signatures:

```
quiver(U, V, **kw)
quiver(U, V, C, **kw)
quiver(X, Y, U, V, **kw)
quiver(X, Y, U, V, C, **kw)
```

Arguments:

***X*, *Y*:** The x and y coordinates of the arrow locations (default is tail of arrow; see *pivot* kwarg)

***U*, *V*:** Give the x and y components of the arrow vectors

***C*:** An optional array used to map colors to the arrows

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:

***units*: ['width' | 'height' | 'dots' | 'inches' | 'x' | 'y' | 'xy']** Arrow units; the arrow dimensions *except for length* are in multiples of this unit.

- 'width' or 'height': the width or height of the axes
- 'dots' or 'inches': pixels or inches, based on the figure dpi
- 'x', 'y', or 'xy': *X*, *Y*, or sqrt(X^2+Y^2) 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 the window is resized; for 'dots' or 'inches', resizing does not change the arrows.

***angles*: ['uv' | 'xy' | array]** With the default 'uv', the arrow aspect ratio is 1, so that if *U*==*V* the angle of the arrow on the plot is 45 degrees CCW from the *x*-axis. With 'xy', the arrow points from (x,y) to (x+u, y+v). Alternatively, arbitrary angles may be specified as an array of values in degrees, CCW from the *x*-axis.

***scale*: [*None* | float]** Data units per arrow length unit, e.g. m/s per plot width; a smaller scale parameter makes the arrow longer. 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*: *None*, or any of the *units* options.** For example, if *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', then the vector will be half the width 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*:** 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** Head width as multiple of shaft width, default is 3

***headlength*: scalar** Head length as multiple of shaft width, default is 5

***headaxislength*: scalar** Head length at shaft intersection, default is 4.5

***minshaft*: scalar** 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** 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' | 'middle' | 'tip']** 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]** This is a synonym for the `PolyCollection` facecolor kwarg. If *C* has been set, *color* has no effect.

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.

linewidths and edgecolors can be used to customize the arrow outlines. Additional `PolyCollection` keyword arguments:

<table>
<tr><th>Property</th><th>Description</th></tr>
<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 | False]</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 | False]</td></tr>
<tr><td>clip_path</td><td>[(Path, Transform) | Patch | None]</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>colorbar</td><td>unknown</td></tr>
<tr><td>contains</td><td>a callable function</td></tr>
<tr><td>edgecolor or edgecolors</td><td>matplotlib color arg or sequence of rgba tuples</td></tr>
<tr><td>facecolor or facecolors</td><td>matplotlib color arg or sequence of rgba tuples</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>[‘/’ | ‘\’ | ‘|’ | ‘-‘ | ‘+’ | ‘x’ | ‘o’ | ‘O’ | ‘.’ | ‘*’]</td></tr>
<tr><td>label</td><td>string or anything printable with ‘%s’ conversion.</td></tr>
<tr><td>linestyle or linestyles or dashes</td><td>[‘solid’ | ‘dashed’, ‘dashdot’, ‘dotted’ | (offset, on-off-dash-seq)]</td></tr>
<tr><td>linewidth or lw or linewidths</td><td>float or sequence of floats</td></tr>
<tr><td>lod</td><td>[True | False]</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>paths</td><td>unknown</td></tr>
<tr><td>picker</td><td>[None|float|boolean|callable]</td></tr>
<tr><td>pickradius</td><td>unknown</td></tr>
<tr><td>rasterized</td><td>[True | False | None]</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>
</table>

Continued on next page

Table 51.19 – continued from previous page

Property	Description
`urls`	unknown
`visible`	[True | False]
`zorder`	any number

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.

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)

relim()

Recompute the data limits based on current artists.

At present, `Collection` instances are not supported.

reset_position()

Make the original position the active position

scatter(*x*, *y*, *s=20*, *c='b'*, *marker='o'*, *cmap=None*, *norm=None*, *vmin=None*, *vmax=None*, *alpha=None*, *linewidths=None*, *faceted=True*, *verts=None*, ***kwargs*)

Make a scatter plot.

Call signatures:

```
scatter(x, y, s=20, c='b', marker='o', cmap=None, norm=None,
        vmin=None, vmax=None, alpha=None, linewidths=None,
        verts=None, **kwargs)
```

Make a scatter plot of *x* versus *y*, where *x*, *y* are converted to 1-D sequences which must be of the same length, *N*.

Keyword arguments:

s: size in points^2. It is a scalar or an array of the same length as *x* and *y*.

c: 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.

marker: can be one of:

marker	description
7	caretdown
4	caretleft
5	caretright
6	caretup
'o'	circle
'D'	diamond
'h'	hexagon1
'H'	hexagon2
'_'	hline

Continued on next page

Table 51.20 – continued from previous page

marker	description
”	nothing
'None'	nothing
None	nothing
' '	nothing
'8'	octagon
'p'	pentagon
','	pixel
'+'	plus
'.'	point
's'	square
'*'	star
'd'	thin_diamond
3	tickdown
0	tickleft
1	tickright
2	tickup
'1'	tri_down
'3'	tri_left
'4'	tri_right
'2'	tri_up
'v'	triangle_down
'<'	triangle_left
'>'	triangle_right
'^'	triangle_up
'|'	vline
'x'	x
'$...$'	render the string using mathtext.
verts	a list of (x, y) pairs used for Path vertices.
path	a Path instance.
(*numsides*, *style*, *angle*)	see below

The marker can also be a tuple (*numsides*, *style*, *angle*), which will create a custom, regular symbol.

***numsides*:** the number of sides

***style*:** the style of the regular symbol:

Value	Description
0	a regular polygon
1	a star-like symbol
2	an asterisk
3	a circle (*numsides* and *angle* is ignored)

angle**:** the angle of rotation of the symbol, in degrees

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.

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.

Other keyword arguments: the color mapping and normalization arguments will be used only if *c* is an array of floats.

cmap**: [*None* | Colormap]** A `matplotlib.colors.Colormap` instance or registered name. If *None*, defaults to rc `image.cmap`. *cmap* is only used if *c* is an array of floats.

norm**: [*None* | Normalize]** A `matplotlib.colors.Normalize` instance is used to scale luminance data to 0, 1. If *None*, use the default `normalize()`. *norm* is only used if *c* is an array of floats.

vmin/vmax**:** *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 *C* is used. Note if you pass a *norm* instance, your settings for *vmin* and *vmax* will be ignored.

alpha**: `0 <= scalar <= 1` or *None*** The alpha value for the patches

linewidths**: [*None* | scalar | sequence]** If *None*, defaults to (lines.linewidth,). Note that this is a tuple, and if you set the linewidths argument you must set it as a sequence of floats, as required by `RegularPolyCollection`.

Optional kwargs control the `Collection` properties; in particular:

edgecolors**:** The string 'none' to plot faces with no outlines

facecolors**:** The string 'none' to plot unfilled outlines

Here are the standard descriptions of all the `Collection` kwargs:

Property	Description
`agg_filter`	unknown
`alpha`	float or None
`animated`	[True | False]
`antialiased` or antialiaseds	Boolean or sequence of booleans
`array`	unknown
`axes`	an `Axes` instance
`clim`	a length 2 sequence of floats
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]

Continued on next page

Table 51.21 – continued from previous page

Property	Description
clip_path	[(Path, Transform) | Patch | None]
cmap	a colormap or registered colormap name
color	matplotlib color arg or sequence of rgba tuples
colorbar	unknown
contains	a callable function
edgecolor or edgecolors	matplotlib color arg or sequence of rgba tuples
facecolor or facecolors	matplotlib color arg or sequence of rgba tuples
figure	a matplotlib.figure.Figure instance
gid	an id string
hatch	['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
label	string or anything printable with '%s' conversion.
linestyle or linestyles or dashes	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
linewidth or lw or linewidths	float or sequence of floats
lod	[True | False]
norm	unknown
offset_position	unknown
offsets	float or sequence of floats
paths	unknown
picker	[None|float|boolean|callable]
pickradius	unknown
rasterized	[True | False | None]
snap	unknown
transform	Transform instance
url	a url string
urls	unknown
visible	[True | False]
zorder	any number

A Collection instance is returned.

semilogx(**args*, ***kwargs*)

Make a plot with log scaling on the x axis.

Call signature:

```
semilogx(*args, **kwargs)
```

semilogx() supports all the keyword arguments of plot() and matplotlib.axes.Axes.set_xscale().

Notable keyword arguments:

***basex*: scalar > 1** Base of the x logarithm

***subsx*: [*None* | sequence]** 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*: ['mask' | 'clip']** Non-positive values in *x* can be masked as invalid, or clipped to a very small positive number

The remaining valid kwargs are `Line2D` properties:

Property	Description
`agg_filter`	unknown
`alpha`	float (0.0 transparent through 1.0 opaque)
`animated`	[True | False]
`antialiased` or aa	[True | False]
`axes`	an `Axes` instance
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color` or c	any matplotlib color
`contains`	a callable function
`dash_capstyle`	['butt' | 'round' | 'projecting']
`dash_joinstyle`	['miter' | 'round' | 'bevel']
`dashes`	sequence of on/off ink in points
`data`	2D array (rows are x, y) or two 1D arrays
`drawstyle`	['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post']
`figure`	a `matplotlib.figure.Figure` instance
`fillstyle`	['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']
`gid`	an id string
`label`	string or anything printable with '%s' conversion.
`linestyle` or ls	[`'-'` | `'--'` | `'-.'` | `':'` | `'None'` | `' '` | "] and any drawstyle in combina
`linewidth` or lw	float value in points
`lod`	[True | False]
`marker`	[`7` | `4` | `5` | `6` | `'o'` | `'D'` | `'h'` | `'H'` | `'_'` | " | `'None'` | `None` | `' '` | `'8'` | `'p`
`markeredgecolor` or mec	any matplotlib color
`markeredgewidth` or mew	float value in points
`markerfacecolor` or mfc	any matplotlib color
`markerfacecoloralt` or mfcalt	any matplotlib color
`markersize` or ms	float
`markevery`	None | integer | (startind, stride)
`picker`	float distance in points or callable pick function `fn(artist, event)`
`pickradius`	float distance in points
`rasterized`	[True | False | None]
`snap`	unknown
`solid_capstyle`	['butt' | 'round' | 'projecting']

Table 51.2

Property	Description
solid_joinstyle	['miter' | 'round' | 'bevel']
transform	a matplotlib.transforms.Transform instance
url	a url string
visible	[True | False]
xdata	1D array
ydata	1D array
zorder	any number

See Also:

loglog() For example code and figure

semilogy(**args*, ***kwargs*)

Make a plot with log scaling on the *y* axis.

call signature:

```
semilogy(*args, **kwargs)
```

semilogy() supports all the keyword arguments of plot() and matplotlib.axes.Axes.set_yscale().

Notable keyword arguments:

***basey*: scalar > 1** Base of the *y* logarithm

***subsy*: [*None* | sequence]** 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']** Non-positive values in *y* can be masked as invalid, or clipped to a very small positive number

The remaining valid kwargs are Line2D properties:

Property	Description
agg_filter	unknown
alpha	float (0.0 transparent through 1.0 opaque)
animated	[True | False]
antialiased or aa	[True | False]
axes	an Axes instance
clip_box	a matplotlib.transforms.Bbox instance
clip_on	[True | False]

Table 51.2

Property	Description
clip_path	[(Path, Transform) | Patch | None]
color or c	any matplotlib color
contains	a callable function
dash_capstyle	['butt' | 'round' | 'projecting']
dash_joinstyle	['miter' | 'round' | 'bevel']
dashes	sequence of on/off ink in points
data	2D array (rows are x, y) or two 1D arrays
drawstyle	['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post']
figure	a matplotlib.figure.Figure instance
fillstyle	['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']
gid	an id string
label	string or anything printable with '%s' conversion.
linestyle or ls	['-' | '--' | '-.' | ':' | 'None' | ' ' | "] and any drawstyle in combina
linewidth or lw	float value in points
lod	[True | False]
marker	[7 | 4 | 5 | 6 | 'o' | 'D' | 'h' | 'H' | '_' | " | 'None' | None | ' ' | '8' | 'p
markeredgecolor or mec	any matplotlib color
markeredgewidth or mew	float value in points
markerfacecolor or mfc	any matplotlib color
markerfacecoloralt or mfcalt	any matplotlib color
markersize or ms	float
markevery	None | integer | (startind, stride)
picker	float distance in points or callable pick function fn(artist, event)
pickradius	float distance in points
rasterized	[True | False | None]
snap	unknown
solid_capstyle	['butt' | 'round' | 'projecting']
solid_joinstyle	['miter' | 'round' | 'bevel']
transform	a matplotlib.transforms.Transform instance
url	a url string
visible	[True | False]
xdata	1D array
ydata	1D array
zorder	any number

See Also:

loglog() For example code and figure

`set_adjustable`(*adjustable*)
ACCEPTS: ['box' | 'datalim' | 'box-forced']

`set_anchor`(*anchor*)
anchor

value	description
'C'	Center
'SW'	bottom left
'S'	bottom
'SE'	bottom right
'E'	right
'NE'	top right
'N'	top
'NW'	top left
'W'	left

`set_aspect`(*aspect*, *adjustable=None*, *anchor=None*)
aspect

value	description
'auto'	automatic; fill position rectangle with data
'normal'	same as 'auto'; deprecated
'equal'	same scaling from data to plot units for x and y
num	a circle will be stretched such that the height is num times the width. aspect=1 is the same as aspect='equal'.

adjustable

value	description
'box'	change physical size of axes
'datalim'	change xlim or ylim
'box-forced'	same as 'box', but axes can be shared

'box' does not allow axes sharing, as this can cause unintended side effect. For cases when sharing axes is fine, use 'box-forced'.

anchor

value	description
'C'	centered
'SW'	lower left corner
'S'	middle of bottom edge
'SE'	lower right corner
etc.	

`set_autoscale_on`(*b*)
Set whether autoscaling is applied on plot commands

accepts: [*True* | *False*]

set_autoscalex_on(*b*)
Set whether autoscaling for the x-axis is applied on plot commands

accepts: [*True* | *False*]

set_autoscaley_on(*b*)
Set whether autoscaling for the y-axis is applied on plot commands

accepts: [*True* | *False*]

set_axes_locator(*locator*)
set axes_locator

ACCEPT [a callable object which takes an axes instance and renderer and] returns a bbox.

set_axis_bgcolor(*color*)
set the axes background color

ACCEPTS: any matplotlib color - see `colors()`

set_axis_off()
turn off the axis

set_axis_on()
turn on the axis

set_axisbelow(*b*)
Set whether the axis ticks and gridlines are above or below most artists

ACCEPTS: [*True* | *False*]

set_color_cycle(*clist*)
Set the color cycle for any future plot commands on this Axes.

clist is a list of mpl color specifiers.

set_cursor_props(**args*)
Set the cursor property as:

```
ax.set_cursor_props(linewidth, color)
```

or:

```
ax.set_cursor_props((linewidth, color))
```

ACCEPTS: a (*float*, *color*) tuple

set_figure(*fig*)
Set the class:`Axes` figure

accepts a class:`Figure` instance

set_frame_on(*b*)

Set whether the axes rectangle patch is drawn

ACCEPTS: [*True* | *False*]

set_navigate(*b*)

Set whether the axes responds to navigation toolbar commands

ACCEPTS: [*True* | *False*]

set_navigate_mode(*b*)

Set the navigation toolbar button status;

> **Warning:** this is not a user-API function.

set_position(*pos*, *which='both'*)

Set the axes position with:

```
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*

value	description
'active'	to change the first
'original'	to change the second
'both'	to change both

set_rasterization_zorder(*z*)

Set zorder value below which artists will be rasterized

set_title(*label*, *fontdict=None*, ***kwargs*)

Call signature:

```
set_title(label, fontdict=None, **kwargs):
```

Set the title for the axes.

kwargs are Text properties:

Property	Description
`agg_filter`	unknown
`alpha`	float (0.0 transparent through 1.0 opaque)
`animated`	[True | False]

Table 51.24 – con

Property	Description
axes	an Axes instance
backgroundcolor	any matplotlib color
bbox	rectangle prop dict
clip_box	a matplotlib.transforms.Bbox instance
clip_on	[True | False]
clip_path	[(Path, Transform) | Patch | None]
color	any matplotlib color
contains	a callable function
family or fontfamily or fontname or name	[FONTNAME | 'serif' | 'sans-serif' | 'cursive' | 'fantasy' | 'mono
figure	a matplotlib.figure.Figure instance
fontproperties or font_properties	a matplotlib.font_manager.FontProperties instance
gid	an id string
horizontalalignment or ha	['center' | 'right' | 'left']
label	string or anything printable with '%s' conversion.
linespacing	float (multiple of font size)
lod	[True | False]
multialignment	['left' | 'right' | 'center']
path_effects	unknown
picker	[None|float|boolean|callable]
position	(x,y)
rasterized	[True | False | None]
rotation	[angle in degrees | 'vertical' | 'horizontal']
rotation_mode	unknown
size or fontsize	[size in points | 'xx-small' | 'x-small' | 'small' | 'medium' | 'large
snap	unknown
stretch or fontstretch	[a numeric value in range 0-1000 | 'ultra-condensed' | 'extra-cond
style or fontstyle	['normal' | 'italic' | 'oblique']
text	string or anything printable with '%s' conversion.
transform	Transform instance
url	a url string
variant or fontvariant	['normal' | 'small-caps']
verticalalignment or va or ma	['center' | 'top' | 'bottom' | 'baseline']
visible	[True | False]
weight or fontweight	[a numeric value in range 0-1000 | 'ultralight' | 'light' | 'normal' |
x	float
y	float
zorder	any number

ACCEPTS: str

See Also:

`text()` for information on how override and the optional args work

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.

set_xlabel(*xlabel*, *fontdict=None*, *labelpad=None*, ***kwargs*)
Call signature:

```
set_xlabel(xlabel, fontdict=None, labelpad=None, **kwargs)
```

Set the label for the xaxis.

labelpad is the spacing in points between the label and the x-axis

Valid kwargs are `Text` properties:

Property	Description
`agg_filter`	unknown
`alpha`	float (0.0 transparent through 1.0 opaque)
`animated`	[True | False]
`axes`	an `Axes` instance
`backgroundcolor`	any matplotlib color
`bbox`	rectangle prop dict
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color`	any matplotlib color
`contains`	a callable function
`family` or fontfamily or fontname or name	[FONTNAME | 'serif' | 'sans-serif' | 'cursive' | 'fantasy' | 'mono
`figure`	a `matplotlib.figure.Figure` instance
`fontproperties` or font_properties	a `matplotlib.font_manager.FontProperties` instance
`gid`	an id string
`horizontalalignment` or ha	['center' | 'right' | 'left']
`label`	string or anything printable with '%s' conversion.
`linespacing`	float (multiple of font size)
`lod`	[True | False]
`multialignment`	['left' | 'right' | 'center']
`path_effects`	unknown
`picker`	[None|float|boolean|callable]
`position`	(x,y)
`rasterized`	[True | False | None]
`rotation`	[angle in degrees | 'vertical' | 'horizontal']

Table 51.25 – con

Property	Description
rotation_mode	unknown
size or fontsize	[size in points | 'xx-small' | 'x-small' | 'small' | 'medium' | 'large
snap	unknown
stretch or fontstretch	[a numeric value in range 0-1000 | 'ultra-condensed' | 'extra-conc
style or fontstyle	['normal' | 'italic' | 'oblique']
text	string or anything printable with '%s' conversion.
transform	Transform instance
url	a url string
variant or fontvariant	['normal' | 'small-caps']
verticalalignment or va or ma	['center' | 'top' | 'bottom' | 'baseline']
visible	[True | False]
weight or fontweight	[a numeric value in range 0-1000 | 'ultralight' | 'light' | 'normal'
x	float
y	float
zorder	any number

ACCEPTS: str

See Also:

text() for information on how override and the optional args work

set_xlim(*left=None*, *right=None*, *emit=True*, *auto=False*, ***kw*)

Call signature:

```
set_xlim(self, *args, **kwargs):
```

Set the data limits for the xaxis

Examples:

```
set_xlim((left, right))
set_xlim(left, right)
set_xlim(left=1) # right unchanged
set_xlim(right=1) # left unchanged
```

Keyword arguments:

***left*: scalar** The left xlim; *xmin*, the previous name, may still be used

***right*: scalar** The right xlim; *xmax*, the previous name, may still be used

***emit*: [*True* | *False*]** Notify observers of limit change

***auto*: [*True* | *False* | *None*]** Turn *x* autoscaling on (*True*), off (*False*; default), or leave unchanged (*None*)

Note, the *left* (formerly *xmin*) value may be greater than the *right* (formerly *xmax*). For example, suppose *x* is years before present. Then one might use:

```
set_ylim(5000, 0)
```

so 5000 years ago is on the left of the plot and the present is on the right.

Returns the current xlimits as a length 2 tuple

ACCEPTS: length 2 sequence of floats

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

set_xscale(*value*, ***kwargs*)
Call signature:

```
set_xscale(value)
```

Set the scaling of the x-axis: 'linear' | 'log' | 'symlog'

ACCEPTS: ['linear' | 'log' | '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.

'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.

set_xticklabels(*labels*, *fontdict=None*, *minor=False*, ***kwargs*)

Call signature:

```
set_xticklabels(labels, fontdict=None, minor=False, **kwargs)
```

Set the xtick labels with list of strings *labels*. Return a list of axis text instances.

kwargs set the `Text` properties. Valid properties are

Property	Description
`agg_filter`	unknown
`alpha`	float (0.0 transparent through 1.0 opaque)
`animated`	[True | False]
`axes`	an `Axes` instance
`backgroundcolor`	any matplotlib color
`bbox`	rectangle prop dict
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color`	any matplotlib color
`contains`	a callable function
`family` or fontfamily or fontname or name	[FONTNAME | 'serif' | 'sans-serif' | 'cursive' | 'fantasy' | 'mono
`figure`	a `matplotlib.figure.Figure` instance
`fontproperties` or font_properties	a `matplotlib.font_manager.FontProperties` instance
`gid`	an id string
`horizontalalignment` or ha	['center' | 'right' | 'left']
`label`	string or anything printable with '%s' conversion.
`linespacing`	float (multiple of font size)
`lod`	[True | False]
`multialignment`	['left' | 'right' | 'center']
`path_effects`	unknown
`picker`	[None|float|boolean|callable]

Table 51.26 – con

Property	Description
position	(x,y)
rasterized	[True | False | None]
rotation	[angle in degrees | 'vertical' | 'horizontal']
rotation_mode	unknown
size or fontsize	[size in points | 'xx-small' | 'x-small' | 'small' | 'medium' | 'large
snap	unknown
stretch or fontstretch	[a numeric value in range 0-1000 | 'ultra-condensed' | 'extra-conc
style or fontstyle	['normal' | 'italic' | 'oblique']
text	string or anything printable with '%s' conversion.
transform	Transform instance
url	a url string
variant or fontvariant	['normal' | 'small-caps']
verticalalignment or va or ma	['center' | 'top' | 'bottom' | 'baseline']
visible	[True | False]
weight or fontweight	[a numeric value in range 0-1000 | 'ultralight' | 'light' | 'normal' |
x	float
y	float
zorder	any number

ACCEPTS: sequence of strings

set_xticks(*ticks*, *minor=False*)
Set the x ticks with list of *ticks*

ACCEPTS: sequence of floats

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.

set_ylabel(*ylabel*, *fontdict=None*, *labelpad=None*, ***kwargs*)
Call signature:

```
set_ylabel(ylabel, fontdict=None, labelpad=None, **kwargs)
```

Set the label for the yaxis

labelpad is the spacing in points between the label and the y-axis

Valid kwargs are Text properties:

Property	Description
agg_filter	unknown

Table 51.27 – con

Property	Description
alpha	float (0.0 transparent through 1.0 opaque)
animated	[True | False]
axes	an Axes instance
backgroundcolor	any matplotlib color
bbox	rectangle prop dict
clip_box	a matplotlib.transforms.Bbox instance
clip_on	[True | False]
clip_path	[(Path, Transform) | Patch | None]
color	any matplotlib color
contains	a callable function
family or fontfamily or fontname or name	[FONTNAME | 'serif' | 'sans-serif' | 'cursive' | 'fantasy' | 'mono
figure	a matplotlib.figure.Figure instance
fontproperties or font_properties	a matplotlib.font_manager.FontProperties instance
gid	an id string
horizontalalignment or ha	['center' | 'right' | 'left']
label	string or anything printable with '%s' conversion.
linespacing	float (multiple of font size)
lod	[True | False]
multialignment	['left' | 'right' | 'center']
path_effects	unknown
picker	[None|float|boolean|callable]
position	(x,y)
rasterized	[True | False | None]
rotation	[angle in degrees | 'vertical' | 'horizontal']
rotation_mode	unknown
size or fontsize	[size in points | 'xx-small' | 'x-small' | 'small' | 'medium' | 'large
snap	unknown
stretch or fontstretch	[a numeric value in range 0-1000 | 'ultra-condensed' | 'extra-cond
style or fontstyle	['normal' | 'italic' | 'oblique']
text	string or anything printable with '%s' conversion.
transform	Transform instance
url	a url string
variant or fontvariant	['normal' | 'small-caps']
verticalalignment or va or ma	['center' | 'top' | 'bottom' | 'baseline']
visible	[True | False]
weight or fontweight	[a numeric value in range 0-1000 | 'ultralight' | 'light' | 'normal'
x	float
y	float
zorder	any number

ACCEPTS: str

See Also:

`text()` for information on how override and the optional args work

set_ylim(*bottom=None*, *top=None*, *emit=True*, *auto=False*, ***kw*)
Call signature:

```
set_ylim(self, *args, **kwargs):
```

Set the data limits for the yaxis

Examples:

```
set_ylim((bottom, top))
set_ylim(bottom, top)
set_ylim(bottom=1) # top unchanged
set_ylim(top=1) # bottom unchanged
```

Keyword arguments:

***bottom*: scalar** The bottom ylim; the previous name, *ymin*, may still be used

***top*: scalar** The top ylim; the previous name, *ymax*, may still be used

***emit*: [*True* | *False*]** Notify observers of limit change

***auto*: [*True* | *False* | *None*]** Turn *y* autoscaling on (*True*), off (*False*; default), or leave unchanged (*None*)

Note, the *bottom* (formerly *ymin*) value may be greater than the *top* (formerly *ymax*). For example, suppose *y* is depth in the ocean. Then one might use:

```
set_ylim(5000, 0)
```

so 5000 m depth is at the bottom of the plot and the surface, 0 m, is at the top.

Returns the current ylimits as a length 2 tuple

ACCEPTS: length 2 sequence of floats

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

set_yscale(*value*, ***kwargs*)
Call signature:

```
set_yscale(value)
```

Set the scaling of the y-axis: 'linear' | 'log' | 'symlog'

ACCEPTS: ['linear' | 'log' | '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.

'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.

set_yticklabels(*labels*, *fontdict=None*, *minor=False*, ***kwargs*)

Call signature:

```
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

Property	Description
agg_filter	unknown
alpha	float (0.0 transparent through 1.0 opaque)
animated	[True | False]
axes	an Axes instance
backgroundcolor	any matplotlib color
bbox	rectangle prop dict
clip_box	a matplotlib.transforms.Bbox instance
clip_on	[True | False]
clip_path	[(Path, Transform) | Patch | None]
color	any matplotlib color
contains	a callable function
family or fontfamily or fontname or name	[FONTNAME | 'serif' | 'sans-serif' | 'cursive' | 'fantasy' | 'mono
figure	a matplotlib.figure.Figure instance
fontproperties or font_properties	a matplotlib.font_manager.FontProperties instance
gid	an id string
horizontalalignment or ha	['center' | 'right' | 'left']
label	string or anything printable with '%s' conversion.
linespacing	float (multiple of font size)
lod	[True | False]
multialignment	['left' | 'right' | 'center']
path_effects	unknown
picker	[None|float|boolean|callable]
position	(x,y)
rasterized	[True | False | None]
rotation	[angle in degrees | 'vertical' | 'horizontal']
rotation_mode	unknown
size or fontsize	[size in points | 'xx-small' | 'x-small' | 'small' | 'medium' | 'large
snap	unknown
stretch or fontstretch	[a numeric value in range 0-1000 | 'ultra-condensed' | 'extra-conc
style or fontstyle	['normal' | 'italic' | 'oblique']
text	string or anything printable with '%s' conversion.
transform	Transform instance
url	a url string
variant or fontvariant	['normal' | 'small-caps']
verticalalignment or va or ma	['center' | 'top' | 'bottom' | 'baseline']
visible	[True | False]
weight or fontweight	[a numeric value in range 0-1000 | 'ultralight' | 'light' | 'normal'
x	float
y	float
zorder	any number

ACCEPTS: sequence of strings

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*

specgram(*x*, *NFFT=256*, *Fs=2*, *Fc=0*, *detrend=<function detrend_none at 0x9bc0a3c>*, *window=<function window_hanning at 0x9bc0924>*, *noverlap=128*, *cmap=None*, *xextent=None*, *pad_to=None*, *sides='default'*, *scale_by_freq=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, **kwargs)
```

Compute a spectrogram of data in *x*. Data are split into *NFFT* length segments and the PSD 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*.

Keyword arguments:

***NFFT*: integer** The number of data points used in each block for the FFT. Must be even; 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.

***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.

***detrend*: 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.

***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.

***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 psd (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*

***sides*: ['default' | 'onesided' | 'twosided']** Specifies which sides of the PSD 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 PSD, while 'twosided' forces two-sided.

***scale_by_freq*: boolean** 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 128.

***Fc*: integer** The center frequency of *x* (defaults to 0), which offsets the y 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*:** The image extent along the x-axis. xextent = (xmin,xmax) The default is (0,max(bins)), where bins is the return value from `specgram()`

kwargs:

Additional kwargs are passed on to imshow which makes the specgram image

Return value is (*Pxx*, *freqs*, *bins*, *im*):

- *bins* are the time points the spectrogram is calculated over
- *freqs* is an array of frequencies
- *Pxx* is an array of shape `(len(times), len(freqs))` of power
- *im* is a `AxesImage` instance

Note: If *x* is real (i.e. non-complex), only the positive spectrum is shown. If *x* is complex, both positive and negative parts of the spectrum are shown. This can be overridden using the *sides* keyword argument.

Example:

spy(*Z*, *precision=0*, *marker=None*, *markersize=None*, *aspect='equal'*, ***kwargs*)

Plot the sparsity pattern on a 2-D array.

Call signature:

```
spy(Z, precision=0, marker=None, markersize=None,
    aspect='equal', **kwargs)
```

`spy(Z)` plots the sparsity pattern of the 2-D array *Z*.

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.

The array will be plotted as it would be printed, with the first index (row) increasing down and the second index (column) increasing to the right.

By default aspect is 'equal', so that each array element occupies a square space; set the aspect kwarg to 'auto' to allow the plot to fill the plot box, or to any scalar number to specify the aspect ratio of an array element directly.

Two plotting styles are available: image or marker. Both are available for full arrays, but only the marker style works for `scipy.sparse.spmatrix` instances.

If *marker* and *markersize* are *None*, an image will be returned and any remaining kwargs are passed to `imshow()`; else, a `Line2D` object will be returned with the value of marker determining the marker type, and any remaining kwargs passed to the `plot()` method.

If *marker* and *markersize* are *None*, useful kwargs include:

- *cmap*
- *alpha*

See Also:

`imshow()` For image options.

For controlling colors, e.g. cyan background and red marks, use:

```
cmap = mcolors.ListedColormap(['c','r'])
```

If *marker* or *markersize* is not *None*, useful kwargs include:

- *marker*
- *markersize*

- *color*

Useful values for *marker* include:

- 's' square (default)
- 'o' circle
- '.' point
- ',' pixel

See Also:

`plot()` For plotting options

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:

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.

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.

stem(*x*, *y*, *linefmt='b-'*, *markerfmt='bo'*, *basefmt='r-'*, *bottom=None*, *label=None*)
Create a stem plot.

Call signature:

```
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*.

Return value is a tuple (*markerline*, *stemlines*, *baseline*).

See Also:

This document for details.

Example:

step(*x*, *y*, **args*, ***kwargs*)

Make a step plot.

Call signature:

```
step(x, y, *args, **kwargs)
```

Additional keyword args to `step()` are the same as those for `plot()`.

x and *y* must be 1-D sequences, and it is assumed, but not checked, that *x* is uniformly increasing.

Keyword arguments:

***where*: ['pre' | 'post' | 'mid']** If 'pre', 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.

streamplot(*x*, *y*, *u*, *v*, *density=1*, *linewidth=None*, *color=None*, *cmap=None*, *norm=None*, *arrowsize=1*, *arrowstyle='-|>'*, *minlength=0.1*, *transform=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 25x25 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.

arrowstyle [str] Arrow style specification. See FancyArrowPatch.

minlength [float] Minimum length of streamline in axes coordinates.

Returns:

stream_container [StreamplotSet]

Container object with attributes lines : matplotlib.collections.LineCollection of streamlines arrows : collection of matplotlib.patches.FancyArrowPatch objects

repesenting 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.

`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. 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:

Property	Description
agg_filter	unknown
alpha	float (0.0 transparent through 1.0 opaque)
animated	[True | False]
axes	an Axes instance
clip_box	a matplotlib.transforms.Bbox instance
clip_on	[True | False]
clip_path	[(Path, Transform) | Patch | None]
contains	a callable function
figure	a matplotlib.figure.Figure instance
fontsize	a float in points
gid	an id string
label	string or anything printable with '%s' conversion.
lod	[True | False]
picker	[None|float|boolean|callable]
rasterized	[True | False | None]
snap	unknown
transform	Transform instance
url	a url string
visible	[True | False]
zorder	any number

text(*x*, *y*, *s*, *fontdict=None*, *withdash=False*, ***kwargs*)

Add text to the axes.

Call signature:

```
text(x, y, s, fontdict=None, **kwargs)
```

Add text in string *s* to axis at location *x*, *y*, data coordinates.

Keyword arguments:

***fontdict*:** A dictionary to override the default text properties. If *fontdict* is *None*, the defaults are determined by your rc parameters.

***withdash*: [*False* | *True*]** Creates a TextWithDash instance instead of a Text instance.

Individual keyword arguments can be used to override any given parameter:

```
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:

```
text(0.5, 0.5,'matplotlib',
     horizontalalignment='center',
```

```
verticalalignment='center',
transform = ax.transAxes)
```

You can put a rectangular box around the text instance (eg. to set a background color) by using the keyword *bbox*. *bbox* is a dictionary of `matplotlib.patches.Rectangle` properties. For example:

```
text(x, y, s, bbox=dict(facecolor='red', alpha=0.5))
```

Valid kwargs are `Text` properties:

Property	Description
`agg_filter`	unknown
`alpha`	float (0.0 transparent through 1.0 opaque)
`animated`	[True | False]
`axes`	an `Axes` instance
`backgroundcolor`	any matplotlib color
`bbox`	rectangle prop dict
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color`	any matplotlib color
`contains`	a callable function
`family` or fontfamily or fontname or name	[FONTNAME | 'serif' | 'sans-serif' | 'cursive' | 'fantasy' | 'mono
`figure`	a `matplotlib.figure.Figure` instance
`fontproperties` or font_properties	a `matplotlib.font_manager.FontProperties` instance
`gid`	an id string
`horizontalalignment` or ha	['center' | 'right' | 'left']
`label`	string or anything printable with '%s' conversion.
`linespacing`	float (multiple of font size)
`lod`	[True | False]
`multialignment`	['left' | 'right' | 'center']
`path_effects`	unknown
`picker`	[None|float|boolean|callable]
`position`	(x,y)
`rasterized`	[True | False | None]
`rotation`	[angle in degrees | 'vertical' | 'horizontal']
`rotation_mode`	unknown
`size` or fontsize	[size in points | 'xx-small' | 'x-small' | 'small' | 'medium' | 'large
`snap`	unknown
`stretch` or fontstretch	[a numeric value in range 0-1000 | 'ultra-condensed' | 'extra-cond
`style` or fontstyle	['normal' | 'italic' | 'oblique']
`text`	string or anything printable with '%s' conversion.

Table 51.29 – con

Property	Description
transform	Transform instance
url	a url string
variant or fontvariant	['normal' | 'small-caps']
verticalalignment or va or ma	['center' | 'top' | 'bottom' | 'baseline']
visible	[True | False]
weight or fontweight	[a numeric value in range 0-1000 | 'ultralight' | 'light' | 'normal'
x	float
y	float
zorder	any number

tick_params(*axis='both'*, ***kwargs*)

Change the appearance of ticks and tick labels.

Keyword arguments:

axis [['x' | 'y' | 'both']] Axis on which to operate; default is 'both'.

reset [[True | False]] 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']] Puts ticks inside or outside the axes.

length Tick length in points.

width Tick width in points.

color Tick color; accepts any mpl color spec.

pad Distance in points between tick and label.

labelsize Tick label font size in points or as a string (e.g. 'large').

labelcolor Tick label color; mpl color spec.

colors Changes the tick color and the label color to the same value: mpl color spec.

zorder Tick and label zorder.

bottom, top, left, right [[bool | 'on' | 'off']] controls whether to draw the respective ticks.

labelbottom, labeltop, labelleft, labelright Boolean or ['on' | 'off'], controls whether to draw the respective tick labels.

Example:

```
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.

ticklabel_format(***kwargs*)

Change the `ScalarFormatter` used by default for linear axes.

Optional keyword arguments:

Keyword	Description
style	['sci' (or 'scientific') | 'plain'] plain turns off scientific notation
scilimits	(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.
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']
useLocale	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.

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.

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 `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*

```
tricontourf(..., Z, V)
```

fill the (len(*V*)-1) regions between the values in *V*

```
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; eg 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

***nchunk*: [0 | integer]** If 0, no subdivision of the domain. Specify a positive integer to divide the domain into subdomains of roughly *nchunk* by *nchunk* points. This may never actually be advantageous, so this option may be removed. Chunking introduces artifacts at the chunk boundaries unless *antialiased* is *False*.

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:

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 `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*

```
tricontourf(..., Z, V)
```

fill the (len(*V*)-1) regions between the values in *V*

```
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; eg 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

***nchunk*: [0 | integer]** If 0, no subdivision of the domain. Specify a positive integer to divide the domain into subdomains of roughly *nchunk* by *nchunk* points. This may never actually be advantageous, so this option may be removed. Chunking introduces artifacts at the chunk boundaries unless *antialiased* is *False*.

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:

tripcolor(**args*, ***kwargs*)

Create a pseudocolor plot of an unstructured triangular grid.

The triangulation can be specified in one of two ways; either:

```
tripcolor(triangulation, ...)
```

where triangulation is a `Triangulation` object, or

```
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. *shading* of 'faceted' is deprecated; please use *edgecolors* instead.

The remaining kwargs are the same as for `pcolor()`.

Example:

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 `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 a explanation of these possibilities.

The remaining args and kwargs are the same as for `plot()`.

Example:

twinx()
Call signature:

```
ax = twinx()
```

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.

Note: For those who are 'picking' artists while using twinx, pick events are only called for the artists in the top-most axes.

twiny()
Call signature:

```
ax = twiny()
```

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.

update_datalim(*xys*, *updatex=True*, *updatey=True*)
Update the data lim bbox with seq of xy tups or equiv. 2-D array

update_datalim_bounds(*bounds*)
Update the datalim to include the given `Bbox` *bounds*

update_datalim_numerix(*x*, *y*)
Update the data lim bbox with seq of xy tups

vlines(*x*, *ymin*, *ymax*, *colors='k'*, *linestyles='solid'*, *label='*', ***kwargs*)
Plot vertical lines.

Call signature:

```
vlines(x, ymin, ymax, color='k', linestyles='solid')
```

Plot vertical lines at each *x* from *ymin* to *ymax*. *ymin* or *ymax* can be scalars or len(*x*) numpy arrays. If they are scalars, then the respective values are constant, else the heights of the lines are determined by *ymin* and *ymax*.

colors **:** A line collection's color args, either a single color or a `len(x)` list of colors

linestyles : ['solid' | 'dashed' | 'dashdot' | 'dotted']

Returns the `matplotlib.collections.LineCollection` that was added.

kwargs are `LineCollection` properties:

Property	Description
`agg_filter`	unknown
`alpha`	float or None
`animated`	[True | False]
`antialiased` or antialiaseds	Boolean or sequence of booleans
`array`	unknown
`axes`	an `Axes` instance
`clim`	a length 2 sequence of floats
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`cmap`	a colormap or registered colormap name
`color`	matplotlib color arg or sequence of rgba tuples
`colorbar`	unknown
`contains`	a callable function
`edgecolor` or edgecolors	matplotlib color arg or sequence of rgba tuples
`facecolor` or facecolors	matplotlib color arg or sequence of rgba tuples
`figure`	a `matplotlib.figure.Figure` instance
`gid`	an id string
`hatch`	['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
`label`	string or anything printable with '%s' conversion.
`linestyle` or linestyles or dashes	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
`linewidth` or lw or linewidths	float or sequence of floats
`lod`	[True | False]
`norm`	unknown
`offset_position`	unknown
`offsets`	float or sequence of floats
`paths`	unknown
`picker`	[None|float|boolean|callable]

Continued on next page

Table 51.30 – continued from previous page

Property	Description
pickradius	unknown
rasterized	[True | False | None]
segments	unknown
snap	unknown
transform	Transform instance
url	a url string
urls	unknown
verts	unknown
visible	[True | False]
zorder	any number

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.

xaxis_inverted()

Returns *True* if the x-axis is inverted.

xcorr(*x*, *y*, *normed=True*, *detrend=<function detrend_none at 0x9bc0a3c>*, *usevlines=True*, *maxlags=10*, ***kwargs*)

Plot the cross correlation between *x* and *y*.

Call signature:

```
xcorr(self, x, y, normed=True, detrend=mlab.detrend_none,
  usevlines=True, maxlags=10, **kwargs)
```

If *normed* = *True*, normalize the data by the cross correlation at 0-th lag. *x* and y are detrended by the *detrend* callable (default no normalization). *x* and *y* must be equal length.

Data are plotted as `plot(lags, c, **kwargs)`

Return value is a tuple (*lags*, *c*, *line*) 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()`.

The default *linestyle* is *None* and the default *marker* is 'o', though these can be overridden with keyword args. The cross correlation is performed with `numpy.correlate()` with *mode* = 2.

If *usevlines* is *True*:

`vlines()` rather than `plot()` is used to draw vertical lines from the origin to the xcorr. Otherwise the plotstyle is determined by the kwargs, which are `Line2D` properties.

The return value is a tuple (*lags*, *c*, *linecol*, *b*) where *linecol* is the `matplotlib.collections.LineCollection` instance and *b* is the *x*-axis.

maxlags is a positive integer detailing the number of lags to show. The default value of *None* will return all `(2*len(x)-1)` lags.

Example:

`xcorr()` is top graph, and `acorr()` is bottom graph.

`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.

`yaxis_inverted`()
Returns *True* if the y-axis is inverted.

`matplotlib.axes.`**`Subplot`**
alias of `AxesSubplot`

class `matplotlib.axes.`**`SubplotBase`**(*fig*, **args*, ***kwargs*)
Base class for subplots, which are `Axes` instances with additional methods to facilitate generating and manipulating a set of `Axes` within a figure.

fig is a `matplotlib.figure.Figure` instance.

args is the tuple (*numRows*, *numCols*, *plotNum*), where the array of subplots in the figure has dimensions *numRows*, *numCols*, and where *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, eg. from 1,1,1 to 2,2,3

`get_geometry`()
get the subplot geometry, eg 2,2,3

`get_subplotspec`()
get the SubplotSpec instance associated with the subplot

`is_first_col`()

`is_first_row`()

`is_last_col`()

`is_last_row`()

label_outer()
set the visible property on ticklabels so xticklabels are visible only if the subplot is in the last row and yticklabels are visible only if the subplot is in the first column

set_subplotspec(*subplotspec*)
set the SubplotSpec instance associated with the subplot

update_params()
update the subplot position from fig.subplotpars

matplotlib.axes.**set_default_color_cycle**(*clist*)
Change the default cycle of colors that will be used by the plot command. This must be called before creating the `Axes` to which it will apply; it will apply to all future axes.

clist is a sequence of mpl color specifiers.

See also: `set_color_cycle()`.

Note: Deprecated 2010/01/03. Set rcParams['axes.color_cycle'] directly.

---

matplotlib.axes.**subplot_class_factory**(*axes_class=None*)

CHAPTER

FIFTYTWO

AXIS

52.1 matplotlib.axis

Classes for the ticks and x and y axis

class matplotlib.axis.**Axis**(*axes*, *pickradius=15*)

Bases: matplotlib.artist.Artist

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

OFFSETTEXTPAD = 3

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.

cla()

clear the current axis

convert_units(*x*)

draw(*artist*, *renderer*, **args*, ***kwargs*)

Draw the axis lines, grid lines, tick lines and labels

get_children()

get_data_interval()

return the Interval instance for this axis data limits

get_gridlines()

Return the grid lines as a list of Line2D instance

get_label()
Return the axis label as a Text instance

get_label_text()
Get the text of the label

get_major_formatter()
Get the formatter of the major ticker

get_major_locator()
Get the locator of the major ticker

get_major_ticks(*numticks=None*)
get the tick instances; grow as necessary

get_majorticklabels()
Return a list of Text instances for the major ticklabels

get_majorticklines()
Return the major tick lines as a list of Line2D instances

get_majorticklocs()
Get the major tick locations in data coordinates as a numpy array

get_minor_formatter()
Get the formatter of the minor ticker

get_minor_locator()
Get the locator of the minor ticker

get_minor_ticks(*numticks=None*)
get the minor tick instances; grow as necessary

get_minorticklabels()
Return a list of Text instances for the minor ticklabels

get_minorticklines()
Return the minor tick lines as a list of Line2D instances

get_minorticklocs()
Get the minor tick locations in data coordinates as a numpy array

get_offset_text()
Return the axis offsetText as a Text instance

get_pickradius()
Return the depth of the axis used by the picker

get_scale()

get_smart_bounds()
get whether the axis has smart bounds

get_ticklabel_extents(*renderer*)
Get the extents of the tick labels on either side of the axes.

get_ticklabels(*minor=False*)
Return a list of Text instances for ticklabels

get_ticklines(*minor=False*)
Return the tick lines as a list of Line2D instances

get_ticklocs(*minor=False*)
Get the tick locations in data coordinates as a numpy array

get_tightbbox(*renderer*)
Return a bounding box that encloses the axis. It only accounts tick labels, axis label, and offsetText.

get_transform()

get_units()
return the units for axis

get_view_interval()
return the Interval instance for this axis view limits

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, eg,

xax.grid(color='r', linestyle='-', linewidth=2)

have_units()

iter_ticks()
Iterate through all of the major and minor ticks.

limit_range_for_scale(*vmin*, *vmax*)

pan(*numsteps*)
Pan *numsteps* (can be positive or negative)

reset_ticks()

set_clip_path(*clippath*, *transform=None*)

set_data_interval()
set the axis data limits

set_default_intervals()
set the default limits for the axis data and view interval if they are not mutated

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 coodinate 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

set_label_text(*label*, *fontdict=None*, ***kwargs*)
Sets the text value of the axis label

ACCEPTS: A string value for the label

set_major_formatter(*formatter*)
Set the formatter of the major ticker

ACCEPTS: A `Formatter` instance

set_major_locator(*locator*)
Set the locator of the major ticker

ACCEPTS: a `Locator` instance

set_minor_formatter(*formatter*)
Set the formatter of the minor ticker

ACCEPTS: A `Formatter` instance

set_minor_locator(*locator*)
Set the locator of the minor ticker

ACCEPTS: a `Locator` instance

set_pickradius(*pickradius*)
Set the depth of the axis used by the picker

ACCEPTS: a distance in points

set_scale(*value*, ***kwargs*)

set_smart_bounds(*value*)
set the axis to have smart bounds

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()`.

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

set_ticks(*ticks*, *minor=False*)
Set the locations of the tick marks from sequence ticks

ACCEPTS: sequence of floats

set_units(*u*)
set the units for axis

ACCEPTS: a units tag

set_view_interval(*vmin*, *vmax*, *ignore=False*)

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.

zoom(*direction*)
Zoom in/out on axis; if *direction* is >0 zoom in, else zoom out

class matplotlib.axis.**Tick**(*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*)
Bases: matplotlib.artist.Artist

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

apply_tickdir(*tickdir*)
Calculate self._pad and self._tickmarkers

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.

draw(*artist*, *renderer*, **args*, ***kwargs*)

get_children()

get_loc()
Return the tick location (data coords) as a scalar

get_pad()
Get the value of the tick label pad in points

get_pad_pixels()

get_view_interval()
return the view Interval instance for the axis this tick is ticking

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]

set_label(*s*)
Set the text of ticklabel

ACCEPTS: str

set_label1(*s*)
Set the text of ticklabel

ACCEPTS: str

set_label2(*s*)
Set the text of ticklabel2

ACCEPTS: str

set_pad(*val*)
Set the tick label pad in points

ACCEPTS: float

class matplotlib.axis.**Ticker**

formatter = None

locator = None

class matplotlib.axis.**XAxis**(*axes*, *pickradius=15*)
Bases: matplotlib.axis.Axis

Init the axis with the parent Axes instance

axis_name = 'x'

contains(*mouseevent*)
Test whether the mouse event occured in the x axis.

get_data_interval()
return the Interval instance for this axis data limits

get_label_position()
Return the label position (top or bottom)

get_minpos()

get_text_heights(*renderer*)
Returns the amount of space one should reserve for text above and below the axes. Returns a tuple (above, below)

get_ticks_position()
Return the ticks position (top, bottom, default or unknown)

get_view_interval()
return the Interval instance for this axis view limits

set_data_interval(*vmin*, *vmax*, *ignore=False*)
set the axis data limits

set_default_intervals()
set the default limits for the axis interval if they are not mutated

set_label_position(*position*)
Set the label position (top or bottom)

ACCEPTS: ['top' | 'bottom']

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']

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()`.

tick_bottom()

use ticks only on bottom

tick_top()

use ticks only on top

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*)

Bases: `matplotlib.axis.Tick`

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

apply_tickdir(*tickdir*)

get_view_interval()

return the Interval instance for this axis view limits

update_position(*loc*)

Set the location of tick in data coords with scalar *loc*

class matplotlib.axis.**YAxis**(*axes*, *pickradius=15*)

Bases: `matplotlib.axis.Axis`

Init the axis with the parent Axes instance

axis_name = 'y'

contains(*mouseevent*)

Test whether the mouse event occurred in the y axis.

Returns *True* | *False*

get_data_interval()
return the Interval instance for this axis data limits

get_label_position()
Return the label position (left or right)

get_minpos()

get_text_widths(*renderer*)

get_ticks_position()
Return the ticks position (left, right, both or unknown)

get_view_interval()
return the Interval instance for this axis view limits

set_data_interval(*vmin*, *vmax*, *ignore=False*)
set the axis data limits

set_default_intervals()
set the default limits for the axis interval if they are not mutated

set_label_position(*position*)
Set the label position (left or right)

ACCEPTS: ['left' | 'right']

set_offset_position(*position*)

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']

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()`.

tick_left()
use ticks only on left

tick_right()
use ticks only on right

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*)

Bases: `matplotlib.axis.Tick`

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

apply_tickdir(*tickdir*)

get_view_interval()
return the Interval instance for this axis view limits

update_position(*loc*)
Set the location of tick in data coords with scalar loc

CHAPTER

FIFTYTHREE

BACKENDS

53.1 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 suh 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.

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.`**`Cursors`**

HAND = 0

MOVE = 3

POINTER = 1

SELECT_REGION = 2

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*)

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

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 overrride

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

filetypes = {'pgf': 'LaTeX PGF Figure', 'ps': 'Postscript', 'emf': 'Enhanced Metafile', 'rgba': 'R

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.

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.

get_supported_filetypes()
Return dict of savefig file formats supported by this backend

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 (eg, 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.

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:

```
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_bmp(**args*, ***kwargs*)

print_emf(**args*, ***kwargs*)

print_eps(**args*, ***kwargs*)

print_figure(*filename*, *dpi=None*, *facecolor='w'*, *edgecolor='w'*, *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

edgecolor the edgecolor of the figure

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.

print_pdf(**args*, ***kwargs*)

print_pgf(**args*, ***kwargs*)

print_png(**args*, ***kwargs*)

print_ps(**args*, ***kwargs*)

print_raw(**args*, ***kwargs*)

print_rgb(**args*, ***kwargs*)

print_svg(**args*, ***kwargs*)

print_svgz(**args*, ***kwargs*)

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 (eg, 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 GUIs.

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.

Call signature:

```
start_event_loop_default(self,timeout=0)
```

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.

Call signature:

```
stop_event_loop_default(self)
```

switch_backends(*FigureCanvasClass*)

Instantiate an instance of FigureCanvasClass

This is used for backend switching, eg, to instantiate a FigureCanvasPS from a FigureCanvasGTK. Note, deep copying is not done, so any changes to one of the instances (eg, setting figure size or line props), will be reflected in the other

class matplotlib.backend_bases.**FigureManagerBase**(*canvas*, *num*)

Helper class for pyplot mode, wraps everything up into a neat bundle

Public attibutes:

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 (eg, a PS backend).

key_press(*event*)

Implement the default mpl key bindings defined at *Navigation Keyboard Shortcuts*

key_press_handler_id = None

The returned id from connecting the default key handler via `FigureCanvasBase.mpl_connnect()`.

To disable default key press handling:

```
manager, canvas = figure.canvas.manager, figure.canvas
canvas.mpl_disconnect(manager.key_press_handler_id)
```

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 (eg, 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**
An abstract base class that provides color, line styles, etc...

copy_properties(*gc*)
Copy properties from gc to self

dashd = {‘solid’: (None, None), ‘dashed’: (0, (6.0, 6.0)), ‘dotted’: (0, (1.0, 3.0)), ‘dashdot’: (0, (3.0, 5.0

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_gid()
Return the object identifier if one is set, None otherwise.

get_hatch()
Gets the current hatch style

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_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

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*, *isRGB=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 rgb or rgba, set `isRGB=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_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'). One may specify customized dash styles by providing a tuple of (offset, dash pairs). For example, the predefiend linestyles have following values.:

> 'dashed' : (0, (6.0, 6.0)), 'dashdot' : (0, (3.0, 5.0, 1.0, 5.0)), 'dotted' : (0, (1.0, 3.0)),

set_linewidth(*w*)
Set the linewidth in points

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

set_url(*url*)
Sets the url for links in compatible backends

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

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:

```
def on_key(event):
    print('you pressed', event.key, event.xdata, event.ydata)

cid = fig.canvas.mpl_connect('key_press_event', on_key)
```

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

`inaxes` **= None**

`lastevent` **= None**

`x` **= None**

`xdata` **= None**

`y` **= None**

`ydata` **= None**

class matplotlib.backend_bases.**MouseEvent**(*name*, *canvas*, *x*, *y*, *button=None*, *key=None*, *step=0*, *dblclick=False*, *guiEvent=None*)

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 button pressed None, 1, 2, 3, 'up', 'down' (up and down are used for scroll events)

key the key depressed when the mouse event triggered (see KeyEvent)

step number of scroll steps (positive for 'up', negative for 'down')

Example usage:

```
def on_press(event):
    print('you pressed', event.button, event.xdata, event.ydata)

cid = fig.canvas.mpl_connect('button_press_event', on_press)
```

x, y in figure coords, 0,0 = bottom, left button pressed None, 1, 2, 3, 'up', 'down'

button = None

dblclick = None

inaxes = None

step = None

x = None

xdata = None

y = None

ydata = None

class matplotlib.backend_bases.**NavigationToolbar2**(*canvas*)

Bases: 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 FigureCanvasBase.mpl_connect() for more information

They must also define

save_figure() save the current figure

set_cursor() if you want the pointer icon to change

_init_toolbar() create your toolbar widget `draw_rubberband()` **(optional)** draw the zoom to rect "rubberband" rectangle

`press()` **(optional)** whenever a mouse button is pressed, you'll be notified with the event

`release()` **(optional)** whenever a mouse button is released, you'll be notified with the event

`dynamic_update()` **(optional)** dynamically update the window while navigating

`set_message()` **(optional)** display message

`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!

back(**args*)
move back up the view lim stack

drag_pan(*event*)
the drag callback in pan/zoom mode

drag_zoom(*event*)
the drag callback in zoom mode

draw()
Redraw the canvases, update the locators

draw_rubberband(*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 whenver 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

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

toolitems = ((‘Home’, ‘Reset original view’, ‘home’, ‘home’), (‘Back’, ‘Back to previous view’, ‘bac

update()
Reset the axes stack

zoom(**args*)
Activate zoom to rect mode

exception matplotlib.backend_bases.**NonGuiException**
Bases: exceptions.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 – eg a `Line2D` pick may define different extra attributes than a `PatchCollection` pick event

Example usage:

```
line, = ax.plot(rand(100), 'o', picker=5)  # 5 points tolerance
```

```python
def on_pick(event):
    thisline = event.artist
    xdata, ydata = thisline.get_data()
    ind = event.ind
    print('on pick line:', zip(xdata[ind], ydata[ind]))

cid = fig.canvas.mpl_connect('pick_event', on_pick)
```

class `matplotlib.backend_bases.`**`RendererBase`**

An abstract base class to handle drawing/rendering operations.

The following methods *must* be implemented in the backend:

- `draw_path()`
- `draw_image()`
- `draw_text()`
- `get_text_width_height_descent()`

The following methods *should* be implemented in the backend for optimization reasons:

- `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`

`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 trianglex.

colors is a Nx3x4 array of RGBA colors for each point of the triangles.

transform is an affine transform to apply to the points.

`draw_image`(*gc*, *x*, *y*, *im*)

Draw the image instance into the current axes;

gc a GraphicsContext containing clipping information

x is the distance in pixels from the left hand side of the canvas.

y the distance from the origin. That is, if origin is upper, y is the distance from top. If origin is lower, y is the distance from bottom

im the `matplotlib._image.Image` instance

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 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_tex(*gc*, *x*, *y*, *s*, *prop*, *angle*, *ismath='TeX!'*)

draw_text(*gc*, *x*, *y*, *s*, *prop*, *angle*, *ismath=False*)
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 in display coords

s a matplotlib.text.Text instance

prop a matplotlib.font_manager.FontProperties instance

angle the rotation angle in degrees

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:

```
if 0: bbox_artist(self, renderer)
```

to if 1, and then the actual bounding box will be blotted 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.

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 coords 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 want to rescale and composite raster images. (like SVG)

option_scale_image()
override this method for renderers that support arbitrary scaling of image (most of the vector backend).

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, eg, 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 Event attributes, the following event attributes are defined:

width width of the canvas in pixels

height height of the canvas in pixels

class matplotlib.backend_bases.**ShowBase**
Bases: object

Simple base class to generate a show() callable in backends.

Subclass must override mainloop() method.

mainloop()

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:

- `_timer_start`: Contains backend-specific code for starting the timer
- `_timer_stop`: Contains backend-specific code for stopping the timer

Optional overrides:

- `_timer_set_single_shot`: Code for setting the timer to single shot operating mode, if supported by the timer object. If not, the `Timer` class itself will store the flag and the `_on_timer` method should be overridden to support such behavior.
- `_timer_set_interval`: Code for setting the interval on the timer, if there is a method for doing so on the timer object.
- `_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:

- `interval`: The time between timer events in milliseconds. Default is 1000 ms.
- `single_shot`: Boolean flag indicating whether this timer should operate as single shot (run once and then stop). Defaults to `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.

`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`.

`interval`

`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`

`start`(*interval=None*)

Start the timer object. `interval` is optional and will be used to reset the timer interval first if provided.

stop()
Stop the timer.

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_class*)

53.2 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.

53.3 matplotlib.backends.backend_qt4agg

Render to qt from agg

class matplotlib.backends.backend_qt4agg.**FigureCanvasQTAgg**(*figure*)
Bases: `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

blit(*bbox=None*)
Blit the region in bbox

draw()
Draw the figure with Agg, and queue a request for a Qt draw.

drawRectangle(*rect*)

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*)

class `matplotlib.backends.backend_qt4agg.`**`FigureManagerQTAgg`**(*canvas*, *num*)

Bases: `matplotlib.backends.backend_qt4.FigureManagerQT`

class `matplotlib.backends.backend_qt4agg.`**`NavigationToolbar2QTAgg`**(*canvas*, *parent*, *coordinates=True*)

Bases: `matplotlib.backends.backend_qt4.NavigationToolbar2QT`

coordinates: should we show the coordinates on the right?

`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.

53.4 matplotlib.backends.backend_wxagg

backend_wxagg.py

A wxPython backend for Agg. This uses the GUI widgets written by Jeremy O'Donoghue (jeremy@o-donoghue.com) and the Agg backend by John Hunter (jdhunter@ace.bsd.uchicago.edu)

Copyright (C) 2003-5 Jeremy O'Donoghue, John Hunter, Illinois Institute of Technology

License: This work is licensed under the matplotlib license(PSF compatible). A copy should be included with this source code.

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 = {'pgf': 'LaTeX PGF Figure', 'ps': 'Postscript', 'emf': 'Enhanced Metafile', 'rgba': 'R

print_figure(*filename*, **args*, ***kwargs*)

class `matplotlib.backends.backend_wxagg.`**FigureFrameWxAgg**(*num*, *fig*)
Bases: `matplotlib.backends.backend_wx.FigureFrameWx`

get_canvas(*fig*)

class `matplotlib.backends.backend_wxagg.`**NavigationToolbar2WxAgg**(*canvas*)
Bases: `matplotlib.backends.backend_wx.NavigationToolbar2Wx`

get_canvas(*frame*, *fig*)

`matplotlib.backends.backend_wxagg.`**new_figure_manager**(*num*, **args*, ***kwargs*)
Create a new figure manager instance

`matplotlib.backends.backend_wxagg.`**new_figure_manager_given_figure**(*num*, *figure*)
Create a new figure manager instance for the given figure.

53.5 matplotlib.backends.backend_pdf

A PDF matplotlib backend Author: Jouni K Seppänen <jks@iki.fi>

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*, *fillp*, *strokep*, *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*)

Bases: `object`

A multi-page PDF file.

Use like this:

```
# Initialize:
pp = PdfPages('foo.pdf')

# As many times as you like, create a figure fig, then either:
fig.savefig(pp, format='pdf') # note the format argument!
# or:
pp.savefig(fig)
```

```
# Once you are done, remember to close the object:
pp.close()
```

(In reality PdfPages is a thin wrapper around PdfFile, in order to avoid confusion when using savefig and forgetting the format argument.)

Create a new PdfPages object that will be written to the file named *filename*. The file is opened at once and any older file with the same name is overwritten.

close()
Finalize this object, making the underlying file a complete PDF file.

infodict()
Return a modifiable information dictionary object (see PDF reference section 10.2.1 'Document Information Dictionary').

savefig(*figure=None*, ***kwargs*)
Save the Figure instance *figure* to this file as a new page. If *figure* is a number, the figure instance is looked up by number, and if *figure* is None, the active figure is saved. Any other keyword arguments are passed to Figure.savefig.

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*)
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

end()
Finalize stream.

write(*data*)
Write some data on 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.

53.6 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:

```
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:
        ...
```

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 differ-

ent 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)
```

encoding

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.

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

class `matplotlib.dviread.Vf`(*filename*)

Bases: `matplotlib.dviread.Dvi`

A virtual font (*.vf file) containing subroutines for dvi files.

Usage:

```
vf = Vf(filename)
glyph = vf[code]
glyph.text, glyph.boxes, glyph.width
```

`matplotlib.dviread.find_tex_file`(*filename*, *format=None*)

Call **kpsewhich** to find a file in the texmf tree. If *format* is not None, it is used as the value for the `--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)

See Also:

Kpathsea documentation The library that **kpsewhich** is part of.

53.7 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:

```
>>> 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.
- Adobe Type 1 Font Format, Adobe Systems Incorporated, third printing, v1.1, 1993. ISBN 0-201-57044-0.

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.

`parts`

`prop`

`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.

CHAPTER

FIFTYFOUR

CBOOK

54.1 matplotlib.cbook

A collection of utility functions and classes. Many (but not all) from the Python Cookbook – hence the name cbook

class matplotlib.cbook.**Bunch**(***kwds*)

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: http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/52308
```

class matplotlib.cbook.**CallbackRegistry**(**args*)

Handle registering and disconnecting for a set of signals and callbacks:

```
>>> def oneat(x):
...     print 'eat', x
>>> def ondrink(x):
...     print 'drink', x

>>> 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
>>> callbacks.process('eat', 456)
eat 456
```

```
>>> callbacks.process('be merry', 456) # nothing will be called
>>> callbacks.disconnect(id_eat)
>>> callbacks.process('eat', 456)      # nothing will be called
```

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*

class `matplotlib.cbook.`**GetRealpathAndStat**

class `matplotlib.cbook.`**Grouper**(*init*=[ ])
Bases: `object`

This class provides a lightweight way to group arbitrary objects together into disjoint sets when a full-blown graph data structure would be overkill.

Objects can be joined using `join()`, tested for connectedness using `joined()`, and all disjoint sets can be retreived by using the object as an iterator.

The objects being joined must be hashable and weak-referenceable.

For example:

```
>>> 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)
```

```
>>> grp.join(d, e)
>>> sorted(map(tuple, grp))
[(d, e), (a, b, c)]
>>> 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.

class matplotlib.cbook.**Idle**(*func*)
Bases: matplotlib.cbook.Scheduler

Schedule callbacks when scheduler is idle

run()

waittime = 0.05

class matplotlib.cbook.**MemoryMonitor**(*nmax=20000*)

clear()

plot(*i0=0*, *isub=1*, *fig=None*)

report(*segments=4*)

xy(*i0=0*, *isub=1*)

class matplotlib.cbook.**Null**(**args*, ***kwargs*)
Null objects always and reliably "do nothing."

class matplotlib.cbook.**RingBuffer**(*size_max*)
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.`**`Scheduler`**

Bases: `threading.Thread`

Base class for timeout and idle scheduling

id = 0

idlelock = <thread.lock object at 0xf7539ba0>

`stop`()

class `matplotlib.cbook.`**`Sorter`**

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.**Timeout**(*wait*, *func*)
Bases: matplotlib.cbook.Scheduler

Schedule recurring events with a wait time in seconds

run()

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.

matplotlib.cbook.**align_iterators**(*func*, **iterables*)
This generator takes a bunch of iterables that are ordered by func It sends out ordered tuples:

```
(func(row), [rows from all iterators matching func(row)])
```

It is used by matplotlib.mlab.recs_join() to join record arrays

matplotlib.cbook.**allequal**(*seq*)
Return *True* if all elements of *seq* compare equal. If *seq* is 0 or 1 length, return *True*

matplotlib.cbook.**allpairs**(*x*)
return all possible pairs in sequence *x*

Condensed by Alex Martelli from this thread on c.l.python

`matplotlib.cbook.alltrue(seq)`
Return *True* if all elements of *seq* evaluate to *True*. If *seq* is empty, return *False*.

class `matplotlib.cbook.converter(missing='Null', missingval=None)`
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.dict_delall(d, keys)`
delete all of the *keys* from the `dict` *d*

`matplotlib.cbook.distances_along_curve(X)`
This function has been moved to matplotlib.mlab – please import it from there

`matplotlib.cbook.exception_to_str(s=None)`

`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.

`matplotlib.cbook.flatten(seq, scalarp=<function is_scalar_or_string at 0x8a82b1c>)`
Returns a generator of flattened nested containers

For example:

```
>>> 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: http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/121294 and Recipe 1.12 in cookbook

`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

`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.

If the filename ends in .gz, the file is implicitly ungzipped.

`matplotlib.cbook.get_split_ind(seq, N)`
seq is a list of words. Return the index into seq such that:

```
len(' '.join(seq[:ind])<=N
```

.

`matplotlib.cbook.is_closed_polygon(X)`
This function has been moved to matplotlib.mlab – please import it from there

`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.isvector`(*X*)
This function has been moved to matplotlib.mlab – please import it from there

`matplotlib.cbook.iterable`(*obj*)
return true if *obj* is iterable

`matplotlib.cbook.less_simple_linear_interpolation`(*x*, *y*, *xi*, *extrap=False*)
This function has been moved to matplotlib.mlab – please import it from there

`matplotlib.cbook.listFiles`(*root*, *patterns='*'*, *recurse=1*, *return_folders=0*)
Recursively list files

from Parmar and Martelli in the Python Cookbook

class `matplotlib.cbook.maxdict`(*maxsize*)
Bases: `dict`

A dictionary with a maximum size; this doesn't override all the relevant methods to contrain 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.onetrue`(*seq*)
Return *True* if one element of *seq* is *True*. It *seq* is empty, return *False*.

`matplotlib.cbook.path_length`(*X*)
This function has been moved to matplotlib.mlab – please import it from there

`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=<open file '<stdout>', mode 'w' at 0xf752a078>*, *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.`**`quad2cubic`**(*q0x*, *q0y*, *q1x*, *q1y*, *q2x*, *q2y*)
This function has been moved to matplotlib.mlab – please import it from there

`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_masked_invalid`**(*x*)

`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'*, *missing-val=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

class matplotlib.cbook.**tostr**(*missing='Null'*, *missingval=''*)
Bases: matplotlib.cbook.converter

convert to string or None

matplotlib.cbook.**unicode_safe**(*s*)

matplotlib.cbook.**unique**(*x*)
Return a list of unique elements of *x*

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:

```
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.

`matplotlib.cbook.`**`vector_lengths`**(*X*, *P=2.0*, *axis=None*)
This function has been moved to matplotlib.mlab – please import it from there

`matplotlib.cbook.`**`wrap`**(*prefix*, *text*, *cols*)
wrap *text* with *prefix* at length *cols*

CHAPTER

FIFTYFIVE

CM (COLORMAP)

55.1 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.

class `matplotlib.cm.`**`ScalarMappable`**(*norm=None*, *cmap=None*)

This is a mixin class to support scalar -> RGBA mapping. Handles normalization and colormapping

norm is an instance of `colors.Normalize` or one of its subclasses, used to map luminance to 0-1. *cmap* is a `cm` colormap instance, for example `cm.jet`

`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 callbackSM listeners to the 'changed' signal

`check_update`(*checker*)

If mappable has changed since the last check, return True; else return False

`get_array`()

Return the array

`get_clim`()

return the min, max of the color limits for image scaling

get_cmap()
return the colormap

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_colorbar(*im*, *ax*)
set the colorbar image and axes associated with mappable

set_norm(*norm*)
set the normalization instance

to_rgba(*x*, *alpha=None*, *bytes=False*)
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.

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 builtin colormaps.

If *name* is a `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 with a corresponding data dictionary in *datad*.

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 `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 `colors.LinearSegmentedColormap` initializer, and the resulting colormap is registered.

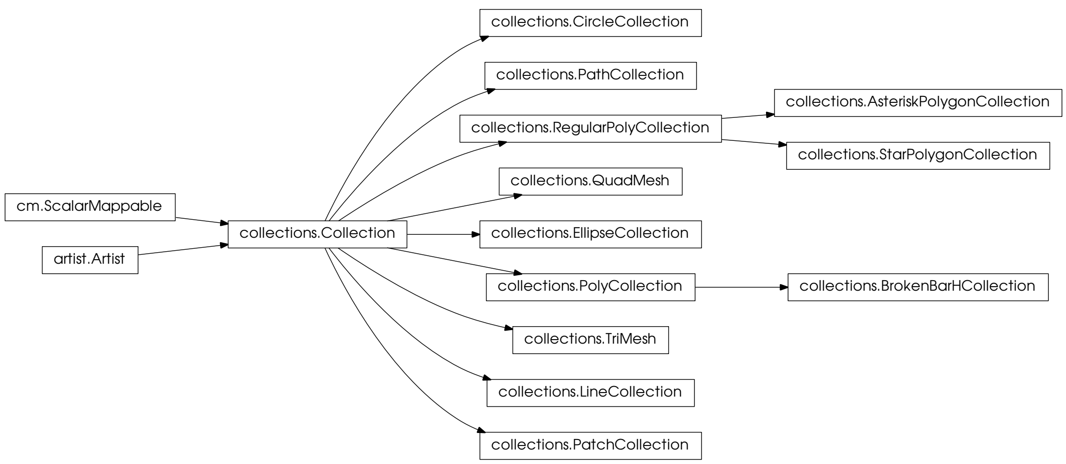
matplotlib.cm.**revcmap**(*data*)

Can only handle specification *data* in dictionary format.

CHAPTER

FIFTYSIX

COLLECTIONS



56.1 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 segemnts)

class `matplotlib.collections.`**`AsteriskPolygonCollection`**(*numsides*, *rotation=0*, *sizes=(1,)*, ***kwargs*)

Bases: `matplotlib.collections.RegularPolyCollection`

Draw a collection of regular asterisks 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:

```
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,
    )
```

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.

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.

class `matplotlib.collections.`**CircleCollection**(*sizes*, ***kwargs*)

Bases: `matplotlib.collections.Collection`

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.

draw(*artist*, *renderer*, **args*, ***kwargs*)

get_sizes()
return sizes of circles

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'*, ***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:

```
prop[i % 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

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 (ie 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

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.

draw(*artist*, *renderer*, **args*, ***kwargs*)

get_dashes()

get_datalim(*transData*)

get_edgecolor()

get_edgecolors()

get_facecolor()

get_facecolors()

get_hatch()
Return the current hatching pattern

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_paths()

get_pickradius()

get_transforms()

get_urls()

get_window_extent(*renderer*)

set_alpha(*alpha*)
Set the alpha tranparencies of the collection. *alpha* must be a float or *None*.

ACCEPTS: float or None

set_antialiased(*aa*)
Set the antialiasing state for rendering.

ACCEPTS: Boolean or sequence of booleans

set_antialiaseds(*aa*)
alias for set_antialiased

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_dashes(*ls*)
alias for set_linestyle

set_edgecolor(*c*)
Set the edgecolor(s) of the collection. *c* can be a matplotlib color arg (all patches have same color), or a sequence of rgba tuples; 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 arg or sequence of rgba tuples

set_edgecolors(*c*)
alias for set_edgecolor

set_facecolor(*c*)
Set the facecolor(s) of the collection. *c* can be a matplotlib color arg (all patches have same color), or a sequence of rgba tuples; 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 arg or sequence of rgba tuples

set_facecolors(*c*)
alias for set_facecolor

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_linestyle(*ls*)
Set the linestyle(s) for the collection.

ACCEPTS: ['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]

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_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_paths()

set_pickradius(*pr*)

set_urls(*urls*)

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 = 1

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.

draw(*artist*, *renderer*, **args*, ***kwargs*)

class `matplotlib.collections.`**LineCollection**(*segments*, *linewidths=None*, *colors=None*, *antialiaseds=None*, *linestyles='solid'*, *offsets=None*, *transOffset=None*, *norm=None*, *cmap=None*, *pickradius=5*, ***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)]
```

i.e., 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 (eg 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 *linewidths*, *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.

The use of `ScalarMappable` is optional. If the `ScalarMappable` array `_A` is not None (ie a call to `set_array()` has been made), at draw time a call to scalar mappable will be made to set the colors.

`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

`get_color`()

`get_colors`()

`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_paths`(*segments*)

`set_segments`(*segments*)

`set_verts`(*segments*)

`zorder` = 2

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 linewidths 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.rcParams` patch setting, in sequence form.

The use of `ScalarMappable` is optional. If the `ScalarMappable` matrix _A is not None (ie a call to set_array has been made), at draw time a call to scalar mappable will be made to set the face colors.

set_paths(*patches*)

class matplotlib.collections.**PathCollection**(*paths*, *sizes=None*, ***kwargs*)

Bases: `matplotlib.collections.Collection`

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
- *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.

draw(*artist*, *renderer*, **args*, ***kwargs*)

get_paths()

get_sizes()

set_paths(*paths*)

class matplotlib.collections.**PolyCollection**(*verts*, *sizes=None*, *closed=True*, ***kwargs*)

Bases: `matplotlib.collections.Collection`

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.

draw(*artist*, *renderer*, **args*, ***kwargs*)

set_paths(*verts*, *closed=True*)
This allows one to delay initialization of the vertices.

set_verts(*verts*, *closed=True*)
This allows one to delay initialization of the vertices.

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 (*meshWidth* + 1, *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 <= *m* <= *meshWidth* and 0 <= *n* <= *meshHeight*, the vertices at mesh coordinates (*m*, *n*), (*m*, *n* + 1), (*m* + 1, *n* + 1), and (*m* + 1, *n*) form one of the quadrilaterals in the mesh. There are thus (*meshWidth* * *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 x ((*meshWidth* + 1) * (*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, meshWidth), (1, 0), (1, 1), and so on.

shading may be 'flat', or 'gouraud'

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`.

draw(*artist*, *renderer*, **args*, ***kwargs*)

get_datalim(*transData*)

get_paths()

set_paths()

class matplotlib.collections.**RegularPolyCollection**(*numsides*, *rotation=0*, *sizes=(1,)*, ***kwargs*)

Bases: `matplotlib.collections.Collection`

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:

```
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,
    )
```

draw(*artist*, *renderer*, **args*, ***kwargs*)

get_numsides()

get_rotation()

get_sizes()

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:

```
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,
    )
```

class `matplotlib.collections.`**`TriMesh`**(*triangulation*, ***kwargs*)

Bases: `matplotlib.collections.Collection`

Class for the efficient drawing of a triangular mesh using Gouraud shading.

A triangular mesh is a `Triangulation` object.

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.

`draw`(*artist*, *renderer*, **args*, ***kwargs*)

`get_paths`()

`set_paths`()

CHAPTER

FIFTYSEVEN

COLORBAR

57.1 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()`.

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.

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.

class matplotlib.colorbar.**ColorbarBase**(*ax*, *cmap=None*, *norm=None*, *alpha=None*, *values=None*, *boundaries=None*, *orientation='vertical'*, *extend='neither'*, *spacing='uniform'*, *ticks=None*, *format=None*, *drawedges=False*, *filled=True*, *extendfrac=None*)

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:

```
colors.Normalize(clip=False)
```

To show the colors versus index instead of on the 0-1 scale, use:

```
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()`.

add_lines(*levels*, *colors*, *linewidths*, *erase=True*)
Draw lines on the colorbar.

colors and *linewidths* must be scalars or sequences the same length as *levels*.

Set *erase* to False to add lines without first removing any previously added lines.

config_axis()

draw_all()
Calculate any free parameters based on the current cmap and norm, and do all the drawing.

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**(*parent*, ***kw*)
Resize and reposition a parent axes, and return a child axes suitable for a colorbar:

```
cax, kw = make_axes(parent, **kw)
```

Keyword arguments may include the following (with defaults):

orientation 'vertical' or 'horizontal'

Property	Description
orientation	vertical or horizontal
fraction	0.15; fraction of original axes to use for colorbar
pad	0.05 if vertical, 0.15 if horizontal; fraction of original axes between colorbar and new image axes
shrink	1.0; fraction by which to shrink the colorbar
aspect	20; ratio of long to short dimensions
anchor	(0.0, 0.5) if vertical; (0.5, 1.0) if horizontal; the anchor point of the colorbar axes
panchor	(1.0, 0.5) if vertical; (0.5, 0.0) if horizontal; the anchor point of the colorbar parent axes

All but the first of these are stripped from the input kw set.

Returns (cax, kw), the child axes and the reduced kw dictionary.

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. Prmary differences are

- *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.:

```
cax, kw = make_axes_gridspec(parent, **kw)
```

Keyword arguments may include the following (with defaults):

orientation 'vertical' or 'horizontal'

Property	Description
orientation	vertical or horizontal
fraction	0.15; fraction of original axes to use for colorbar
pad	0.05 if vertical, 0.15 if horizontal; fraction of original axes between colorbar and new image axes
shrink	1.0; fraction by which to shrink the colorbar
aspect	20; ratio of long to short dimensions
anchor	(0.0, 0.5) if vertical; (0.5, 1.0) if horizontal; the anchor point of the colorbar axes
panchor	(1.0, 0.5) if vertical; (0.5, 0.0) if horizontal; the anchor point of the colorbar parent axes

All but the first of these are stripped from the input kw set.

Returns (cax, kw), the child axes and the reduced kw dictionary.

CHAPTER

FIFTYEIGHT

COLORS

58.1 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 a single instance, *colorConverter*, of the `ColorConverter` class providing methods for converting single color specifications or sequences of them to *RGB* or *RGBA*.

Commands which take color arguments can use several formats to specify the colors. For the basic builtin colors, you can use a single letter

- b: blue
- g: green
- r: red
- c: cyan
- m: magenta
- y: yellow
- k: black
- w: white

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'
```

or you can pass an R , G , B tuple, where each of R , G , B 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__()` .

`inverse`(*value*)

class `matplotlib.colors.`**`ColorConverter`**

Bases: `object`

Provides methods for converting color specifications to *RGB* or *RGBA*

Caching is used for more efficient conversion upon repeated calls with the same argument.

Ordinarily only the single instance instantiated in this module, *colorConverter*, is needed.

`cache` = {}

`colors` = {'c': (0.0, 0.75, 0.75), 'b': (0.0, 0.0, 1.0), 'w': (1.0, 1.0, 1.0), 'g': (0.0, 0.5, 0.0), 'y': (0.75, 0.75

`to_rgb`(*arg*)

Returns an *RGB* tuple of three floats from 0-1.

arg can be an *RGB* or *RGBA* sequence or a string in any of several forms:

1. a letter from the set 'rgbcmykw'
2. a hex color string, like '#00FFFF'
3. a standard name, like 'aqua'
4. a float, like '0.4', indicating gray on a 0-1 scale

if *arg* is *RGBA*, the *A* will simply be discarded.

to_rgba(*arg*, *alpha=None*)
Returns an *RGBA* tuple of four floats from 0-1.

For acceptable values of *arg*, see `to_rgb()`. In addition, if *arg* is "none" (case-insensitive), then (0,0,0,0) will be returned. If *arg* is an *RGBA* sequence and *alpha* is not *None*, *alpha* will replace the original *A*.

to_rgba_array(*c*, *alpha=None*)
Returns a numpy array of *RGBA* tuples.

Accepts a single mpl color spec or a sequence of specs.

Special case to handle "no color": if *c* is "none" (case-insensitive), then an empty array will be returned. Same for an empty list.

class `matplotlib.colors.`**Colormap**(*name*, *N=256*)
Bases: `object`

Base class for all scalar to rgb mappings

Important methods:

- `set_bad()`
- `set_under()`
- `set_over()`

Public class attributes: `N` : number of rgb quantization levels `name` : name of colormap

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 rgb values for a shaded relief image given a data array.

Specify the azimuth (measured clockwise from south) and altitude (measured up from the plane of the surface) of the light source in degrees.

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).

shade(*data*, *cmap*)

Take the input data array, convert to HSV values in the given colormap, then adjust those color values to given 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.

shade_rgb(*rgb*, *elevation*, *fraction=1.0*)

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.

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*)
Bases: `matplotlib.colors.Colormap`

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 < len(colors)
```

the list will be truncated at *N*. If:

```
N > len(colors)
```

the list will be extended by repetition.

class `matplotlib.colors.`**`LogNorm`**(*vmin=None*, *vmax=None*, *clip=False*)

Bases: `matplotlib.colors.Normalize`

Normalize a given value to the 0-1 range on a log scale

If *vmin* or *vmax* is not given, they are taken from the input's minimum and maximum value respectively. 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:

```
vmin==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 taken from the input's minimum and maximum value respectively. 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:

```
vmin==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`

Normalize a given value to the 0-1 range

If *vmin* or *vmax* is not given, they are taken from the input's minimum and maximum value respectively. 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:

```
vmin==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

`matplotlib.colors.`**`hex2color`**(*s*)

Take a hex string *s* and return the corresponding rgb 3-tuple Example: #efefef -> (0.93725, 0.93725, 0.93725)

`matplotlib.colors.`**`hsv_to_rgb`**(*hsv*)

convert hsv values in a numpy array to rgb values both input and output arrays have shape (M,N,3)

`matplotlib.colors.`**`is_color_like`**(*c*)

Return *True* if *c* can be converted to *RGB*

`matplotlib.colors.`**`makeMappingArray`**(*N*, *data*, *gamma=1.0*)

Create an *N* -element 1-d lookup table

data represented by a list of x,y0,y1 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 y0 represents the value of y for values of x <= to that given, and y1 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 `result[x*(N-1)]` gives the closest value for values of x between 0 and 1.

`matplotlib.colors.`**`no_norm`**
alias of `NoNorm`

`matplotlib.colors.`**`normalize`**
alias of `Normalize`

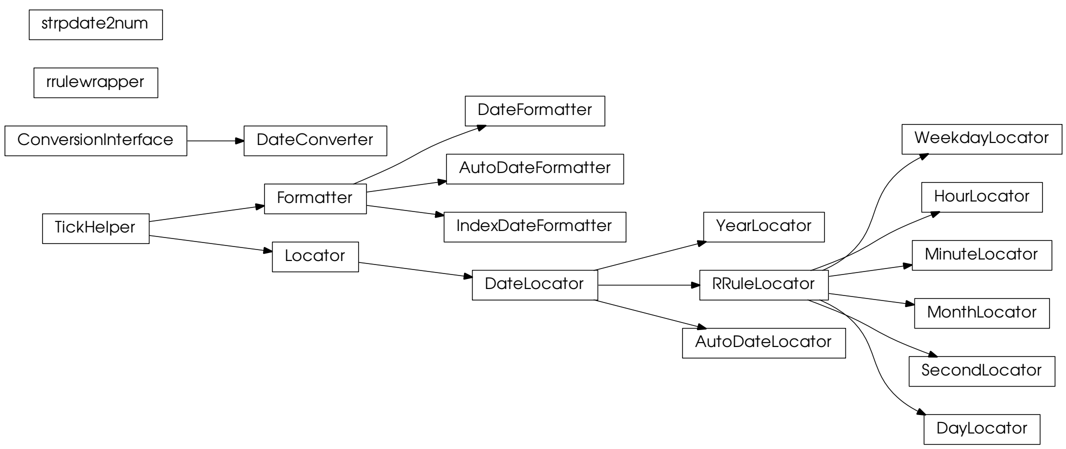
`matplotlib.colors.`**`rgb2hex`**`(`*rgb*`)`
Given an rgb or rgba sequence of 0-1 floats, return the hex string

`matplotlib.colors.`**`rgb_to_hsv`**`(`*arr*`)`
convert rgb values in a numpy array to hsv values input and output arrays should have shape (M,N,3)

CHAPTER

FIFTYNINE

DATES



59.1 matplotlib.dates

Matplotlib provides sophisticated date plotting capabilities, standing on the shoulders of python `datetime`, the add-on modules `pytz` and `dateutils`. `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 with the tz keyword argument 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.

59.1.1 Date tickers

Most of the date tickers can locate single or multiple values. For example:

```
# 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:

```
# tick on mondays every second week
loc = WeekdayLocator(byweekday=MO, interval=2)
```

The rrule locator allows completely general date ticking:

```
# 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 specifed days of the month
- `WeekdayLocator`: Locate days of the week, eg MO, TU

- MonthLocator: locate months, eg 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 `dateutils.rrule` (dateutil) which allow 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.

59.1.2 Date formatters

Here all 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.

`matplotlib.dates.`**`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.

`matplotlib.dates.`**`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.

Return value is a `datetime` instance in timezone *tz* (default to rcparams TZ value).

If *x* is a sequence, a sequence of `datetime` objects will be returned.

`matplotlib.dates.`**`drange`**`(dstart, dend, delta)`

Return a date range as float Gregorian ordinals. *dstart* and *dend* are `datetime` instances. *delta* is a `datetime.timedelta` instance.

`matplotlib.dates.`**`epoch2num`**`(e)`

Convert an epoch or sequence of epochs to the new date format, that is days since 0001.

`matplotlib.dates.`**`num2epoch`**`(d)`

Convert days since 0001 to epoch. *d* can be a number or sequence.

`matplotlib.dates.`**`mx2num`**(*mxdates*)

Convert mx `datetime` instance (or sequence of mx instances) to the new date format.

class `matplotlib.dates.`**`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 an `strftime()` format string; *tz* is the** `tzinfo` instance.

`illegal_s` = <_sre.SRE_Pattern object at 0xaa53ac0>

`set_tzinfo`(*tz*)

`strftime`(*dt*, *fmt*)

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:

```
self.scaled = {
   365.0  : '%Y',
   30.    : '%b %Y',
   1.0    : '%b %d %Y',
   1./24. : '%H:%M:%D',
   }
```

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:

```
formatter = AutoDateFormatter()
formatter.scaled[1/(24.*60.)] = '%M:%S' # only show min and sec
```

Autofmt the date labels. The default format is the one to use if none of the times in scaled match

class matplotlib.dates.**DateLocator**(*tz=None*)
Bases: matplotlib.ticker.Locator

tz is a tzinfo instance.

datalim_to_dt()

hms0d = {'byminute': 0, 'byhour': 0, 'bysecond': 0}

nonsingular(*vmin*, *vmax*)

set_tzinfo(*tz*)

viewlim_to_dt()

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*)

class matplotlib.dates.**AutoDateLocator**(*tz=None*, *minticks=5*, *maxticks=None*, *interval_multiples=False*)
Bases: matplotlib.dates.DateLocator

On autoscale, this class picks the best MultipleDateLocator 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 class: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:

```
self.intervald = {
  YEARLY  : [1, 2, 4, 5, 10],
  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]
}
```

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:

```
locator = AutoDateLocator()
locator.intervald[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.

refresh()
Refresh internal information based on current limits.

set_axis(*axis*)

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:

```
# 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.

class matplotlib.dates.**MonthLocator**(*bymonth=None*, *bymonthday=1*, *interval=1*, *tz=None*)
Bases: matplotlib.dates.RRuleLocator

Make ticks on occurances of each month month, eg 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 occurance.

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 `dateutils.rrule`.

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 occurances 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 occurances 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 occurances 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 occurances 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.rrule`(*freq*, *dtstart=None*, *interval=1*, *wkst=None*, *count=None*, *until=None*, *bysetpos=None*, *bymonth=None*, *bymonthday=None*, *byyearday=None*, *byeaster=None*, *byweekno=None*, *byweekday=None*, *byhour=None*, *byminute=None*, *bysecond=None*, *cache=False*)

Bases: `dateutil.rrule.rrulebase`

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*, *nlyearday=None*, *hour=None*, *minute=None*, *second=None*, *microsecond=None*)

The relativedelta type is based on the specification of the excelent 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's two different ways to build a relativedelta instance. The first one is passing it two date/datetime classes:

relativedelta(datetime1, datetime2)

And the other way is to use the following keyword arguments:

year, month, day, hour, minute, second, microsecond: Absolute information.

years, months, weeks, days, hours, minutes, seconds, microseconds: Relative information, may be negative.

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.

`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.

CHAPTER

SIXTY

FIGURE

60.1 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=True*, *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()`. 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, axisbg='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. Eg., if you want two axes that are otherwise identical to be added to the figure, make sure you give them unique labels:

```
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:

```
fig.add_axes(ax)
```

In all cases, the `Axes` instance will be returned.

In addition to *projection*, the following kwargs are supported:

Property	Description
`adjustable`	['box' | 'datalim' | 'box-forced']
`agg_filter`	unknown
`alpha`	float (0.0 transparent through 1.0 opaque)
`anchor`	unknown
`animated`	[True | False]
`aspect`	unknown
`autoscale_on`	unknown
`autoscalex_on`	unknown
`autoscaley_on`	unknown
`axes`	an `Axes` instance
`axes_locator`	unknown
`axis_bgcolor`	any matplotlib color - see `colors()`
`axis_off`	unknown
`axis_on`	unknown
`axisbelow`	[*True* | *False*]
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color_cycle`	unknown
`contains`	a callable function
`cursor_props`	a (*float*, *color*) tuple
`figure`	unknown
	Continued on next page

Table 60.1 – continued from previous page

Property	Description
frame_on	[*True* | *False*]
gid	an id string
label	string or anything printable with '%s' conversion.
lod	[True | False]
navigate	[*True* | *False*]
navigate_mode	unknown
picker	[None|float|boolean|callable]
position	unknown
rasterization_zorder	unknown
rasterized	[True | False | None]
snap	unknown
title	str
transform	Transform instance
url	a url string
visible	[True | False]
xbound	unknown
xlabel	str
xlim	length 2 sequence of floats
xmargin	unknown
xscale	['linear' | 'log' | 'symlog']
xticklabels	sequence of strings
xticks	sequence of floats
ybound	unknown
ylabel	str
ylim	length 2 sequence of floats
ymargin	unknown
yscale	['linear' | 'log' | 'symlog']
yticklabels	sequence of strings
yticks	sequence of floats
zorder	any number

add_axobserver(*func*)
whenever the axes state change, `func(self)` will be called

add_subplot(**args*, ***kwargs*)
Add a subplot. Examples:

```
fig.add_subplot(111)

# equivalent but more general
fig.add_subplot(1,1,1)
```

```
# add subplot with red background
fig.add_subplot(212, axisbg='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', 'polar', '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.

The following kwargs are supported:

Property	Description
`adjustable`	['box' | 'datalim' | 'box-forced']
`agg_filter`	unknown
`alpha`	float (0.0 transparent through 1.0 opaque)
`anchor`	unknown
`animated`	[True | False]
`aspect`	unknown
`autoscale_on`	unknown
`autoscalex_on`	unknown
`autoscaley_on`	unknown
`axes`	an `Axes` instance
`axes_locator`	unknown
`axis_bgcolor`	any matplotlib color - see `colors()`
`axis_off`	unknown
`axis_on`	unknown
`axisbelow`	[*True* | *False*]
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color_cycle`	unknown
`contains`	a callable function
`cursor_props`	a (*float*, *color*) tuple

Continued on next page

Table 60.2 – continued from previous page

Property	Description
figure	unknown
frame_on	[*True* | *False*]
gid	an id string
label	string or anything printable with '%s' conversion.
lod	[True | False]
navigate	[*True* | *False*]
navigate_mode	unknown
picker	[None|float|boolean|callable]
position	unknown
rasterization_zorder	unknown
rasterized	[True | False | None]
snap	unknown
title	str
transform	Transform instance
url	a url string
visible	[True | False]
xbound	unknown
xlabel	str
xlim	length 2 sequence of floats
xmargin	unknown
xscale	['linear' | 'log' | 'symlog']
xticklabels	sequence of strings
xticks	sequence of floats
ybound	unknown
ylabel	str
ylim	length 2 sequence of floats
ymargin	unknown
yscale	['linear' | 'log' | 'symlog']
yticklabels	sequence of strings
yticks	sequence of floats
zorder	any number

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 xlabels.

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*, ***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 from which space for a new colorbar axes will be stolen

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:

Property	Description
orientation	vertical or horizontal
fraction	0.15; fraction of original axes to use for colorbar
pad	0.05 if vertical, 0.15 if horizontal; fraction of original axes between colorbar and new image axes
shrink	1.0; fraction by which to shrink the colorbar
aspect	20; ratio of long to short dimensions
anchor	(0.0, 0.5) if vertical; (0.5, 1.0) if horizontal; the anchor point of the colorbar axes
panchor	(1.0, 0.5) if vertical; (0.5, 0.0) if horizontal; the anchor point of the colorbar parent axes

colorbar properties:

Property	Description
extend	['neither' | 'both' | 'min' | 'max'] If not 'neither', make pointed end(s) for out-of- range values. These are set for a given colormap using the colormap set_under and set_over methods.
extendfrac	[*None* | 'auto' | length | lengths] If set to *None*, both the minimum and maximum triangular colorbar extensions with have a length of 5% of the interior colorbar length (this is the default setting). If set to 'auto', makes the triangular colorbar extensions the same lengths as the interior boxes (when *spacing* is set to 'uniform') or the same lengths as the respective adjacent interior boxes (when *spacing* is set to 'proportional'). If a scalar, indicates the length of both the minimum and maximum triangular colorbar extensions as a fraction of the interior colorbar length. A two-element sequence of fractions may also be given, indicating the lengths of the minimum and maximum colorbar extensions respectively as a fraction of the interior colorbar length.
spacing	['uniform' | 'proportional'] Uniform spacing gives each discrete color the same space; proportional makes the space proportional to the data interval.
ticks	[None | list of ticks | Locator object] If None, ticks are determined automatically from the input.
format	[None | format string | Formatter object] If None, the `ScalarFormatter` is used. If a format string is given, e.g. '%.3f', that is used. An alternative `Formatter` object may be given instead.
drawedges	[False | True] If true, draw lines at color boundaries.

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.

Property	Description
boundaries	None or a sequence
values	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.

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:

```
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*, ***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:

Keyword	Description
xo or yo	An integer, the *x* and *y* image offset in pixels
cmap	a `matplotlib.colors.Colormap` instance, eg cm.jet. If *None*, default to the rc `image.cmap` value
norm	a `matplotlib.colors.Normalize` instance. The default is normalization(). This scales luminance -> 0-1
vmin|vmax	are 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.
alpha	the alpha blending value, default is *None*
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

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 size [0,1,0,1].

An `matplotlib.image.FigureImage` instance is returned.

Additional kwargs are Artist kwargs passed on to `FigureImage`

gca(***kwargs*)

Return 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:

Property	Description
`adjustable`	['box' | 'datalim' | 'box-forced']
`agg_filter`	unknown

Continued on next page

Table 60.3 – continued from previous page

Property	Description
alpha	float (0.0 transparent through 1.0 opaque)
anchor	unknown
animated	[True | False]
aspect	unknown
autoscale_on	unknown
autoscalex_on	unknown
autoscaley_on	unknown
axes	an Axes instance
axes_locator	unknown
axis_bgcolor	any matplotlib color - see colors()
axis_off	unknown
axis_on	unknown
axisbelow	[*True* | *False*]
clip_box	a matplotlib.transforms.Bbox instance
clip_on	[True | False]
clip_path	[(Path, Transform) | Patch | None]
color_cycle	unknown
contains	a callable function
cursor_props	a (*float*, *color*) tuple
figure	unknown
frame_on	[*True* | *False*]
gid	an id string
label	string or anything printable with '%s' conversion.
lod	[True | False]
navigate	[*True* | *False*]
navigate_mode	unknown
picker	[None|float|boolean|callable]
position	unknown
rasterization_zorder	unknown
rasterized	[True | False | None]
snap	unknown
title	str
transform	Transform instance
url	a url string
visible	[True | False]
xbound	unknown
xlabel	str
xlim	length 2 sequence of floats
xmargin	unknown
xscale	['linear' | 'log' | 'symlog']

Continued on next page

Table 60.3 – continued from previous page

Property	Description
xticklabels	sequence of strings
xticks	sequence of floats
ybound	unknown
ylabel	str
ylim	length 2 sequence of floats
ymargin	unknown
yscale	['linear' | 'log' | 'symlog']
yticklabels	sequence of strings
yticks	sequence of floats
zorder	any number

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()

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*)
Call signature:

```
ginput(self, 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 overriden 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*)
Set the hold state. If hold is None (default), toggle the hold state. Else set the hold state to boolean value b.

Eg:

```
hold()      # toggle hold
hold(True)  # hold is on
hold(False) # hold is off
```

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:

```
legend( (line1, line2, line3),
        ('label1', 'label2', 'label3'),
        'upper right')
```

The *loc* location codes are:

```
'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

***scatteroffsets*: 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.

***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.

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.

Keyword	Description
borderpad	the fractional whitespace inside the legend border
labelspacing	the vertical space between the legend entries
handlelength	the length of the legend handles
handletextpad	the pad between the legend handle and text
borderaxespad	the pad between the axes and legend border
columnspacing	the spacing between columns

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:

```
savefig(fname, dpi=None, facecolor='w', edgecolor='w',
        orientation='portrait', papertype=None, format=None,
        transparent=False, bbox_inches=None, pad_inches=0.1)
```

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`]** The resolution in dots per inch. If *None* it will default to the value `savefig.dpi` in the matplotlibrc file.

***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.

***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 the 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*)
Set the height of the figure in inches

ACCEPTS: float

`set_figwidth`(*val*)
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*, *forward=False*)
Set the figure size in inches

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; eg you can resize the figure window from the shell

ACCEPTS: a w,h tuple with w,h in inches

set_tight_layout(*tight*)
Set whether `tight_layout()` is used upon drawing. If None, the rcParams['figure.autolayout'] value will be set.

ACCEPTS: [True | False | 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

A `matplotlib.text.Text` instance is returned.

Example:

```
fig.suptitle('this is the figure title', fontsize=12)
```

text(*x*, *y*, *s*, **args*, ***kwargs*)

Add text to figure.

Call signature:

```
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:

Property	Description
`agg_filter`	unknown
`alpha`	float (0.0 transparent through 1.0 opaque)
`animated`	[True | False]
`axes`	an `Axes` instance
`backgroundcolor`	any matplotlib color
`bbox`	rectangle prop dict
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color`	any matplotlib color
`contains`	a callable function
`family` or fontfamily or fontname or name	[FONTNAME | 'serif' | 'sans-serif' | 'cursive' | 'fantasy' | 'mono
`figure`	a `matplotlib.figure.Figure` instance
`fontproperties` or font_properties	a `matplotlib.font_manager.FontProperties` instance
`gid`	an id string
`horizontalalignment` or ha	['center' | 'right' | 'left']
`label`	string or anything printable with '%s' conversion.
`linespacing`	float (multiple of font size)
`lod`	[True | False]
`multialignment`	['left' | 'right' | 'center']
`path_effects`	unknown
`picker`	[None|float|boolean|callable]
`position`	(x,y)
`rasterized`	[True | False | None]
`rotation`	[angle in degrees | 'vertical' | 'horizontal']
`rotation_mode`	unknown
`size` or fontsize	[size in points | 'xx-small' | 'x-small' | 'small' | 'medium' | 'large
`snap`	unknown
`stretch` or fontstretch	[a numeric value in range 0-1000 | 'ultra-condensed' | 'extra-conc

Table 60.4 – cont

Property	Description
style or fontstyle	['normal' | 'italic' | 'oblique']
text	string or anything printable with '%s' conversion.
transform	Transform instance
url	a url string
variant or fontvariant	['normal' | 'small-caps']
verticalalignment or va or ma	['center' | 'top' | 'bottom' | 'baseline']
visible	[True | False]
weight or fontweight	[a numeric value in range 0-1000 | 'ultralight' | 'light' | 'normal' |
x	float
y	float
zorder	any number

tight_layout(*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 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*)

Call signature:

```
waitforbuttonpress(self, timeout=-1)
```

Blocking call to interact with the figure.

This will return True is 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.

class matplotlib.figure.**SubplotParams**(*left=None*, *bottom=None*, *right=None*, *top=None*, *wspace=None*, *hspace=None*)

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

hspace [0.2] The amount of height reserved for white space between subplots

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

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, eg

Example usage:

```
# 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

CHAPTER

SIXTYONE

FONT_MANAGER

61.1 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'*)

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 Vera 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 "Bitstream Vera 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.

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 anywhere in the list returns 0.0.

A match by generic font name will return 0.1.

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'.
- stretch: A numeric value in the range 0-1000 or one of 'ultra-condensed', 'extra-condensed', 'condensed', 'semi-condensed', 'normal', 'semi-expanded', 'expanded', 'extra-expanded' or 'ultra-expanded'
- weight: A numeric value in the range 0-1000 or one of 'ultralight', 'light', 'normal', 'regular', 'book', 'medium', 'roman', 'semibold', 'demibold', 'demi', 'bold', 'heavy', 'extra bold', 'black'
- size: Either an relative value of 'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large' or an absolute font size, e.g. 12

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.

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.

get_name()
Return the name of the font that best matches the font properties.

get_size()
Return the font size.

get_size_in_points()

get_slant()
Return the font style. Values are: 'normal', 'italic' or 'oblique'.

get_stretch()
Return the font stretch or width. Options are: 'ultra-condensed', 'extra-condensed', 'condensed', 'semi-condensed', 'normal', 'semi-expanded', 'expanded', 'extra-expanded', 'ultra-expanded'.

get_style()
Return the font style. Values are: 'normal', 'italic' or 'oblique'.

get_variant()
Return the font variant. Values are: 'normal' or 'small-caps'.

get_weight()
Set the font weight. Options are: A numeric value in the range 0-1000 or one of 'light', 'normal', 'regular', 'book', 'medium', 'roman', 'semibold', 'demibold', 'demi', 'bold', 'heavy', 'extra bold', 'black'

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', or a real font name.

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', or a real font name.

set_size(*size*)
Set the font size. Either an relative value of 'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large' or an absolute font size, e.g. 12.

set_slant(*style*)
Set the font style. Values are: 'normal', 'italic' or 'oblique'.

set_stretch(*stretch*)
Set the font stretch or width. Options are: 'ultra-condensed', 'extra-condensed', 'condensed', 'semi-condensed', 'normal', 'semi-expanded', 'expanded', 'extra-expanded' or 'ultra-expanded', or a numeric value in the range 0-1000.

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*)
Set the font weight. May be either a numeric value in the range 0-1000 or one of 'ultralight', 'light', 'normal', 'regular', 'book', 'medium', 'roman', 'semibold', 'demibold', 'demi', 'bold', 'heavy', 'extra bold', 'black'

`matplotlib.font_manager.OSXInstalledFonts`(*directories=None*, *fontext='ttf'*)
Get list of font files on OS X - ignores font suffix by default.

`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'*)
Grab a list of all the fonts that are being tracked by fontconfig by making a system call to `fc-list`. This is an easy way to grab all of the fonts the user wants to be made available to applications, without needing knowing where all of them reside.

`matplotlib.font_manager.get_fontext_synonyms`(*fontext*)
Return a list of file extensions 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'.

61.2 matplotlib.fontconfig_pattern

A module for parsing and generating fontconfig patterns.

See the fontconfig pattern specification for more information.

class `matplotlib.fontconfig_pattern.`**`FontconfigPatternParser`**
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`**()
sub(repl, string[, count = 0]) –> newstring Return the string obtained by replacing the leftmost non-overlapping occurrences of pattern in string by the replacement repl.

`matplotlib.fontconfig_pattern.`**`family_unescape`**()
sub(repl, string[, count = 0]) –> newstring 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`**()
sub(repl, string[, count = 0]) –> newstring Return the string obtained by replacing the leftmost non-overlapping occurrences of pattern in string by the replacement repl.

`matplotlib.fontconfig_pattern.value_unescape()`

sub(repl, string[, count = 0]) –> newstring Return the string obtained by replacing the leftmost non-overlapping occurrences of pattern in string by the replacement repl.

CHAPTER

SIXTYTWO

GRIDSPEC

62.1 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*.

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 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).

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, eg 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 SuplotSpec instance.

set_height_ratios(*height_ratios*)

set_width_ratios(*width_ratios*)

class `matplotlib.gridspec.`**GridSpecFromSubplotSpec**(*nrows*, *ncols*, *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 stars from 0.

get_geometry()
get the subplot geometry, eg 2,2,3. Unlike SuplorParams, 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

CHAPTER

SIXTYTHREE

LEGEND

63.1 matplotlib.legend

The legend module defines the Legend class, which is responsible for drawing legends associated with axes and/or figures.

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 (See *Legend guide* for more information).

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`()

class `matplotlib.legend.`**`Legend`**(*parent*, *handles*, *labels*, *loc=None*, *numpoints=None*, *markerscale=None*, *scatterpoints=3*, *scatteryoffsets=None*, *prop=None*, *fontsize=None*, *pad=None*, *labelsep=None*, *handlelen=None*, *handletextsep=None*, *axespad=None*, *borderpad=None*, *labelspacing=None*, *handlelength=None*, *handleheight=None*, *handletextpad=None*, *borderaxespad=None*, *columnspacing=None*, *ncol=1*, *mode=None*, *fancybox=None*, *shadow=None*, *title=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 axis 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 noramilzed 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:

Keyword	Description
loc	a location code
prop	the font property
fontsize	the font size (used only if prop is not specified)
markerscale	the relative size of legend markers vs. original
numpoints	the number of points in the legend for line
scatterpoints	the number of points in the legend for scatter plot
scatteryoffsets	a list of yoffsets for scatter symbols in legend
frameon	if True, draw a frame around the legend. If None, use rc
fancybox	if True, draw a frame with a round fancybox. If None, use rc
shadow	if True, draw a shadow behind legend
ncol	number of columns
borderpad	the fractional whitespace inside the legend border
labelspacing	the vertical space between the legend entries
handlelength	the length of the legend handles
handleheight	the length of the legend handles
handletextpad	the pad between the legend handle and text
borderaxespad	the pad between the axes and legend border
columnspacing	the spacing between columns
title	the legend title
bbox_to_anchor	the bbox that the legend will be anchored.
bbox_transform	the transform for the bbox. transAxes if None.

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.

codes = {'right': 5, 'center left': 6, 'upper right': 1, 'lower right': 4, 'best': 0, 'center': 10, 'lower le

contains(*event*)

draggable(*state=None*, *use_blit=False*, *update='loc'*)

Set the draggable state – if state is

- None : toggle the current state
- True : turn draggable on
- False : turn draggable off

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* paramter of the legend is changed. If "bbox", the *bbox_to_anchor* parameter is changed.

draw(*artist*, *renderer*, **args*, ***kwargs*)
Draw everything that belongs to the legend

draw_frame(*b*)
b is a boolean. Set draw frame to b

get_bbox_to_anchor()
return the bbox that the legend will be anchored

get_children()
return a list of child artists

classmethod get_default_handler_map()
A class method that returns the default handler map.

get_frame()
return the Rectangle instance used to frame the legend

get_frame_on()
Get whether the legend box patch is drawn

static get_legend_handler(*legend_handler_map*, *orig_handle*)
return a legend handler from *legend_handler_map* that corresponds to *orig_handler*.

legend_handler_map should be a dictionary object (that is returned by the get_legend_handler_map method).

It first checks if the *orig_handle* itself is a key in the *legend_hanler_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.

get_legend_handler_map()
return the handler map.

get_lines()
return a list of lines.Line2D instances in the legend

get_patches()
return a list of patch instances in the legend

get_texts()
return a list of text.Text instance in the legend

get_title()
return Text instance for the legend title

get_window_extent(**args*, ***kwargs*)
return a extent of the 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.

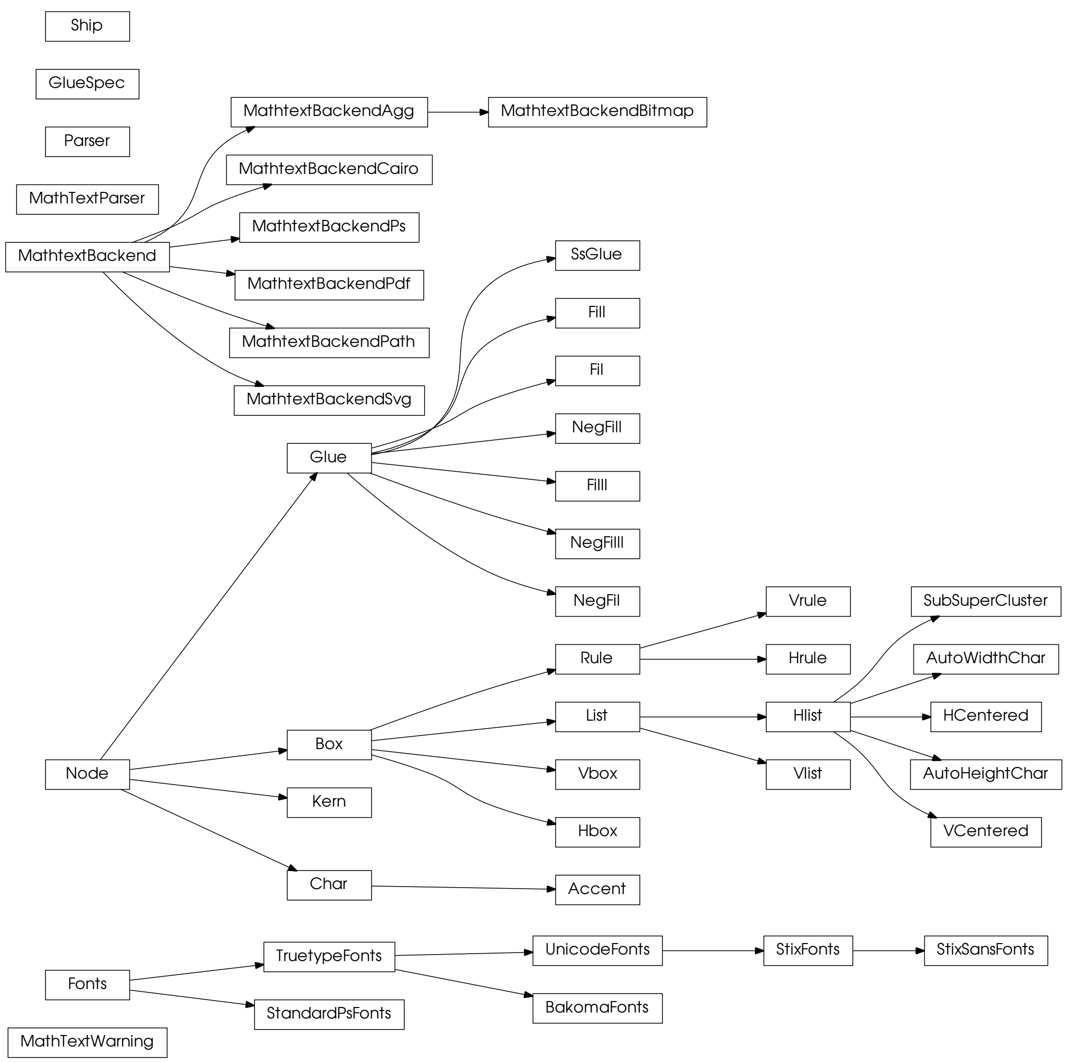
classmethod update_default_handler_map(*handler_map*)
A class method to update the default handler map.

zorder = 5

CHAPTER

SIXTYFOUR

MATHTEXT



64.1 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.

If you find TeX expressions that don't parse or render properly, please email mdroe@stsci.edu, but please check KNOWN ISSUES below first.

class `matplotlib.mathtext.`**`Accent`**(*c*, *state*)

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*)
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()

matplotlib.mathtext.**Error**(*msg*)
Helper class to raise parser errors.

class matplotlib.mathtext.**Fil**
Bases: matplotlib.mathtext.Glue

class matplotlib.mathtext.**Fill**
Bases: matplotlib.mathtext.Glue

class matplotlib.mathtext.**Filll**
Bases: matplotlib.mathtext.Glue

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.

default_font_prop: A `FontProperties` object to use for the default non-math font, or the base font for Unicode (generic) font rendering.

mathtext_backend: A subclass of `MathTextBackend` used to delegate the actual rendering.

destroy()
Fix any cyclical references before the object is about to be destroyed.

get_kern(*font1*, *fontclass1*, *sym1*, *fontsize1*, *font2*, *fontclass2*, *sym2*, *fontsize2*, *dpi*)
Get the kerning distance for font between *sym1* and *sym2*.

fontX: one of the TeX font names:

```
tt, it, rm, cal, sf, bf or default/regular (non-math)
```

fontclassX: TODO

symX: a symbol in raw TeX form. e.g. '1', 'x' or 'sigma'

fontsizeX: the fontsize in points

dpi: the current dots-per-inch

get_metrics(*font*, *font_class*, *sym*, *fontsize*, *dpi*)
font: one of the TeX font names:

```
tt, it, rm, cal, sf, bf or default/regular (non-math)
```

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

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.

render_rect_filled(*x1*, *y1*, *x2*, *y2*)
Draw a filled rectangle from (*x1*, *y1*) to (*x2*, *y2*).

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.

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.)

grow()

shrink()

class `matplotlib.mathtext.GlueSpec`(*width=0.0*, *stretch=0.0*, *stretch_order=0*, *shrink=0.0*, *shrink_order=0*)

Bases: `object`

See `Glue`.

copy()

classmethod factory(*glue_type*)

class `matplotlib.mathtext.HCentered`(*elements*)

Bases: `matplotlib.mathtext.Hlist`

A convenience class to create an `Hlist` whose contents are centered within its enclosing box.

class `matplotlib.mathtext.Hbox`(*width*)

Bases: `matplotlib.mathtext.Box`

A box with only width (zero height and depth).

class `matplotlib.mathtext.Hlist`(*elements*, *w=0.0*, *m='additional'*, *do_kern=True*)

Bases: `matplotlib.mathtext.List`

A horizontal list of boxes.

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.

- *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, eg r'IQ: $sigma_i=15$'

dpi The dots-per-inch to render the text

fontsize The font size in points

parse(*s*, *dpi=72*, *prop=None*)
Parse the given math expression *s* at the given *dpi*. If *prop* is provided, it is a `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 `parse()` with the same expression should be fast.

to_mask(*texstr*, *dpi=120*, *fontsize=14*)

texstr A valid mathtext string, eg r'IQ: $sigma_i=15$'

dpi The dots-per-inch to render the text

fontsize The font size in points

Returns a tuple (*array*, *depth*)

- *array* is an NxM uint8 alpha ubyte mask array of rasterized tex.

- depth is the offset of the baseline from the bottom of the image in pixels.

to_png(*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.

filename A writable filename or fileobject

texstr A valid mathtext string, eg r'IQ: $sigma_i=15$'

color A valid matplotlib color argument

dpi The dots-per-inch to render the text

fontsize The font size in points

Returns the offset of the baseline from the bottom of the image in pixels.

to_rgba(*texstr*, *color='black'*, *dpi=120*, *fontsize=14*)

texstr A valid mathtext string, eg r'IQ: $sigma_i=15$'

color Any matplotlib color argument

dpi The dots-per-inch to render the text

fontsize The font size in points

Returns a tuple (*array*, *depth*)

- *array* is an NxM uint8 alpha ubyte mask array of rasterized tex.
- depth is the offset of the baseline from the bottom of the image in pixels.

exception `matplotlib.mathtext.`**`MathTextWarning`**

Bases: `exceptions.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_filled_rect()`
- `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_filled_rect(*x1*, *y1*, *x2*, *y2*)
Draw a filled black rectangle from (*x1*, *y1*) to (*x2*, *y2*).

render_glyph(*ox*, *oy*, *info*)
Draw a glyph described by *info* to the reference point (*ox*, *oy*).

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(*x1*, *y1*, *x2*, *y2*)

set_canvas_size(*w*, *h*, *d*)

class matplotlib.mathtext.**MathtextBackendBitmap**
Bases: matplotlib.mathtext.MathtextBackendAgg

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

Parser.**accent**(*s*, *loc*, *toks*)

Parser.**auto_delim**(*s*, *loc*, *toks*)

Parser.**binom**(*s*, *loc*, *toks*)

Parser.**c_over_c**(*s*, *loc*, *toks*)

Parser.**customspace**(*s*, *loc*, *toks*)

Parser.**end_group**(*s*, *loc*, *toks*)

Parser.**font**(*s*, *loc*, *toks*)

Parser.**frac**(*s*, *loc*, *toks*)

Parser.**function**(*s*, *loc*, *toks*)

Parser.**genfrac**(*s*, *loc*, *toks*)

Parser.**get_state**()

Get the current State of the parser.

Parser.**group**(*s*, *loc*, *toks*)

Parser.**is_dropsub**(*nucleus*)

Parser.**is_overunder**(*nucleus*)

Parser.**is_slanted**(*nucleus*)

Parser.**main**(*s*, *loc*, *toks*)

Parser.**math**(*s*, *loc*, *toks*)

Parser.**math_string**(*s*, *loc*, *toks*)

Parser.**non_math**(*s*, *loc*, *toks*)

Parser.**operatorname**(*s*, *loc*, *toks*)

Parser.**overline**(*s*, *loc*, *toks*)

Parser.**parse**(*s*, *fonts_object*, *fontsize*, *dpi*)

Parse expression *s* using the given *fonts_object* for output, at the given *fontsize* and *dpi*.

Returns the parse tree of `Node` instances.

Parser.**pop_state**()

Pop a `State` off of the stack.

Parser.**push_state**()

Push a new `State` onto the stack which is just a copy of the current state.

Parser.**required_group**(*s*, *loc*, *toks*)

Parser.**simple_group**(*s*, *loc*, *toks*)

Parser.**space**(*s*, *loc*, *toks*)

Parser.**sqrt**(*s*, *loc*, *toks*)

Parser.**stackrel**(*s*, *loc*, *toks*)

Parser.**start_group**(*s*, *loc*, *toks*)

Parser.**subsuper**(*s*, *loc*, *toks*)

Parser.**symbol**(*s*, *loc*, *toks*)

Parser.**unknown_symbol**(*s*, *loc*, *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.**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 = '/usr/src/tmp/python-module-matplotlib-buildroot/usr/lib/python2.7/site-packages/matp

fontmap = {'bf': 'pncb8a', 'tt': 'pcrr8a', 'it': 'pncri8a', None: 'psyr', 'cal': 'pzcmi8a', 'rm': 'pncr8

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).

class **`CachedFont`**(*font*)

`TruetypeFonts.`**`destroy`**()

`TruetypeFonts.`**`get_kern`**(*font1*, *fontclass1*, *sym1*, *fontsize1*, *font2*, *fontclass2*, *sym2*, *fontsize2*, *dpi*)

`TruetypeFonts.`**`get_underline_thickness`**(*font*, *fontsize*, *dpi*)

`TruetypeFonts.`**`get_xheight`**(*font*, *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.

`vpack`(*h=0.0*, *m='additional'*, *l=inf*)
The main duty of `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, `vpack(h, 'exactly')` produces a box whose height is exactly *h*, while `vpack(h, 'additional')` yields a box whose height is the natural height plus *h*. The default values produce a box with the natural width.

class `matplotlib.mathtext.`**`Vrule`**(*state*)

Bases: `matplotlib.mathtext.Rule`

Convenience class to create a vertical rule.

`matplotlib.mathtext.`**`get_unicode_index`**(*symbol*) → integer

Return the integer index (from the Unicode table) of symbol. *symbol* can be a single unicode character, a TeX command (i.e. r'pi'), or a Type1 symbol name (i.e. 'phi').

`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.

s A math expression. The math portion should be enclosed in dollar signs.

filename_or_obj A filepath or writable file-like object to write the image data to.

prop If provided, a FontProperties() object describing the size and style of the text.

dpi Override the output dpi, otherwise use the default associated with the output format.

format The output format, eg. 'svg', 'pdf', 'ps' or 'png'. If not provided, will be deduced from the filename.

`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.

CHAPTER

SIXTYFIVE

MLAB

65.1 `matplotlib.mlab`

Numerical python functions written for compatability with MATLAB commands with the same names.

65.1.1 MATLAB compatible functions

`cohere()` Coherence (normalized cross spectral density)

`csd()` Cross spectral density uing 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 uing Welch's average periodogram

`rk4()` A 4th order runge kutta integrator for 1D or ND systems

`specgram()` Spectrogram (power spectral density over segments of time)

65.1.2 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

65.1.3 record array helper functions

A collection of helper methods for numpyrecord arrays

See *misc-examples-index*

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 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:

```
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_git.Window()
win.set_size_request(600,800)
win.add(scroll)
win.show_all()
gtk_git.main()
```

65.1.4 Deprecated functions

The following are deprecated; please import directly from numpy (with care–function signatures may differ):

`load()` load ASCII file - use numpy.loadtxt

`save()` save ASCII file - use numpy.savetxt

class `matplotlib.mlab.`**`FIFOBuffer`**(*nmax*)

A FIFO queue to hold incoming *x*, *y* data in a rotating buffer using numpy arrays under the hood. It is assumed that you will call asarrays much less frequently than you add data to the queue – otherwise another data structure will be faster.

This can be used to support plots where data is added from a real time feed and the plot object wants to grab data from the buffer and plot it to screen less freqeuently than the incoming.

If you set the *dataLim* attr to `BBox` (eg `matplotlib.Axes.dataLim`), the *dataLim* will be updated as new data come in.

TODO: add a grow method that will extend nmax

Note: mlab seems like the wrong place for this class.

Buffer up to *nmax* points.

`add`(*x*, *y*)

Add scalar *x* and *y* to the queue.

`asarrays`()

Return *x* and *y* as arrays; their length will be the len of data added or *nmax*.

`last`()

Get the last *x*, *y* or *None*. *None* if no data set.

`register`(*func*, *N*)

Call *func* every time *N* events are passed; *func* signature is `func(fifo)`.

update_datalim_to_current()
Update the *datalim* in the current data in the fifo.

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**

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.`**`PCA`**(*a*)
compute the SVD of a and store data for PCA. Use project to project the data onto a reduced set of dimensions

Inputs:

a: a numobservations x numdims array

Attrs:

a a centered unit sigma version of input a

numrows, *numcols*: the dimensions of a

mu : a numdims array of means of a

sigma : a numdims array of atandard deviation of a

fracs : the proportion of variance of each of the principal components

Wt : the weight vector for projecting a numdims point or array into PCA space

Y : a projected into PCA space

The factor loadings are in the Wt factor, ie the factor loadings for the 1st principal component are given by Wt[0]

`center`(*x*)
center 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.`**`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 at 0x9bc0a3c>*, *window=<function window_hanning at 0x9bc0924>*, *noverlap=0*, *pad_to=None*, *sides='default'*, *scale_by_freq=None*)

The coherence between *x* and *y*. Coherence is the normalized cross spectral density:

$$C_{xy} = \frac{|P_{xy}|^2}{P_{xx}P_{yy}} \tag{65.1}$$

x*, *y Array or sequence containing the data

Keyword arguments:

***NFFT*: integer** The number of data points used in each block for the FFT. Must be even; 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.

***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.

***detrend*: 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.

***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.

***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 psd (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*

***sides*: ['default' | 'onesided' | 'twosided']** Specifies which sides of the PSD 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 PSD, while 'twosided' forces two-sided.

***scale_by_freq*: boolean** 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).

The return value is the tuple (Cxy, f), where f are the frequencies of the coherence vector. For cohere, scaling the individual densities by the sampling frequency has no effect, since the factors cancel out.

See Also:

psd() and csd() For information about the methods used to compute P_{xy}, P_{xx} and P_{yy}.

matplotlib.mlab.**cohere_pairs**(*X*, *ij*, *NFFT=256*, *Fs=2*, *detrend=<function detrend_none at 0x9bc0a3c>*, *window=<function window_hanning at 0x9bc0924>*, *noverlap=0*, *preferSpeedOverMemory=True*, *progressCallback=<function donothing_callback at 0x9bc0ca4>*, *returnPxx=False*)

Call signature:

```
Cxy, Phase, freqs = cohere_pairs( X, ij, ...)
```

Compute the coherence and phase for all pairs *ij*, in *X*.

X is a *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:

```
ij = []
for i in range(64):
    for j in range(i+1,64):
        ij.append( (i,j) )
```

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 *preferSpeedOverMemory* is False, `cohere_pairs()` will still give significant performace gains over calling `cohere()` for each pair, and will use subtantially less memory than if *preferSpeedOverMemory* is True. In my tests with a 43000,64 array over all nonredundant pairs, *preferSpeedOverMemory* = True delivered a 33% performance boost on a 1.7GHZ Athlon with 512MB RAM compared with *preferSpeedOverMemory* = False. But both solutions were more than 10x faster than naively crunching all possible pairs through `cohere()`.

Returns:

```
(Cxy, Phase, freqs)
```

where:

- *Cxy*: dictionary of (i, j) tuples -> coherence vector for that pair. I.e., `Cxy[(i,j) = cohere(X[:,i], X[:,j])`. Number of dictionary keys is `len(ij)`.
- *Phase*: dictionary of phases of the cross spectral density at each frequency for each pair. Keys are (i, j).
- ***freqs*: vector of frequencies, equal in length to either the** coherence or phase vectors for any (i, j) key.

Eg., to make a coherence Bode plot:

```
subplot(211)
plot( freqs, Cxy[(12,19)])
subplot(212)
plot( freqs, Phase[(12,19)])
```

For a large number of pairs, `cohere_pairs()` can be much more efficient than just calling `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 `cohere_pairs()` and `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.`**`contiguous_regions`**`(`*mask*`)`

return a list of (ind0, ind1) such that mask[ind0:ind1].all() is True and we cover all such regions

TODO: this is a pure python implementation which probably has a much faster numpy impl matplotlib.mlab.**cross_from_above**(*x*, *threshold*)

return the indices into *x* where *x* crosses some threshold from below, eg the i's where:

```
x[i-1]>threshold and x[i]<=threshold
```

See Also:

cross_from_below() and contiguous_regions()

matplotlib.mlab.**cross_from_below**(*x*, *threshold*)

return the indices into *x* where *x* crosses some threshold from below, eg the i's where:

```
x[i-1]<threshold and x[i]>=threshold
```

Example code:

```
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() and contiguous_regions()

matplotlib.mlab.**csd**(*x*, *y*, *NFFT=256*, *Fs=2*, *detrend=<function detrend_none at 0x9bc0a3c>*, *window=<function window_hanning at 0x9bc0924>*, *noverlap=0*, *pad_to=None*, *sides='default'*, *scale_by_freq=None*)

The cross power spectral density by Welch's average periodogram method. The vectors *x* and *y* are divided into *NFFT* length blocks. Each block is detrended by the function *detrend* and windowed by the function *window*. *noverlap* gives the length of the overlap between blocks. The product of the direct FFTs of *x* and *y* are averaged over each segment to compute *Pxy*, 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*.

x*, *y Array or sequence containing the data

Keyword arguments:

***NFFT*: integer** The number of data points used in each block for the FFT. Must be even; 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.

***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.

***detrend*: 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.

***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.

***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 psd (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*

***sides*: ['default' | 'onesided' | 'twosided']** Specifies which sides of the PSD 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 PSD, while 'twosided' forces two-sided.

***scale_by_freq*: boolean** 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).

Returns the tuple (*Pxy*, *freqs*).

Refs: Bendat & Piersol – Random Data: Analysis and Measurement Procedures, John Wiley & Sons (1986)

`matplotlib.mlab.csv2rec`(*fname*, *comments='#'*, *skiprows=0*, *checkrows=0*, *delimiter=','*, *converterd=None*, *names=None*, *missing=''*, *missingd=None*, *use_mrecords=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
- *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

  If no rows are found, *None* is returned – see `examples/loadrec.py`

`matplotlib.mlab.csvformat_factory`(*format*)

`matplotlib.mlab.demean`(*x*, *axis=0*)

Return x minus its mean along the specified axis

`matplotlib.mlab.detrend`(*x*, *key=None*)

`matplotlib.mlab.detrend_linear`(*y*)

Return y minus best fit line; 'linear' detrending

`matplotlib.mlab.`**`detrend_mean`**(*x*)
Return x minus the mean(x)

`matplotlib.mlab.`**`detrend_none`**(*x*)
Return x: no detrending

`matplotlib.mlab.`**`dist`**(*x*, *y*)
Return the distance between two points.

`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://softsurfer.com/Archive/algorithm_0102/algorithm_0102.htm#Distance%20to%2

`matplotlib.mlab.`**`distances_along_curve`**(*X*)
Computes the distance between a set of successive points in *N* dimensions.

Where *X* is an *M* x *N* array or matrix. The distances between successive rows is computed. Distance is the standard Euclidean distance.

`matplotlib.mlab.`**`donothing_callback`**(**args*)

`matplotlib.mlab.`**`entropy`**(*y*, *bins*)
Return the entropy of the data in *y*.

$$\sum p_i \log_2(p_i) \tag{65.2}$$

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:

```
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 at 0x9bc0a3c>*, *window=<function window_none at 0x9bc095c>*)
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:

```
>>> 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.

matplotlib.mlab.**griddata**(*x*, *y*, *z*, *xi*, *yi*, *interp='nn'*)
zi = griddata(x,y,z,xi,yi) fits a surface of the form *z = f*(*x, y*) to the data in the (usually) nonuniformly spaced vectors (*x*, *y*, *z*). griddata() interpolates this surface at the points specified by (*xi*, *yi*) to produce *zi*. *xi* and *yi* must describe a regular grid, can be either 1D or 2D, but must be monotonically increasing.

A masked array is returned if any grid points are outside convex hull defined by input data (no extrapolation is done).

If interp keyword is set to 'nn' (default), uses natural neighbor interpolation based on Delaunay triangulation. By default, this algorithm is provided by the matplotlib.delaunay package, written by Robert Kern. The triangulation algorithm in this package is known to fail on some nearly pathological cases. For this reason, a separate toolkit (`mpl_tookits.natgrid`) has been created that provides a more robust algorithm fof triangulation and interpolation. This toolkit is based on the NCAR natgrid library, which contains code that is not redistributable under a BSD-compatible license. When installed, this function will use the `mpl_toolkits.natgrid` algorithm, otherwise it will use the built-in `matplotlib.delaunay` package.

If the interp keyword is set to '`linear`', then linear interpolation is used instead of natural neighbor. In this case, the output grid is assumed to be regular with a constant grid spacing in both the x and y directions. For regular grids with nonconstant grid spacing, you must use natural neighbor interpolation. Linear interpolation is only valid if `matplotlib.delaunay` package is used - `mpl_tookits.natgrid` only provides natural neighbor interpolation.

The natgrid matplotlib toolkit can be downloaded from http://sourceforge.net/project/showfiles.php?group_id=80706&package_id=142792

`matplotlib.mlab.`**`identity`**(*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:

```
                    /  1  if i0=i1=...=iR,
id[i0,i1,...,iR] = -|
                    \  0  otherwise.
```

Optionally a *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`**(*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.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.l1norm(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.levypdf(x, gamma, alpha)`
Returm the levy pdf evaluated at *x* for params *gamma*, *alpha*

`matplotlib.mlab.liaupunov(x, fprime)`
x is a very long trajectory from a map, and *fprime* returns the derivative of *x*.

This function will be removed from matplotlib.

Returns : .. math:

```
\lambda = \frac{1}{n}\sum \ln|f^'(x_i)|
```

See Also:

Lyapunov Exponent Sec 10.5 Strogatz (1994) "Nonlinear Dynamics and Chaos". Wikipedia article on Lyapunov Exponent.

Note: What the function here calculates may not be what you really want; *caveat emptor*.

It also seems that this function's name is badly misspelled.

`matplotlib.mlab.load(fname, comments='#', delimiter=None, converters=None, skiprows=0, usecols=None, unpack=False, dtype=<type 'numpy.float64'>)`
Load ASCII data from *fname* into an array and return the array.

Deprecated: use numpy.loadtxt.

The data must be regular, same number of values in every row

fname can be a filename or a file handle. Support for gzipped files is automatic, if the filename ends in '.gz'.

matfile data is not supported; for that, use `scipy.io.mio` module.

Example usage:

```
X = load('test.dat')  # data in two columns
t = X[:,0]
y = X[:,1]
```

Alternatively, you can do the same with "unpack"; see below:

```
X = load('test.dat')    # a matrix of data
x = load('test.dat')    # a single column of data
```

- *comments*: the character used to indicate the start of a comment in the file
- *delimiter* is a string-like character used to seperate values in the file. If *delimiter* is unspecified or *None*, any whitespace string is a separator.
- *converters*, if not *None*, is a dictionary mapping column number to a function that will convert that column to a float (or the optional *dtype* if specified). Eg, if column 0 is a date string:

  ```
  converters = {0:datestr2num}
  ```

- *skiprows* is the number of rows from the top to skip.
- *usecols*, if not *None*, is a sequence of integer column indexes to extract where 0 is the first column, eg `usecols=[1,4,5]` to extract just the 2nd, 5th and 6th columns
- *unpack*, if *True*, will transpose the matrix allowing you to unpack into named arguments on the left hand side:

  ```
  t,y = load('test.dat', unpack=True) # for  two column data
  x,y,z = load('somefile.dat', usecols=[3,5,7], unpack=True)
  ```

- *dtype*: the array will have this dtype. default: `numpy.float_`

See Also:

See `examples/pylab_examples/load_converter.py` in the source tree Exercises many of these options.

`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*) `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.`**`movavg`**(*x*, *n*)

Compute the len(*n*) moving average of *x*.

`matplotlib.mlab.`**`norm_flat`**(*a*, *p=2*)

norm(a,p=2) -> l-p norm of a.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*, **args*)

Return the normal pdf evaluated at *x*; args provides *mu*, *sigma*

`matplotlib.mlab.`**`offset_line`**(*y*, *yerr*)

Offsets an array *y* by +/- an error and returns a tuple (y - err, y + 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 +/- err are computed component-wise.
- A tuple of length 2. In this case, yerr[0] is the error below *y* and yerr[1] is error above *y*. For example:

```
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* x *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.`**`poly_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()`, eg:

```
xv, yv = poly_below(0, x, y)
ax.fill(xv, yv)
```

matplotlib.mlab.**poly_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 ith 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*). Eg 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 >= the 25th and < 50th 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.**prepca**(*P*, *frac=0*)

WARNING: this function is deprecated – please see class PCA instead

Compute the principal components of *P*. *P* is a (*numVars*, *numObs*) array. *frac* is the minimum fraction of variance that a component must contain to be included.

Return value is a tuple of the form (*Pcomponents*, *Trans*, *fracVar*) where:

- *Pcomponents* : a (numVars, numObs) array
- ***Trans*** [the weights matrix, ie, *Pcomponents* = *Trans* *] *P*
- ***fracVar*** [the fraction of the variance accounted for by each] component returned

A similar function of the same name was in the MATLAB R13 Neural Network Toolbox but is not found in later versions; its successor seems to be called "processpcs".

matplotlib.mlab.**psd**(*x*, *NFFT=256*, *Fs=2*, *detrend=<function detrend_none at 0x9bc0a3c>*, *window=<function window_hanning at 0x9bc0924>*, *noverlap=0*, *pad_to=None*, *sides='default'*, *scale_by_freq=None*)

The power spectral density by Welch's average periodogram method. The vector *x* is divided into *NFFT* length blocks. Each block is detrended by the function *detrend* and windowed by the function *window*. *noverlap* gives the length of the overlap between blocks. The absolute(fft(block))**2 of each segment are averaged to compute *Pxx*, with a scaling to correct for power loss due to windowing.

If len(*x*) < *NFFT*, it will be zero padded to *NFFT*.

x Array or sequence containing the data

Keyword arguments:

***NFFT*: integer** The number of data points used in each block for the FFT. Must be even; 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.

***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.

***detrend*: 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.

***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.

***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 psd (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*

***sides*: ['default' | 'onesided' | 'twosided']** Specifies which sides of the PSD 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 PSD, while 'twosided' forces two-sided.

***scale_by_freq*: boolean** 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).

Returns the tuple (*Pxx*, *freqs*).

Refs:

> Bendat & Piersol – Random Data: Analysis and Measurement Procedures, John Wiley & Sons (1986)

`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.

r: numpy recarray

header: list of 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 : 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'

Example:

```
precision=[0,2,3]
```

Output:

```
ID    Price   Return
ABC   12.54    0.234
XYZ    6.32   -0.076
```

`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. eg ('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:

```
stats = ( ('sales', len, 'numsales'), ('sales', np.mean, 'avgsale') )
```

Return record array has *dtype* names for each attribute name in the 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 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:

```
r = recs_join("date", "close", recs=[r0, r1], missing=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()`.

y0 initial state vector

t sample times

derivs returns the derivative of the system and has the signature `dy = derivs(yi, ti)`

Example 1

```
## 2D system

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)
```

Example 2:

```
## 1D system
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.

`matplotlib.mlab.`**`rms_flat`**(*a*)
Return the root mean square of all the elements of *a*, flattened out.

`matplotlib.mlab.`**`safe_isinf`**(*x*)
`numpy.isinf()` for arbitrary types

`matplotlib.mlab.`**`safe_isnan`**(*x*)
`numpy.isnan()` for arbitrary types

`matplotlib.mlab.`**`save`**(*fname, X, fmt='%.18e', delimiter=' '*)
Save the data in *X* to file *fname* using *fmt* string to convert the data to strings.

Deprecated. Use numpy.savetxt.

fname can be a filename or a file handle. If the filename ends in '.gz', the file is automatically saved in compressed gzip format. The `load()` function understands gzipped files transparently.

Example usage:

```
save('test.out', X)          # X is an array
save('test1.out', (x,y,z))   # x,y,z equal sized 1D arrays
save('test2.out', x)         # x is 1D
save('test3.out', x, fmt='%1.4e')  # use exponential notation
```

delimiter is used to separate the fields, eg. *delimiter* ',' for comma-separated values.

`matplotlib.mlab.`**`segments_intersect`**(*s1, s2*)
Return *True* if *s1* and *s2* intersect. *s1* and *s2* are defined as:

```
s1: (x1, y1), (x2, y2)
s2: (x3, y3), (x4, y4)
```

`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 at physik.uni-regensburg.de

(inspired by a original implementation by Halldor Bjornsson, Icelandic Meteorological Office, March 2006 halldor at vedur.is)

`matplotlib.mlab.specgram`(*x*, *NFFT=256*, *Fs=2*, *detrend=<function detrend_none at 0x9bc0a3c>*, *window=<function window_hanning at 0x9bc0924>*, *noverlap=128*, *pad_to=None*, *sides='default'*, *scale_by_freq=None*)

Compute a spectrogram of data in *x*. Data are split into *NFFT* length segments and the PSD 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*.

If *x* is real (i.e. non-complex) only the spectrum of the positive frequencie is returned. If *x* is complex then the complete spectrum is returned.

Keyword arguments:

***NFFT*: integer** The number of data points used in each block for the FFT. Must be even; 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.

***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.

***detrend*: 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.

***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.

***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 psd (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*

***sides*: ['default' | 'onesided' | 'twosided']** Specifies which sides of the PSD 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 PSD, while 'twosided' forces two-sided.

***scale_by_freq*: boolean** 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 128.

Returns a tuple (*Pxx*, *freqs*, *t*):

- *Pxx*: 2-D array, columns are the periodograms of successive segments
- *freqs*: 1-D array of frequencies corresponding to the rows in Pxx
- *t*: 1-D array of times corresponding to midpoints of segments.

See Also:

`psd()` `psd()` differs in the default overlap; in returning the mean of the segment periodograms; and in not returning times.

`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:

```
x = linspace(0,2*pi,20);  y = sin(x); yp = cos(x)
xi = linspace(0,2*pi,40);
yi = stineman_interp(xi,x,y,yp);
plot(x,y,'o',xi,yi)
```

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* = *None*, the routine automatically determines the slopes using the `slopes()` routine.

x is assumed to be sorted in increasing order.

For values `xi[j] < x[0]` or `xi[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 at vedur.is

Completely reworked and optimized for Python by Norbert Nemec, Institute of Theoretical Physics, University or Regensburg, April 2006 Norbert.Nemec at physik.uni-regensburg.de

`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*.

`matplotlib.mlab.`**`window_hanning`**(*x*)

return x times the hanning window of len(x)

`matplotlib.mlab.`**`window_none`**(*x*)

No window function; simply return x

CHAPTER

SIXTYSIX

NXUTILS

66.1 matplotlib.nxutils

matplotlib.nxutils.**pnpoly**(*x*, *y*, *xyverts*)

inside = pnpoly(x, y, xyverts)

Return 1 if x,y is inside the polygon, 0 otherwise.

xyverts a sequence of x,y vertices.

A point on the boundary may be treated as inside or outside. Deprecated since version 1.2.0: Use `contains_point()` instead.

matplotlib.nxutils.**points_inside_poly**(*xypoints*, *xyverts*)

mask = points_inside_poly(xypoints, xyverts)

Returns a boolean ndarray, True for points inside the polygon.

xypoints a sequence of N x,y pairs.

xyverts sequence of x,y vertices of the polygon.

A point on the boundary may be treated as inside or outside. Deprecated since version 1.2.0: Use `contains_points()` instead.

CHAPTER

SIXTYSEVEN

PATH

67.1 matplotlib.path

Contains a class for managing paths (polylines).

class `matplotlib.path.`**`Path`**(*vertices*, *codes=None*, *_interpolation_steps=1*, *closed=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()` 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.

vertices is an Nx2 numpy float array, masked array or Python sequence.

codes is an N-length numpy array or Python sequence of type `matplotlib.path.Path.code_type`.

These two arrays must have the same length in the first dimension.

If *codes* is None, *vertices* will be treated as a series of line segments.

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()`.

interpolation_steps is 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.

CLOSEPOLY = 79

CURVE3 = 3

CURVE4 = 4

LINETO = 2

MOVETO = 1

NUM_VERTICES = [1, 1, 1, 2, 3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

STOP = 0

classmethod arc(*theta1*, *theta2*, *n=None*, *is_wedge=False*)

(staticmethod) Returns 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.

code_type
alias of uint8

contains_path(*path*, *transform=None*)
Returns *True* if this path completely contains the given path.

If *transform* is not *None*, the path will be transformed before performing the test.

contains_point(*point*, *transform=None*, *radius=0.0*)
Returns *True* if the path contains the given point.

If *transform* is not *None*, the path will be transformed before performing the test.

radius allows the path to be made slightly larger or smaller.

contains_points(*points*, *transform=None*, *radius=0.0*)
Returns a bool array which is *True* if the path contains the corresponding point.

If *transform* is not *None*, the path will be transformed before performing the test.

radius allows the path to be made slightly larger or smaller.

get_extents(*transform=None*)
Returns the extents (*xmin*, *ymin*, *xmax*, *ymax*) of the path.

Unlike computing the extents on the *vertices* alone, this algorithm will take into account the curves and deal with control points appropriately.

classmethod **hatch**(*hatchpattern*, *density=6*)
Given a hatch specifier, *hatchpattern*, generates a Path that can be used in a repeated hatching pattern. *density* is the number of lines per unit square.

interpolated(*steps*)
Returns a new path resampled to length N x steps. Does not currently handle interpolating curves.

intersects_bbox(*bbox*, *filled=True*)
Returns *True* if this path intersects a given Bbox.

filled, 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.

filled, 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*)
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.

***transform*: if not None, the given affine transformation will** be applied to the path.

***remove_nans*: if True, will remove all NaNs from the path and** insert MOVETO commands to skip over them.

***clip*: if not None, must be a four-tuple (x1, y1, x2, y2)** defining a rectangle in which to clip the path.

***snap*: if None, auto-snap to pixels, to reduce** fuzziness of rectilinear lines. If True, force snapping, and if False, don't snap.

***stroke_width*: the width of the stroke being drawn. Needed** as a hint for the snapping algorithm.

***simplify*: 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*: If True, curve segments will be returned as curve** segments. If False, all curves will be converted to line segments.

classmethod `make_compound_path`(**args*)

(staticmethod) Make a compound path from a list of Path objects. Only polygons (not curves) are supported.

classmethod `make_compound_path_from_polys`(*XY*)

(static method) 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`

`to_polygons`(*transform=None*, *width=0*, *height=0*)

Convert this path to a list of polygons. Each polygon 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.

`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()

(staticmethod) Returns a Path of the unit circle. 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.

classmethod unit_circle_righthalf()

(staticmethod) Returns 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()

(staticmethod) Returns a Path of the unit rectangle from (0, 0) to (1, 1).

classmethod unit_regular_asterisk(*numVertices*)

(staticmethod) Returns 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*)

(staticmethod) Returns a Path 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*)

(staticmethod) Returns a Path for a unit regular star with the given numVertices and radius of 1.0, centered at (0, 0).

classmethod wedge(*theta1*, *theta2*, *n=None*)

(staticmethod) Returns 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.**cleanup_path**(*path*, *trans*, *remove_nans*, *clip*, *snap*, *simplify*, *curves*)

matplotlib.path.**convert_path_to_polygons**(*path*, *trans*, *width*, *height*)

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 Path instances.

transforms is a sequence of Affine2D instances.

offsets is a sequence of (x, y) offsets (or an Nx2 array)

offset_transform is a `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)

`matplotlib.path.`**`get_path_extents`**(*path*, *trans*)

`matplotlib.path.`**`get_paths_extents`**(*paths*, *transforms*=[ ])

Given a sequence of `Path` objects and optional `Transform` objects, returns the bounding box that encapsulates all of them.

paths is a sequence of `Path` instances.

transforms is an optional sequence of `Affine2D` instances to apply to each path.

`matplotlib.path.`**`path_in_path`**(*a*, *atrans*, *b*, *btrans*)

`matplotlib.path.`**`path_intersects_path`**(*p1*, *p2*)

`matplotlib.path.`**`point_in_path`**(*x*, *y*, *path*, *trans*)

`matplotlib.path.`**`point_in_path_collection`**(*x*, *y*, *r*, *trans*, *paths*, *transforms*, *offsets*, *offsetTrans*, *filled*)

`matplotlib.path.`**`points_in_path`**(*points*, *path*, *trans*)

CHAPTER

SIXTYEIGHT

PYPLOT

68.1 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.:

```
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*, ***kwargs*)

Plot the autocorrelation of *x*.

Call signature:

```
acorr(x, normed=True, detrend=mlab.detrend_none, usevlines=True,
      maxlags=10, **kwargs)
```

If *normed* = *True*, normalize the data by the autocorrelation at 0-th lag. *x* is detrended by the *detrend* callable (default no normalization).

Data are plotted as `plot(lags, c, **kwargs)`

Return value is a tuple (*lags*, *c*, *line*) 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()`

The default *linestyle* is None and the default *marker* is `'o'`, though these can be overridden with keyword args. The cross correlation is performed with `numpy.correlate()` with *mode* = 2.

If *usevlines* is *True*, `vlines()` rather than `plot()` is used to draw vertical lines from the origin to the acorr. Otherwise, the plot style is determined by the kwargs, which are `Line2D` properties.

maxlags is a positive integer detailing the number of lags to show. The default value of *None* will return all `(2*len(x)-1)` lags.

The return value is a tuple (*lags*, *c*, *linecol*, *b*) where

- *linecol* is the `LineCollection`
- *b* is the *x*-axis.

See Also:

`plot()` or `vlines()` For documentation on valid kwargs.

Example:

`xcorr()` is top graph, and `acorr()` is bottom graph.

Additional kwargs: hold = [True|False] overrides default hold state

`matplotlib.pyplot.`**`annotate`**`(`**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)
```

Keyword arguments:

Annotate the *x*, *y* point *xy* with text *s* at *x*, *y* location *xytext*. (If *xytext* = *None*, defaults to *xy*, and if *textcoords* = *None*, defaults to *xycoords*).

arrowprops, if not *None*, is a dictionary of line properties (see `matplotlib.lines.Line2D`) for the arrow that connects annotation to the point.

If the dictionary has a key *arrowstyle*, a FancyArrowPatch instance is created with the given dictionary and is drawn. Otherwise, a YAArow patch instance is created and drawn. Valid keys for YAArow are

Key	Description
width	the width of the arrow in points
frac	the fraction of the arrow length occupied by the head
head-width	the width of the base of the arrow head in points
shrink	oftentimes it is convenient to have the arrowtip and base a bit away from the text and point being annotated. If *d* is the distance between the text and annotated point, shrink will shorten the arrow so the tip and base are shink percent of the distance *d* away from the endpoints. ie, `shrink=0.05 is 5%`
?	any key for `matplotlib.patches.polygon`

Valid keys for FancyArrowPatch are

Key	Description
arrowstyle	the arrow style
connectionstyle	the connection style
relpos	default is (0.5, 0.5)
patchA	default is bounding box of the text
patchB	default is None
shrinkA	default is 2 points
shrinkB	default is 2 points
mutation_scale	default is text size (in points)
mutation_aspect	default is 1.
?	any key for `matplotlib.patches.PathPatch`

xycoords and *textcoords* are strings that indicate the coordinates of *xy* and *xytext*.

Property	Description
'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'	0,1 is lower left of axes and 1,1 is upper right
'data'	use the coordinate system of the object being annotated (default)
'offset points'	Specify an offset (in points) from the *xy* value
'polar'	you can specify *theta*, *r* 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.

If a 'points' or 'pixels' option is specified, values will be added to the bottom-left and if negative, values will be subtracted from the top-right. Eg:

```
# 10 points to the right of the left border of the axes and
# 5 points below the top border
xy=(10,-5), xycoords='axes points'
```

You may use an instance of `Transform` or `Artist`. See *Annotating Axes* for more details.

The *annotation_clip* attribute contols 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".

Additional kwargs are Text properties:

Property	Description
`agg_filter`	unknown
`alpha`	float (0.0 transparent through 1.0 opaque)
`animated`	[True | False]
`axes`	an `Axes` instance
`backgroundcolor`	any matplotlib color
`bbox`	rectangle prop dict

Table 68.1 – cont

Property	Description
clip_box	a matplotlib.transforms.Bbox instance
clip_on	[True | False]
clip_path	[(Path, Transform) | Patch | None]
color	any matplotlib color
contains	a callable function
family or fontfamily or fontname or name	[FONTNAME | 'serif' | 'sans-serif' | 'cursive' | 'fantasy' | 'mono
figure	a matplotlib.figure.Figure instance
fontproperties or font_properties	a matplotlib.font_manager.FontProperties instance
gid	an id string
horizontalalignment or ha	['center' | 'right' | 'left']
label	string or anything printable with '%s' conversion.
linespacing	float (multiple of font size)
lod	[True | False]
multialignment	['left' | 'right' | 'center']
path_effects	unknown
picker	[None|float|boolean|callable]
position	(x,y)
rasterized	[True | False | None]
rotation	[angle in degrees | 'vertical' | 'horizontal']
rotation_mode	unknown
size or fontsize	[size in points | 'xx-small' | 'x-small' | 'small' | 'medium' | 'large
snap	unknown
stretch or fontstretch	[a numeric value in range 0-1000 | 'ultra-condensed' | 'extra-conc
style or fontstyle	['normal' | 'italic' | 'oblique']
text	string or anything printable with '%s' conversion.
transform	Transform instance
url	a url string
variant or fontvariant	['normal' | 'small-caps']
verticalalignment or va or ma	['center' | 'top' | 'bottom' | 'baseline']
visible	[True | False]
weight or fontweight	[a numeric value in range 0-1000 | 'ultralight' | 'light' | 'normal' |
x	float
y	float
zorder	any number

`matplotlib.pyplot.arrow(x, y, dx, dy, hold=None, **kwargs)`

Add an arrow to the axes.

Call signature:

```
arrow(x, y, dx, dy, **kwargs)
```

Draws arrow on specified axis from (*x*, *y*) to (*x* + *dx*, *y* + *dy*). Uses FancyArrow patch to construct the arrow.

Optional kwargs 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 `Patch`) 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
clip_box	a matplotlib.transforms.Bbox instance
clip_on	[True | False]
clip_path	[(Path, Transform) | Patch | None]
color	matplotlib color spec
contains	a callable function
edgecolor or ec	mpl color spec, or None for default, or 'none' for no color
facecolor or fc	mpl color spec, or None for default, or 'none' for no color
figure	a matplotlib.figure.Figure instance
fill	[True | False]
gid	an id string
hatch	['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
label	string or anything printable with '%s' conversion.
linestyle or ls	['solid' | 'dashed' | 'dashdot' | 'dotted']
linewidth or lw	float or None for default
lod	[True | False]
path_effects	unknown
picker	[None|float|boolean|callable]
rasterized	[True | False | None]
snap	unknown
transform	Transform instance
url	a url string
visible	[True | False]
zorder	any number

Example:

Additional kwargs: hold = [True|False] overrides default hold state

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.

`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, axisbg='w')` where *rect* = [left, bottom, width, height] in normalized (0, 1) units. *axisbg* 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.

kwarg	Accepts	Desctiption
axisbg	color	the axes background color
frameon	[True|False]	display the frame?
sharex	otherax	current axes shares xaxis attribute with otherax
sharey	otherax	current axes shares yaxis attribute with otherax
polar	[True|False]	use a polar axes?

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.

Call signature:

```
axhline(y=0, xmin=0, xmax=1, **kwargs)
```

Draw a horizontal line at *y* from *xmin* to *xmax*. With the default values of *xmin* = 0 and *xmax* = 1, this line will always span the horizontal extent of the axes, regardless of the xlim settings, even if you change them, eg. 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.

Return value is the `Line2D` instance. kwargs are the same as kwargs to plot, and can be used to control the line properties. Eg.,

- 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:

```
>>> axhline(y=1)
```

- draw a default hline at $y = .5$ that spans the the middle half of the xrange:

```
>>> axhline(y=.5, xmin=0.25, xmax=0.75)
```

Valid kwargs are `Line2D` properties, with the exception of 'transform':

Property	Description
`agg_filter`	unknown
`alpha`	float (0.0 transparent through 1.0 opaque)
`animated`	[True | False]
`antialiased` or aa	[True | False]
`axes`	an `Axes` instance
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color` or c	any matplotlib color
`contains`	a callable function
`dash_capstyle`	['butt' | 'round' | 'projecting']
`dash_joinstyle`	['miter' | 'round' | 'bevel']
`dashes`	sequence of on/off ink in points
`data`	2D array (rows are x, y) or two 1D arrays
`drawstyle`	['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post']
`figure`	a `matplotlib.figure.Figure` instance
`fillstyle`	['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']
`gid`	an id string
`label`	string or anything printable with '%s' conversion.
`linestyle` or ls	[`'-'` | `'--'` | `'-.'` | `':'` | `'None'` | `' '` | `''`] and any drawstyle in combina
`linewidth` or lw	float value in points
`lod`	[True | False]
`marker`	[`7` | `4` | `5` | `6` | `'o'` | `'D'` | `'h'` | `'H'` | `'_'` | `''` | `'None'` | `None` | `' '` | `'8'` | `'p`
`markeredgecolor` or mec	any matplotlib color
`markeredgewidth` or mew	float value in points
`markerfacecolor` or mfc	any matplotlib color
`markerfacecoloralt` or mfcalt	any matplotlib color
`markersize` or ms	float
`markevery`	None | integer | (startind, stride)
`picker`	float distance in points or callable pick function `fn(artist, event)`
`pickradius`	float distance in points

Table 68.2

Property	Description
rasterized	[True | False | None]
snap	unknown
solid_capstyle	['butt' | 'round' | 'projecting']
solid_joinstyle	['miter' | 'round' | 'bevel']
transform	a matplotlib.transforms.Transform instance
url	a url string
visible	[True | False]
xdata	1D array
ydata	1D array
zorder	any number

See Also:

axhspan() for example plot and source code

Additional kwargs: hold = [True|False] overrides default hold state

matplotlib.pyplot.**axhspan**(*ymin*, *ymax*, *xmin=0*, *xmax=1*, *hold=None*, ***kwargs*)

Add a horizontal span (rectangle) across the axis.

Call signature:

```
axhspan(ymin, ymax, xmin=0, xmax=1, **kwargs)
```

y coords are in data units and *x* coords are in axes (relative 0-1) units.

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, eg. 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.

Return value is a matplotlib.patches.Polygon instance.

Examples:

- draw a gray rectangle from *y* = 0.25-0.75 that spans the horizontal extent of the axes:

```
>>> axhspan(0.25, 0.75, facecolor='0.5', alpha=0.5)
```

Valid kwargs are 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
clip_box	a matplotlib.transforms.Bbox instance
clip_on	[True | False]
clip_path	[(Path, Transform) | Patch | None]
color	matplotlib color spec
contains	a callable function
edgecolor or ec	mpl color spec, or None for default, or 'none' for no color
facecolor or fc	mpl color spec, or None for default, or 'none' for no color
figure	a matplotlib.figure.Figure instance
fill	[True | False]
gid	an id string
hatch	['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
label	string or anything printable with '%s' conversion.
linestyle or ls	['solid' | 'dashed' | 'dashdot' | 'dotted']
linewidth or lw	float or None for default
lod	[True | False]
path_effects	unknown
picker	[None|float|boolean|callable]
rasterized	[True | False | None]
snap	unknown
transform	Transform instance
url	a url string
visible	[True | False]
zorder	any number

Example:

Additional kwargs: hold = [True|False] overrides default hold state

`matplotlib.pyplot.axis(*v, **kwargs)`

Set or get the axis properties.:

```
>>> axis()
```

returns the current axes limits `[xmin, xmax, ymin, ymax]`.:

```
>>> axis(v)
```

sets the min and max of the x and y axes, with `v = [xmin, xmax, ymin, ymax]`.:

```
>>> axis('off')
```

turns off the axis lines and labels.:

```
>>> 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.:

```
>>> axis('scaled')
```

achieves the same result by changing the dimensions of the plot box instead of the axis data limits.:

```
>>> axis('tight')
```

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('image')
```

is 'scaled' with the axis limits equal to the data limits.:

```
>>> axis('auto')
```

and:

```
>>> axis('normal')
```

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.

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.

Call signature:

```
axvline(x=0, ymin=0, ymax=1, **kwargs)
```

Draw a vertical line at *x* from *ymin* to *ymax*. With the default values of *ymin* = 0 and *ymax* = 1, this line will always span the vertical extent of the axes, regardless of the ylim settings, even if you change them, eg. with the `set_ylim()` command. That is, the vertical extent is in axes coords: 0=bottom, 0.5=middle, 1.0=top but the *x* location is in data coordinates.

Return value is the `Line2D` instance. kwargs are the same as kwargs to plot, and can be used to control the line properties. Eg.,

- 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 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>
<tr><th>Property</th><th>Description</th></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>animated</td><td>[True | False]</td></tr>
<tr><td>antialiased or aa</td><td>[True | False]</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 | False]</td></tr>
<tr><td>clip_path</td><td>[(Path, Transform) | Patch | None]</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' | 'round' | 'projecting']</td></tr>
<tr><td>dash_joinstyle</td><td>['miter' | 'round' | 'bevel']</td></tr>
<tr><td>dashes</td><td>sequence of on/off ink in points</td></tr>
<tr><td>data</td><td>2D array (rows are x, y) or two 1D arrays</td></tr>
<tr><td>drawstyle</td><td>['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post']</td></tr>
<tr><td>figure</td><td>a matplotlib.figure.Figure instance</td></tr>
<tr><td>fillstyle</td><td>['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']</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>['-' | '--' | '-.' | ':' | 'None' | ' ' | ''] and any drawstyle in combina</td></tr>
<tr><td>linewidth or lw</td><td>float value in points</td></tr>
<tr><td>lod</td><td>[True | False]</td></tr>
<tr><td>marker</td><td>[7 | 4 | 5 | 6 | 'o' | 'D' | 'h' | 'H' | '_' | '' | 'None' | None | ' ' | '8' | 'p</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></td><td></td></tr>
</table>

Table 68.3

Property	Description
markersize or ms	float
markevery	None | integer | (startind, stride)
picker	float distance in points or callable pick function `fn(artist, event)`
pickradius	float distance in points
rasterized	[True | False | None]
snap	unknown
solid_capstyle	['butt' | 'round' | 'projecting']
solid_joinstyle	['miter' | 'round' | 'bevel']
transform	a `matplotlib.transforms.Transform` instance
url	a url string
visible	[True | False]
xdata	1D array
ydata	1D array
zorder	any number

See Also:

axhspan() for example plot and source code

Additional kwargs: hold = [True|False] overrides default hold state

`matplotlib.pyplot.`**`axvspan`**(*xmin*, *xmax*, *ymin=0*, *ymax=1*, *hold=None*, ***kwargs*)

Add a vertical span (rectangle) across the axes.

Call signature:

```
axvspan(xmin, xmax, ymin=0, ymax=1, **kwargs)
```

x coords are in data units and *y* coords are in axes (relative 0-1) units.

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, eg. 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.

Return value is the `matplotlib.patches.Polygon` instance.

Examples:

- draw a vertical green translucent rectangle from x=1.25 to 1.55 that spans the yrange of the axes:

```
>>> axvspan(1.25, 1.55, facecolor='g', alpha=0.5)
```

Valid kwargs are 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
clip_box	a matplotlib.transforms.Bbox instance
clip_on	[True | False]
clip_path	[(Path, Transform) | Patch | None]
color	matplotlib color spec
contains	a callable function
edgecolor or ec	mpl color spec, or None for default, or 'none' for no color
facecolor or fc	mpl color spec, or None for default, or 'none' for no color
figure	a matplotlib.figure.Figure instance
fill	[True | False]
gid	an id string
hatch	['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
label	string or anything printable with '%s' conversion.
linestyle or ls	['solid' | 'dashed' | 'dashdot' | 'dotted']
linewidth or lw	float or None for default
lod	[True | False]
path_effects	unknown
picker	[None|float|boolean|callable]
rasterized	[True | False | None]
snap	unknown
transform	Transform instance
url	a url string
visible	[True | False]
zorder	any number

See Also:

axhspan() for example plot and source code

Additional kwargs: hold = [True|False] overrides default hold state

matplotlib.pyplot.**bar**(*left*, *height*, *width=0.8*, *bottom=None*, *hold=None*, ***kwargs*)

Make a bar plot.

Call signature:

```
bar(left, height, width=0.8, bottom=0, **kwargs)
```

Make a bar plot with rectangles bounded by:

> ***left*, *left* + *width*, *bottom*, *bottom* + *height*** (left, right, bottom and top edges)

left, *height*, *width*, and *bottom* can be either scalars or sequences

Return value is a list of `matplotlib.patches.Rectangle` instances.

Required arguments:

Argument	Description
left	the x coordinates of the left sides of the bars
height	the heights of the bars

Optional keyword arguments:

Keyword	Description
width	the widths of the bars
bottom	the y coordinates of the bottom edges of the bars
color	the colors of the bars
edgecolor	the colors of the bar edges
linewidth	width of bar edges; None means use default linewidth; 0 means don't draw edges.
xerr	if not None, will be used to generate errorbars on the bar chart
yerr	if not None, will be used to generate errorbars on the bar chart
ecolor	specifies the color of any errorbar
capsize	(default 3) determines the length in points of the error bar caps
error_kw	dictionary of kwargs to be passed to errorbar method. *ecolor* and *capsize* may be specified here rather than as independent kwargs.
align	'edge' (default) | 'center'
orientation	'vertical' | 'horizontal'
log	[False|True] False (default) leaves the orientation axis as-is; True sets it to log scale

For vertical bars, *align* = 'edge' aligns bars by their left edges in left, while *align* = 'center' interprets these values as the *x* coordinates of the bar centers. For horizontal bars, *align* = 'edge' aligns bars by their bottom edges in bottom, while *align* = 'center' interprets these values as the *y* coordinates of the bar centers.

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:

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
clip_box	a matplotlib.transforms.Bbox instance
clip_on	[True | False]
clip_path	[(Path, Transform) | Patch | None]
color	matplotlib color spec
contains	a callable function
edgecolor or ec	mpl color spec, or None for default, or 'none' for no color
facecolor or fc	mpl color spec, or None for default, or 'none' for no color
figure	a matplotlib.figure.Figure instance
fill	[True | False]
gid	an id string
hatch	['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
label	string or anything printable with '%s' conversion.
linestyle or ls	['solid' | 'dashed' | 'dashdot' | 'dotted']
linewidth or lw	float or None for default
lod	[True | False]
path_effects	unknown
picker	[None|float|boolean|callable]
rasterized	[True | False | None]
snap	unknown
transform	Transform instance
url	a url string
visible	[True | False]
zorder	any number

Example: A stacked bar chart.

Additional kwargs: hold = [True|False] overrides default hold state

matplotlib.pyplot.**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 analagous 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 analagous 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 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.

linewidths and edgecolors can be used to customize the barb. Additional `PolyCollection` keyword arguments:

Property	Description
`agg_filter`	unknown
`alpha`	float or None
`animated`	[True | False]
`antialiased` or antialiaseds	Boolean or sequence of booleans
`array`	unknown
`axes`	an `Axes` instance
`clim`	a length 2 sequence of floats

Continued on next page

Table 68.4 – continued from previous page

Property	Description
clip_box	a matplotlib.transforms.Bbox instance
clip_on	[True | False]
clip_path	[(Path, Transform) | Patch | None]
cmap	a colormap or registered colormap name
color	matplotlib color arg or sequence of rgba tuples
colorbar	unknown
contains	a callable function
edgecolor or edgecolors	matplotlib color arg or sequence of rgba tuples
facecolor or facecolors	matplotlib color arg or sequence of rgba tuples
figure	a matplotlib.figure.Figure instance
gid	an id string
hatch	['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
label	string or anything printable with '%s' conversion.
linestyle or linestyles or dashes	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
linewidth or lw or linewidths	float or sequence of floats
lod	[True | False]
norm	unknown
offset_position	unknown
offsets	float or sequence of floats
paths	unknown
picker	[None|float|boolean|callable]
pickradius	unknown
rasterized	[True | False | None]
snap	unknown
transform	Transform instance
url	a url string
urls	unknown
visible	[True | False]
zorder	any number

Example:

Additional kwargs: hold = [True|False] overrides default hold state

matplotlib.pyplot.**barh**(*bottom*, *width*, *height=0.8*, *left=None*, *hold=None*, ***kwargs*)

Make a horizontal bar plot.

Call signature:

```
barh(bottom, width, height=0.8, left=0, **kwargs)
```

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

Return value is a list of `matplotlib.patches.Rectangle` instances.

Required arguments:

Argument	Description
bottom	the vertical positions of the bottom edges of the bars
width	the lengths of the bars

Optional keyword arguments:

Keyword	Description
height	the heights (thicknesses) of the bars
left	the x coordinates of the left edges of the bars
color	the colors of the bars
edgecolor	the colors of the bar edges
linewidth	width of bar edges; None means use default linewidth; 0 means don't draw edges.
xerr	if not None, will be used to generate errorbars on the bar chart
yerr	if not None, will be used to generate errorbars on the bar chart
ecolor	specifies the color of any errorbar
capsize	(default 3) determines the length in points of the error bar caps
align	'edge' (default) | 'center'
log	[False|True] False (default) leaves the horizontal axis as-is; True sets it to log scale

Setting *align* = 'edge' aligns bars by their bottom edges in bottom, while *align* = 'center' interprets these values as the *y* coordinates of the bar centers.

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 barh as the basis for stacked bar charts, or candlestick plots.

other optional kwargs:

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
clip_box	a matplotlib.transforms.Bbox instance
clip_on	[True | False]
clip_path	[(Path, Transform) | Patch | None]
color	matplotlib color spec
contains	a callable function
edgecolor or ec	mpl color spec, or None for default, or 'none' for no color
facecolor or fc	mpl color spec, or None for default, or 'none' for no color
figure	a matplotlib.figure.Figure instance
fill	[True | False]
gid	an id string
hatch	['/' | '\\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
label	string or anything printable with '%s' conversion.
linestyle or ls	['solid' | 'dashed' | 'dashdot' | 'dotted']
linewidth or lw	float or None for default
lod	[True | False]
path_effects	unknown
picker	[None|float|boolean|callable]
rasterized	[True | False | None]
snap	unknown
transform	Transform instance
url	a url string
visible	[True | False]
zorder	any number

Additional kwargs: hold = [True|False] overrides default hold state

`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=False, sym='b+', vert=True, whis=1.5, positions=None, widths=None, patch_artist=False, bootstrap=None, usermedians=None, conf_intervals=None, hold=None*`)`
Make a box and whisker plot.

Call signature:

```
boxplot(x, notch=False, sym='+', vert=True, whis=1.5,
        positions=None, widths=None, patch_artist=False,
        bootstrap=None, usermedians=None, conf_intervals=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.

Function Arguments:

x **:** Array or a sequence of vectors.

notch [[False (default) | True]] If False (default), produces a rectangular box plot. If True, will produce a notched box plot

sym [[default 'b+']] The default symbol for flier points. Enter an empty string ('') if you don't want to show fliers.

vert [[False | True (default)]] If True (default), makes the boxes vertical. If False, makes horizontal boxes.

whis [[default 1.5]] Defines the length of the whiskers as a function of the inner quartile range. They extend to the most extreme data point within (`whis*(75%-25%)`) data range.

bootstrap [[*None* (default) | integer]] Specifies whether to bootstrap the confidence intervals around the median for notched boxplots. If bootstrap==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 it's 95% confidence intervals. Values between 1000 and 10000 are recommended.

usermedians [[default None]] 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* == None, the median will be computed directly as normal.

conf_intervals [[default None]] Array or sequence whose first dimension (or length) is compatible with *x* and whose second dimension is 2. When the current element of *conf_intervals* is not None, the notch locations computed by matplotlib are overridden (assuming notch is True). When an element of *conf_intervals* is None, boxplot compute notches the method specified by the other kwargs (e.g. *bootstrap*).

positions [[default 1,2,...,n]] Sets the horizontal positions of the boxes. The ticks and limits are automatically set to match the positions.

widths [[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.

patch_artist [[False (default) | True]] If False produces boxes with the Line2D artist If True produces boxes with the Patch artist

Returns 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: horizonal lines at the median of each box.
- whiskers: the vertical lines extending to the most extreme, n-outlier data points.
- caps: the horizontal lines at the ends of the whiskers.
- fliers: points representing data that extend beyone the whiskers (outliers).

Example:

Additional kwargs: hold = [True|False] overrides default hold state

`matplotlib.pyplot.`**`broken_barh`**(*xranges*, *yrange*, *hold=None*, ***kwargs*)
Plot horizontal bars.

Call signature:

```
broken_barh(self, xranges, yrange, **kwargs)
```

A collection of horizontal bars spanning *yrange* with a sequence of *xranges*.

Required arguments:

Argument	Description
xranges	sequence of (*xmin*, *xwidth*)
yrange	sequence of (*ymin*, *ywidth*)

kwargs are `matplotlib.collections.BrokenBarHCollection` properties:

Property	Description
`agg_filter`	unknown
`alpha`	float or None
`animated`	[True | False]
`antialiased` or antialiaseds	Boolean or sequence of booleans
`array`	unknown
`axes`	an `Axes` instance
`clim`	a length 2 sequence of floats

Continued on next page

Table 68.5 – continued from previous page

Property	Description
clip_box	a matplotlib.transforms.Bbox instance
clip_on	[True | False]
clip_path	[(Path, Transform) | Patch | None]
cmap	a colormap or registered colormap name
color	matplotlib color arg or sequence of rgba tuples
colorbar	unknown
contains	a callable function
edgecolor or edgecolors	matplotlib color arg or sequence of rgba tuples
facecolor or facecolors	matplotlib color arg or sequence of rgba tuples
figure	a matplotlib.figure.Figure instance
gid	an id string
hatch	['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
label	string or anything printable with '%s' conversion.
linestyle or linestyles or dashes	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
linewidth or lw or linewidths	float or sequence of floats
lod	[True | False]
norm	unknown
offset_position	unknown
offsets	float or sequence of floats
paths	unknown
picker	[None|float|boolean|callable]
pickradius	unknown
rasterized	[True | False | None]
snap	unknown
transform	Transform instance
url	a url string
urls	unknown
visible	[True | False]
zorder	any number

these can either be a single argument, ie:

```
facecolors = 'black'
```

or a sequence of arguments for the various bars, ie:

```
facecolors = ('black', 'red', 'green')
```

Example:

Additional kwargs: hold = [True|False] overrides default hold state

`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 eg '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.

Additional kwargs: hold = [True|False] overrides default hold state

`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 at 0x9bc0a3c>*, *window=<function window_hanning at 0x9bc0924>*, *noverlap=0*, *pad_to=None*, *sides='default'*, *scale_by_freq=None*, *hold=None*, ***kwargs*`)`

Plot the coherence between *x* and *y*.

Call signature:

```
cohere(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, **kwargs)
```

Plot the coherence between *x* and *y*. Coherence is the normalized cross spectral density:

$$C_{xy} = \frac{|P_{xy}|^2}{P_{xx}P_{yy}} \tag{68.1}$$

Keyword arguments:

***NFFT*: integer** The number of data points used in each block for the FFT. Must be even; 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.

***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.

***detrend*: 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.

***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.

***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 psd (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*

***sides*: ['default' | 'onesided' | 'twosided']** Specifies which sides of the PSD 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 PSD, while 'twosided' forces two-sided.

***scale_by_freq*: boolean** 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.

The return value is a tuple (Cxy, f), where f are the frequencies of the coherence vector.

kwargs are applied to the lines.

References:

- Bendat & Piersol – Random Data: Analysis and Measurement Procedures, John Wiley & Sons (1986)

kwargs control the `Line2D` properties of the coherence plot:

Property	Description
`agg_filter`	unknown
`alpha`	float (0.0 transparent through 1.0 opaque)
`animated`	[True | False]
`antialiased` or aa	[True | False]
`axes`	an `Axes` instance
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color` or c	any matplotlib color
`contains`	a callable function
`dash_capstyle`	['butt' | 'round' | 'projecting']
`dash_joinstyle`	['miter' | 'round' | 'bevel']
`dashes`	sequence of on/off ink in points
`data`	2D array (rows are x, y) or two 1D arrays
`drawstyle`	['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post']
`figure`	a `matplotlib.figure.Figure` instance
`fillstyle`	['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']
`gid`	an id string
`label`	string or anything printable with '%s' conversion.
`linestyle` or ls	[`'-'` | `'--'` | `'-.'` | `':'` | `'None'` | `' '` | ''] and any drawstyle in combina
`linewidth` or lw	float value in points
`lod`	[True | False]
`marker`	[`7` | `4` | `5` | `6` | `'o'` | `'D'` | `'h'` | `'H'` | `'_'` | '' | `'None'` | `None` | `' '` | `'8'` | `'p`
`markeredgecolor` or mec	any matplotlib color

Table 68.6

Property	Description
markeredgewidth or mew	float value in points
markerfacecolor or mfc	any matplotlib color
markerfacecoloralt or mfcalt	any matplotlib color
markersize or ms	float
markevery	None | integer | (startind, stride)
picker	float distance in points or callable pick function fn(artist, event)
pickradius	float distance in points
rasterized	[True | False | None]
snap	unknown
solid_capstyle	['butt' | 'round' | 'projecting']
solid_joinstyle	['miter' | 'round' | 'bevel']
transform	a matplotlib.transforms.Transform instance
url	a url string
visible	[True | False]
xdata	1D array
ydata	1D array
zorder	any number

Example:

Additional kwargs: hold = [True|False] overrides default hold state

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 from which space for a new colorbar axes will be stolen

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:

Property	Description
orientation	vertical or horizontal
fraction	0.15; fraction of original axes to use for colorbar
pad	0.05 if vertical, 0.15 if horizontal; fraction of original axes between colorbar and new image axes
shrink	1.0; fraction by which to shrink the colorbar
aspect	20; ratio of long to short dimensions
anchor	(0.0, 0.5) if vertical; (0.5, 1.0) if horizontal; the anchor point of the colorbar axes
panchor	(1.0, 0.5) if vertical; (0.5, 0.0) if horizontal; the anchor point of the colorbar parent axes

colorbar properties:

Property	Description
extend	['neither' | 'both' | 'min' | 'max'] If not 'neither', make pointed end(s) for out-of- range values. These are set for a given colormap using the colormap set_under and set_over methods.
extendfrac	[*None* | 'auto' | length | lengths] If set to *None*, both the minimum and maximum triangular colorbar extensions with have a length of 5% of the interior colorbar length (this is the default setting). If set to 'auto', makes the triangular colorbar extensions the same lengths as the interior boxes (when *spacing* is set to 'uniform') or the same lengths as the respective adjacent interior boxes (when *spacing* is set to 'proportional'). If a scalar, indicates the length of both the minimum and maximum triangular colorbar extensions as a fraction of the interior colorbar length. A two-element sequence of fractions may also be given, indicating the lengths of the minimum and maximum colorbar extensions respectively as a fraction of the interior colorbar length.
spacing	['uniform' | 'proportional'] Uniform spacing gives each discrete color the same space; proportional makes the space proportional to the data interval.
ticks	[None | list of ticks | Locator object] If None, ticks are determined automatically from the input.
format	[None | format string | Formatter object] If None, the `ScalarFormatter` is used. If a format string is given, e.g. '%.3f', that is used. An alternative `Formatter` object may be given instead.
drawedges	[False | True] If true, draw lines at color boundaries.

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.

Property	Description
boundaries	None or a sequence
values	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.

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:

```
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.

`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 builtin colors, you can use a single letter

Alias	Color
'b'	blue
'g'	green
'r'	red
'c'	cyan
'm'	magenta
'y'	yellow
'k'	black
'w'	white

For a greater range of colors, you have two options. You can specify the color using an html hex string, as in:

```
color = '#eeefff'
```

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:

```
color = 'red'
color = 'burlywood'
color = 'chartreuse'
```

The example below creates a subplot with a dark slate gray background:

```
subplot(111, axisbg=(0.1843, 0.3098, 0.3098))
```

Here is an example that creates a pale turqoise title:

```
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:

```
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:

```
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)
```

contour *N* automatically-chosen levels.

```
contour(Z,V)
contour(X,Y,Z,V)
```

draw contour lines at the values specified in sequence *V*

```
contourf(..., V)
```

fill the `len(V)-1` regions between the values in *V*

```
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:

***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; eg 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 rcParams['lines.antialiased'].

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:

***nchunk*: [0 | integer]** If 0, no subdivision of the domain. Specify a positive integer to divide the domain into subdomains of roughly *nchunk* by *nchunk* points. This may never actually be advantageous, so this option may be removed. Chunking introduces artifacts at the chunk boundaries unless *antialiased* is *False*.

***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 *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:

Additional kwargs: hold = [True|False] overrides default hold state

`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 *N* automatically-chosen levels.

```
contour(Z,V)
contour(X,Y,Z,V)
```

draw contour lines at the values specified in sequence *V*

```
contourf(..., V)
```

fill the `len(V)-1` regions between the values in *V*

```
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:

***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; eg 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 rcParams['lines.antialiased'].

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:

***nchunk*: [0 | integer]** If 0, no subdivision of the domain. Specify a positive integer to divide the domain into subdomains of roughly *nchunk* by *nchunk* points. This may never actually be advantageous, so this option may be removed. Chunking introduces artifacts at the chunk boundaries unless *antialiased* is *False*.

***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 *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:

Additional kwargs: hold = [True|False] overrides default hold state

`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=256*, *Fs=2*, *Fc=0*, *detrend=<function detrend_none at 0x9bc0a3c>*, *window=<function window_hanning at 0x9bc0924>*, *noverlap=0*, *pad_to=None*, *sides='default'*, *scale_by_freq=None*, *hold=None*, ***kwargs*)

Plot 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, **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*. 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.

Returns the tuple (*Pxy*, *freqs*). *P* is the cross spectrum (complex valued), and $10\log_{10}|P_{xy}|$ is plotted.

Keyword arguments:

***NFFT*: integer** The number of data points used in each block for the FFT. Must be even; 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.

***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.

***detrend*: 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.

***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.

***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 psd (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*

***sides*: ['default' | 'onesided' | 'twosided']** Specifies which sides of the PSD 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 PSD, while 'twosided' forces two-sided.

***scale_by_freq*: boolean** 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.

References: Bendat & Piersol – Random Data: Analysis and Measurement Procedures, John Wiley & Sons (1986)

kwargs control the Line2D properties:

Property	Description
`agg_filter`	unknown
`alpha`	float (0.0 transparent through 1.0 opaque)
`animated`	[True | False]
`antialiased` or aa	[True | False]
`axes`	an `Axes` instance
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color` or c	any matplotlib color
`contains`	a callable function
`dash_capstyle`	['butt' | 'round' | 'projecting']
`dash_joinstyle`	['miter' | 'round' | 'bevel']
`dashes`	sequence of on/off ink in points
`data`	2D array (rows are x, y) or two 1D arrays
`drawstyle`	['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post']
`figure`	a `matplotlib.figure.Figure` instance
`fillstyle`	['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']
`gid`	an id string
`label`	string or anything printable with '%s' conversion.
`linestyle` or ls	[`'-'` | `'--'` | `'-.'` | `':'` | `'None'` | `' '` | "] and any drawstyle in combina
`linewidth` or lw	float value in points
`lod`	[True | False]
`marker`	[`7` | `4` | `5` | `6` | `'o'` | `'D'` | `'h'` | `'H'` | `'_'` | " | `'None'` | `None` | `' '` | `'8'` | `'p`
`markeredgecolor` or mec	any matplotlib color
`markeredgewidth` or mew	float value in points
`markerfacecolor` or mfc	any matplotlib color
`markerfacecoloralt` or mfcalt	any matplotlib color
`markersize` or ms	float
`markevery`	None | integer | (startind, stride)
`picker`	float distance in points or callable pick function `fn(artist, event)`
`pickradius`	float distance in points
`rasterized`	[True | False | None]
`snap`	unknown

Table 68.7

Property	Description
solid_capstyle	['butt' | 'round' | 'projecting']
solid_joinstyle	['miter' | 'round' | 'bevel']
transform	a matplotlib.transforms.Transform instance
url	a url string
visible	[True | False]
xdata	1D array
ydata	1D array
zorder	any number

Example:

Additional kwargs: hold = [True|False] overrides default hold state

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 in interactive mode to update a figure that has been altered using one or more plot object method calls; it is not needed if figure modification is done entirely with pyplot functions, if a sequence of modifications ends with a pyplot function, or if matplotlib is in non-interactive mode and the sequence of modifications ends with show() or savefig().

A more object-oriented alternative, given any Figure instance, fig, that was created using a pyplot function, is:

```
fig.canvas.draw()
```

matplotlib.pyplot.**errorbar**(*x, y, yerr=None, xerr=None, fmt='-', ecolor=None, elinewidth=None, capsize=3, barsabove=False, lolims=False, uplims=False, xlolims=False, xuplims=False, errorevery=1, capthick=None, hold=None, **kwargs*)

Plot an errorbar graph.

Call signature:

```
errorbar(x, y, yerr=None, xerr=None,
         fmt='-', ecolor=None, elinewidth=None, capsize=3,
         barsabove=False, lolims=False, uplims=False,
         xlolims=False, xuplims=False, errorevery=1,
         capthick=None)
```

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*.

Optional keyword arguments:

***xerr/yerr*: [scalar | N, Nx1, or 2xN array-like]** If a scalar number, len(N) array-like object, or an Nx1 array-like object, errorbars are drawn +/- value.

If a sequence of shape 2xN, errorbars are drawn at -row1 and +row2

***fmt*: '-'** The plot format symbol. If *fmt* is *None*, only the errorbars are plotted. This is used for adding errorbars to a bar plot, for example.

***ecolor*: [*None* | mpl color]** A matplotlib color arg which gives the color the errorbar lines; if *None*, use the marker color.

***elinewidth*: scalar** The linewidth of the errorbar lines. If *None*, use the linewidth.

***capsize*: scalar** The length of the error bar caps in points

***capthick*: scalar** 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*: [*True* | *False*]** if *True*, will plot the errorbars above the plot symbols. Default is below.

***lolims* / *uplims* / *xlolims* / *xuplims*: [*False* | *True*]** 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*.

***errorevery*: positive integer** subsamples the errorbars. Eg if everyerror=5, errorbars for every 5-th datapoint will be plotted. The data plot itself still shows all data points.

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:

```
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 *markeredgewith*.

valid kwargs for the marker properties are

Property	Description
agg_filter	unknown
alpha	float (0.0 transparent through 1.0 opaque)
animated	[True | False]
antialiased or aa	[True | False]
axes	an Axes instance
clip_box	a matplotlib.transforms.Bbox instance
clip_on	[True | False]
clip_path	[(Path, Transform) | Patch | None]
color or c	any matplotlib color
contains	a callable function
dash_capstyle	['butt' | 'round' | 'projecting']
dash_joinstyle	['miter' | 'round' | 'bevel']
dashes	sequence of on/off ink in points
data	2D array (rows are x, y) or two 1D arrays
drawstyle	['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post']
figure	a matplotlib.figure.Figure instance
fillstyle	['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']
gid	an id string
label	string or anything printable with '%s' conversion.
linestyle or ls	['-' | '--' | '-.' | ':' | 'None' | ' ' | "] and any drawstyle in combina
linewidth or lw	float value in points
lod	[True | False]
marker	[7 | 4 | 5 | 6 | 'o' | 'D' | 'h' | 'H' | '_' | " | 'None' | None | ' ' | '8' | 'p
markeredgecolor or mec	any matplotlib color
markeredgewidth or mew	float value in points
markerfacecolor or mfc	any matplotlib color
markerfacecoloralt or mfcalt	any matplotlib color
markersize or ms	float
markevery	None | integer | (startind, stride)
picker	float distance in points or callable pick function fn(artist, event)
pickradius	float distance in points
rasterized	[True | False | None]
snap	unknown

Table 68.8

Property	Description
solid_capstyle	['butt' | 'round' | 'projecting']
solid_joinstyle	['miter' | 'round' | 'bevel']
transform	a matplotlib.transforms.Transform instance
url	a url string
visible	[True | False]
xdata	1D array
ydata	1D array
zorder	any number

Returns (*plotline*, *caplines*, *barlinecols*):

***plotline*: Line2D instance** *x*, *y* plot markers and/or line

***caplines*: list of error bar cap** Line2D instances

***barlinecols*: list of** LineCollection instances for the horizontal and vertical error ranges.

Example:

Additional kwargs: hold = [True|False] overrides default hold state

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:

Keyword	Description
xo or yo	An integer, the *x* and *y* image offset in pixels
cmap	a `matplotlib.colors.Colormap` instance, eg cm.jet. If *None*, default to the rc `image.cmap` value
norm	a `matplotlib.colors.Normalize` instance. The default is normalization(). This scales luminance -> 0-1
vmin|vmax	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.
alpha	the alpha blending value, default is *None*
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

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 size [0,1,0,1].

An `matplotlib.image.FigureImage` instance is returned.

Additional kwargs are Artist kwargs passed on to :class:'~matplotlib.image.FigureImage'Addition kwargs: hold = [True|False] overrides default hold state

`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:

```
figlegend( (line1, line2, line3),
           ('label1', 'label2', 'label3'),
           'upper right' )
```

See Also:

`legend()`

`matplotlib.pyplot.`**`figtext`**(**args*, ***kwargs*)

Add text to figure.

Call signature:

```
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:

Property	Description
`agg_filter`	unknown
`alpha`	float (0.0 transparent through 1.0 opaque)
`animated`	[True | False]
`axes`	an `Axes` instance
`backgroundcolor`	any matplotlib color
`bbox`	rectangle prop dict
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color`	any matplotlib color
`contains`	a callable function
`family` or fontfamily or fontname or name	[FONTNAME | 'serif' | 'sans-serif' | 'cursive' | 'fantasy' | 'mono
`figure`	a `matplotlib.figure.Figure` instance
`fontproperties` or font_properties	a `matplotlib.font_manager.FontProperties` instance
`gid`	an id string
`horizontalalignment` or ha	['center' | 'right' | 'left']
`label`	string or anything printable with '%s' conversion.
`linespacing`	float (multiple of font size)
`lod`	[True | False]
`multialignment`	['left' | 'right' | 'center']
`path_effects`	unknown
`picker`	[None|float|boolean|callable]
`position`	(x,y)
`rasterized`	[True | False | None]
`rotation`	[angle in degrees | 'vertical' | 'horizontal']
`rotation_mode`	unknown
`size` or fontsize	[size in points | 'xx-small' | 'x-small' | 'small' | 'medium' | 'large
`snap`	unknown
`stretch` or fontstretch	[a numeric value in range 0-1000 | 'ultra-condensed' | 'extra-conc
`style` or fontstyle	['normal' | 'italic' | 'oblique']
`text`	string or anything printable with '%s' conversion.
`transform`	`Transform` instance
`url`	a url string
`variant` or fontvariant	['normal' | 'small-caps']

Table 68.9 – cont

Property	Description
verticalalignment or va or ma	['center' | 'top' | 'bottom' | 'baseline']
visible	[True | False]
weight or fontweight	[a numeric value in range 0-1000 | 'ultralight' | 'light' | 'normal' |
x	float
y	float
zorder	any number

matplotlib.pyplot.**figure**(*num=None*, *figsize=None*, *dpi=None*, *facecolor=None*, *edgecolor=None*, *frameon=True*, *FigureClass=<class 'matplotlib.figure.Figure'>*, ***kwargs*)

Create a new figure.

call signature:

```
figure(num=None, figsize=(8, 6), dpi=80, facecolor='w', edgecolor='k')
```

Create a new figure and return a `matplotlib.figure.Figure` instance. If *num* = *None*, the figure number will be incremented and a new figure will be created. The returned figure objects have a *number* attribute holding this number.

If *num* is an integer, and `figure(num)` already exists, make it active and return a reference to it. If `figure(num)` does not exist it will be created. Numbering starts at 1, MATLAB style:

```
figure(1)
```

The same applies if *num* is a string. In this case *num* will be used as an explicit figure label:

```
figure("today")
```

and in windowed backends, the window title will be set to this figure label.

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.

Optional keyword arguments:

Keyword	Description
figsize	width x height in inches; defaults to rc figure.figsize
dpi	resolution; defaults to rc figure.dpi
facecolor	the background color; defaults to rc figure.facecolor
edgecolor	the border color; defaults to rc figure.edgecolor

rcParams defines the default values, which can be modified in the matplotlibrc file

FigureClass is a `Figure` or derived class that will be passed on to `new_figure_manager()` in the backends which allows you to hook custom Figure classes into the pylab interface. Additional kwargs will be passed on to your figure init function.

`matplotlib.pyplot.fill`(**args*, ***kwargs*)

Plot filled polygons.

Call signature:

```
fill(*args, **kwargs)
```

args is a variable length argument, allowing for multiple *x*, *y* pairs with an optional color format string; see `plot()` for details on the argument parsing. For example, to plot a polygon with vertices at *x*, *y* in blue.:

```
ax.fill(x,y, 'b' )
```

An arbitrary number of *x*, *y*, *color* groups can be specified:

```
ax.fill(x1, y1, 'g', x2, y2, 'r')
```

Return value is a list of `Patch` instances that were added.

The same color strings that `plot()` supports are supported by the fill format string.

If you would like to fill below a curve, eg. shade a region between 0 and *y* along *x*, use `fill_between()`

The *closed* kwarg will close the polygon when *True* (default).

kwargs control the `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
clip_box	a matplotlib.transforms.Bbox instance
clip_on	[True | False]
clip_path	[(Path, Transform) | Patch | None]
color	matplotlib color spec
contains	a callable function
edgecolor or ec	mpl color spec, or None for default, or 'none' for no color
facecolor or fc	mpl color spec, or None for default, or 'none' for no color
figure	a matplotlib.figure.Figure instance
fill	[True | False]
gid	an id string
hatch	['/' | '\\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
label	string or anything printable with '%s' conversion.
linestyle or ls	['solid' | 'dashed' | 'dashdot' | 'dotted']
linewidth or lw	float or None for default
lod	[True | False]
path_effects	unknown
picker	[None|float|boolean|callable]
rasterized	[True | False | None]
snap	unknown
transform	Transform instance
url	a url string
visible	[True | False]
zorder	any number

Example:

Additional kwargs: hold = [True|False] overrides default hold state

matplotlib.pyplot.**fill_between**(*x*, *y1*, *y2=0*, *where=None*, *interpolate=False*, *hold=None*, ***kwargs*)

Make filled polygons between two curves.

Call signature:

```
fill_between(x, y1, y2=0, where=None, **kwargs)
```

Create a PolyCollection filling the regions between *y1* and *y2* where `where==True`

x **:** An N-length array of the x data

y1 **:** An N-length array (or scalar) of the y data

***y2* :** An N-length array (or scalar) of the y data

***where* :** 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* :** 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.

***kwargs* :** Keyword args passed on to the `PolyCollection`.

kwargs control the `Polygon` properties:

Property	Description
`agg_filter`	unknown
`alpha`	float or None
`animated`	[True | False]
`antialiased` or antialiaseds	Boolean or sequence of booleans
`array`	unknown
`axes`	an `Axes` instance
`clim`	a length 2 sequence of floats
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`cmap`	a colormap or registered colormap name
`color`	matplotlib color arg or sequence of rgba tuples
`colorbar`	unknown
`contains`	a callable function
`edgecolor` or edgecolors	matplotlib color arg or sequence of rgba tuples
`facecolor` or facecolors	matplotlib color arg or sequence of rgba tuples
`figure`	a `matplotlib.figure.Figure` instance
`gid`	an id string
`hatch`	[‘/’ | ‘\’ | ‘|’ | ‘-‘ | ‘+’ | ‘x’ | ‘o’ | ‘O’ | ‘.’ | ‘*’]
`label`	string or anything printable with ‘%s’ conversion.
`linestyle` or linestyles or dashes	[‘solid’ | ‘dashed’, ‘dashdot’, ‘dotted’ | (offset, on-off-dash-seq)]
`linewidth` or lw or linewidths	float or sequence of floats
`lod`	[True | False]
`norm`	unknown
`offset_position`	unknown
`offsets`	float or sequence of floats
`paths`	unknown
`picker`	[None|float|boolean|callable]
`pickradius`	unknown
`rasterized`	[True | False | None]

Continued on next page

Table 68.10 – continued from previous page

Property	Description
snap	unknown
transform	Transform instance
url	a url string
urls	unknown
visible	[True | False]
zorder	any number

See Also:

fill_betweenx() for filling between two sets of x-values

Additional kwargs: hold = [True|False] overrides default hold state

matplotlib.pyplot.**fill_betweenx**(*y*, *x1*, *x2=0*, *where=None*, *hold=None*, ***kwargs*)

Make filled polygons between two horizontal curves.

Call signature:

```
fill_between(y, x1, x2=0, where=None, **kwargs)
```

Create a PolyCollection filling the regions between *x1* and *x2* where `where==True`

y **:** An N-length array of the y data

x1 **:** An N-length array (or scalar) of the x data

x2 **:** An N-length array (or scalar) of the x data

where **:** 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`

kwargs **:** keyword args passed on to the PolyCollection

kwargs control the Polygon properties:

Property	Description
agg_filter	unknown
alpha	float or None
animated	[True | False]
antialiased or antialiaseds	Boolean or sequence of booleans
array	unknown
axes	an Axes instance
clim	a length 2 sequence of floats

Continued on next page

Table 68.11 – continued from previous page

Property	Description
clip_box	a matplotlib.transforms.Bbox instance
clip_on	[True | False]
clip_path	[(Path, Transform) | Patch | None]
cmap	a colormap or registered colormap name
color	matplotlib color arg or sequence of rgba tuples
colorbar	unknown
contains	a callable function
edgecolor or edgecolors	matplotlib color arg or sequence of rgba tuples
facecolor or facecolors	matplotlib color arg or sequence of rgba tuples
figure	a matplotlib.figure.Figure instance
gid	an id string
hatch	['/' | '\\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
label	string or anything printable with '%s' conversion.
linestyle or linestyles or dashes	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
linewidth or lw or linewidths	float or sequence of floats
lod	[True | False]
norm	unknown
offset_position	unknown
offsets	float or sequence of floats
paths	unknown
picker	[None|float|boolean|callable]
pickradius	unknown
rasterized	[True | False | None]
snap	unknown
transform	Transform instance
url	a url string
urls	unknown
visible	[True | False]
zorder	any number

See Also:

fill_between() for filling between two sets of y-values

Additional kwargs: hold = [True|False] overrides default hold state

matplotlib.pyplot.**findobj**(*o=None*, *match=None*)

Find artist objects.

pyplot signature: findobj(o=gcf(), match=None, include_self=True)

Recursively find all :class:matplotlib.artist.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: eg 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.pyplot.flag()`
set the default colormap to flag and apply to current image if any. See help(colormaps) for more information

`matplotlib.pyplot.gca(**kwargs)`
Return the current axis instance. This can be used to control axis properties either using set or the `Axes` methods, for example, setting the xaxis range:

```
plot(t,s)
set(gca(), 'xlim', [0,10])
```

or:

```
plot(t,s)
a = gca()
a.set_xlim([0,10])
```

`matplotlib.pyplot.gcf()`
Return a reference to the current figure.

`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.

`matplotlib.pyplot.get_current_fig_manager()`

`matplotlib.pyplot.get_figlabels()`
Return a list of existing figure labels.

`matplotlib.pyplot.get_fignums()`
Return a list of existing figure numbers.

`matplotlib.pyplot.get_plot_commands()`
Get a sorted list of all of the plotting commands.

`matplotlib.pyplot.ginput(*args, **kwargs)`
Call signature:

```
ginput(self, 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 overriden 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.

`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.

Call signature:

```
grid(self, b=None, which='major', axis='both', **kwargs)
```

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, eg:

```
ax.grid(color='r', linestyle='-', linewidth=2)
```

Valid `Line2D` kwargs are

Property	Description
agg_filter	unknown
alpha	float (0.0 transparent through 1.0 opaque)
animated	[True | False]
antialiased or aa	[True | False]
axes	an Axes instance
clip_box	a matplotlib.transforms.Bbox instance
clip_on	[True | False]
clip_path	[(Path, Transform) | Patch | None]
color or c	any matplotlib color
contains	a callable function
dash_capstyle	['butt' | 'round' | 'projecting']
dash_joinstyle	['miter' | 'round' | 'bevel']
dashes	sequence of on/off ink in points
data	2D array (rows are x, y) or two 1D arrays
drawstyle	['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post']
figure	a matplotlib.figure.Figure instance
fillstyle	['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']
gid	an id string
label	string or anything printable with '%s' conversion.
linestyle or ls	['-' | '--' | '-.' | ':' | 'None' | ' ' | ''] and any drawstyle in combina
linewidth or lw	float value in points
lod	[True | False]
marker	[7 | 4 | 5 | 6 | 'o' | 'D' | 'h' | 'H' | '_' | '' | 'None' | None | ' ' | '8' | 'p
markeredgecolor or mec	any matplotlib color
markeredgewidth or mew	float value in points
markerfacecolor or mfc	any matplotlib color
markerfacecoloralt or mfcalt	any matplotlib color
markersize or ms	float
markevery	None | integer | (startind, stride)
picker	float distance in points or callable pick function fn(artist, event)
pickradius	float distance in points
rasterized	[True | False | None]
snap	unknown
solid_capstyle	['butt' | 'round' | 'projecting']
solid_joinstyle	['miter' | 'round' | 'bevel']
transform	a matplotlib.transforms.Transform instance
url	a url string
visible	[True | False]
xdata	1D array
ydata	1D array
zorder	any number

`matplotlib.pyplot.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 at 0x8761f44>*, *mincnt=None*, *marginals=False*, *hold=None*, ***kwargs*)

Make a hexagonal binning plot.

Call signature:

```
hexbin(x, y, C = None, gridsize = 100, bins = None,
       xscale = 'linear', yscale = 'linear',
       cmap=None, norm=None, vmin=None, vmax=None,
       alpha=None, linewidths=None, edgecolors='none'
       reduce_C_function = np.mean, mincnt=None, marginals=True
       **kwargs)
```

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 occurences 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 *reduce_C_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*.)

x, *y* and/or *C* may be masked arrays, in which case only unmasked points will be plotted.

Optional keyword arguments:

***gridsize*: [100 | integer]** 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, gridsize can be a tuple with two elements specifying the number of hexagons in the *x*-direction and the *y*-direction.

***bins*: [*None* | 'log' | integer | sequence]** 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.

***xscale*: ['linear' | 'log']** Use a linear or log10 scale on the horizontal axis.

***scale*: ['linear' | 'log']** Use a linear or log10 scale on the vertical axis.

***mincnt*: [*None* | a positive integer]** If not *None*, only display cells with more than *mincnt* number of points in the cell

***marginals*: [*True* | *False*]** if marginals is *True*, plot the marginal density as colormapped rectagles along the bottom of the x-axis and left of the y-axis

***extent*: [*None* | scalars (left, right, bottom, top)]** The limits of the bins. The default assigns the limits based on gridsize, x, y, xscale and yscale.

Other keyword arguments controlling color mapping and normalization arguments:

***cmap*: [*None* | Colormap]** a `matplotlib.colors.Colormap` instance. If *None*, defaults to rc `image.cmap`.

***norm*: [*None* | Normalize]** `matplotlib.colors.Normalize` instance is used to scale luminance data to 0,1.

***vmin* / *vmax*: scalar** *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 *C* is used. Note if you pass a norm instance, your settings for *vmin* and *vmax* will be ignored.

alpha*: scalar between 0 and 1, or *None the alpha value for the patches

***linewidths*: [*None* | scalar]** If *None*, defaults to rc lines.linewidth. Note that this is a tuple, and if you set the linewidths argument you must set it as a sequence of floats, as required by `RegularPolyCollection`.

Other keyword arguments controlling the Collection properties:

***edgecolors*: [*None* | `'none'` | mpl color | color sequence]** 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 the outlines in the default color.

If a matplotlib color arg or sequence of rgba tuples, draws the outlines in the specified color.

Here are the standard descriptions of all the `Collection` kwargs:

Property	Description
`agg_filter`	unknown
`alpha`	float or None
`animated`	[True | False]
`antialiased` or antialiaseds	Boolean or sequence of booleans
`array`	unknown
`axes`	an `Axes` instance
`clim`	a length 2 sequence of floats
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]

Continued on next page

Table 68.13 – continued from previous page

Property	Description
clip_path	[(Path, Transform) | Patch | None]
cmap	a colormap or registered colormap name
color	matplotlib color arg or sequence of rgba tuples
colorbar	unknown
contains	a callable function
edgecolor or edgecolors	matplotlib color arg or sequence of rgba tuples
facecolor or facecolors	matplotlib color arg or sequence of rgba tuples
figure	a matplotlib.figure.Figure instance
gid	an id string
hatch	['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
label	string or anything printable with '%s' conversion.
linestyle or linestyles or dashes	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
linewidth or lw or linewidths	float or sequence of floats
lod	[True | False]
norm	unknown
offset_position	unknown
offsets	float or sequence of floats
paths	unknown
picker	[None|float|boolean|callable]
pickradius	unknown
rasterized	[True | False | None]
snap	unknown
transform	Transform instance
url	a url string
urls	unknown
visible	[True | False]
zorder	any number

The return value is 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*.

Example:

Additional kwargs: hold = [True|False] overrides default hold state

`matplotlib.pyplot.hist`(*x*, *bins=10*, *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*, ***kwargs*)

Plot a histogram.

Call signature:

```
hist(x, bins=10, 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)
```

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.

Keyword arguments:

***bins*:** Either an integer number of bins or a sequence giving the bins. If *bins* is an integer, *bins* + 1 bin edges will be returned, consistent with `numpy.histogram()` for numpy version >= 1.3, and with the *new* = True argument in earlier versions. Unequally spaced bins are supported if *bins* is a sequence.

***range*:** 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.

***normed*:** If *True*, the first element of the return tuple will be the counts normalized to form a probability density, i.e., `n/(len(x)*dbin)`. In a probability density, the integral of the histogram should be 1; you can verify that with a trapezoidal integration of the probability density function:

```
pdf, bins, patches = ax.hist(...)
print np.sum(pdf * np.diff(bins))
```

Note: 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

***weights*:** 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.

***cumulative*:** 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.

***histtype*: ['bar' | 'barstacked' | 'step' | 'stepfilled']** The type of histogram to draw.

- 'bar' is a traditional bar-type histogram. If multiple data are given the bars are aranged 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.

***align*: ['left' | 'mid' | 'right']** 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.

***orientation*: ['horizontal' | 'vertical']** If 'horizontal', `barh()` will be used for bar-type histograms and the *bottom* kwarg will be the left edges.

***rwidth*:** The relative width of the bars as a fraction of the bin width. If *None*, automatically compute the width. Ignored if *histtype* = 'step' or 'stepfilled'.

***log*:** 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.

***color*:** Color spec or sequence of color specs, one per dataset. Default (*None*) uses the standard line color sequence.

***label*:** 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:

```
ax.hist(10+2*np.random.randn(1000), label='men')
ax.hist(12+3*np.random.randn(1000), label='women', alpha=0.5)
ax.legend()
```

***stacked*:** If *True*, multiple data are stacked on top of each other If *False* multiple data are aranged side by side if histtype is 'bar' or on top of each other if histtype is 'step'

.

kwargs are used to update the properties of the `Patch` instances returned by *hist*:

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
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color`	matplotlib color spec
`contains`	a callable function
`edgecolor` or ec	mpl color spec, or None for default, or 'none' for no color
`facecolor` or fc	mpl color spec, or None for default, or 'none' for no color
`figure`	a `matplotlib.figure.Figure` instance
`fill`	[True | False]
`gid`	an id string
`hatch`	['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
`label`	string or anything printable with '%s' conversion.
`linestyle` or ls	['solid' | 'dashed' | 'dashdot' | 'dotted']
`linewidth` or lw	float or None for default
`lod`	[True | False]
`path_effects`	unknown
`picker`	[None|float|boolean|callable]
`rasterized`	[True | False | None]
`snap`	unknown
`transform`	`Transform` instance
`url`	a url string
`visible`	[True | False]
`zorder`	any number

Example:

Additional kwargs: hold = [True|False] overrides default hold state

`matplotlib.pyplot.`**`hist2d`**(*x*, *y*, *bins=10*, *range=None*, *normed=False*, *weights=None*, *cmin=None*, *cmax=None*, *hold=None*, ***kwargs*)

Make a 2D histogram plot.

Call signature:

```
hist2d(x, y, bins = None, range=None, weights=None, cmin=None, cmax=None **kwargs)
```

Make a 2d histogram plot of *x* versus *y*, where *x*, *y* are 1-D sequences of the same length.

The return value is `(counts, xedges, yedges, Image)`.

Optional keyword arguments: *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*: [*None* | array_like shape(2,2)]** 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*:[True|False]** Normalize histogram. The default value is False

***weights*: [*None* | array]** An array of values w_i weighing each sample (x_i, y_i).

cmin [[None| scalar]] 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 [[None| scalar]] 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

Remaining keyword arguments are passed directly to `pcolorfast()`.

Rendering the histogram with a logarithmic color scale is accomplished by passing a `colors.LogNorm` instance to the *norm* keyword argument.

Example:

Additional kwargs: hold = [True|False] overrides default hold state

`matplotlib.pyplot.`**`hlines`**(*y*, *xmin*, *xmax*, *colors='k'*, *linestyles='solid'*, *label=''*, *hold=None*, ***kwargs*)

Plot horizontal lines.

call signature:

```
hlines(y, xmin, xmax, colors='k', linestyles='solid', **kwargs)
```

Plot horizontal lines at each *y* from *xmin* to *xmax*.

Returns the `LineCollection` that was added.

Required arguments:

***y*:** a 1-D numpy array or iterable.

***xmin* and *xmax*:** can be scalars or `len(x)` numpy arrays. If they are scalars, then the respective values are constant, else the widths of the lines are determined by *xmin* and *xmax*.

Optional keyword arguments:

***colors*:** a line collections color argument, either a single color or a `len(y)` list of colors

***linestyles*:** ['solid' | 'dashed' | 'dashdot' | 'dotted']

Example:

Additional kwargs: hold = [True|False] overrides default hold state

`matplotlib.pyplot.hold(b=None)`

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(True)  # hold is on
hold(False) # hold is off
```

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.

`matplotlib.pyplot.hot()`

set the default colormap to hot and apply to current image if any. See help(colormaps) for more information

`matplotlib.pyplot.hsv()`

set the default colormap to hsv and apply to current image if any. See help(colormaps) for more information

`matplotlib.pyplot.imread(*args, **kwargs)`

Read an image from a file into an array.

fname may be a string path 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()`.

matplotlib.pyplot.**imsave**(**args*, ***kwargs*)

Save an array as in 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, eg 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.

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*, ***kwargs*)

Display an image on the axes.

Call signature:

```
imshow(X, cmap=None, norm=None, aspect=None, interpolation=None,
       alpha=None, vmin=None, vmax=None, origin=None, extent=None,
       **kwargs)
```

Display the image in *X* to current axes. *X* may be a float array, a uint8 array or a PIL image. If *X* is an array, *X* can have the following shapes:

- MxN – luminance (grayscale, float array only)
- MxNx3 – RGB (float or uint8 array)

- MxNx4 – RGBA (float or uint8 array)

The value for each component of MxNx3 and MxNx4 float arrays should be in the range 0.0 to 1.0; MxN float arrays may be normalised.

An `matplotlib.image.AxesImage` instance is returned.

Keyword arguments:

***cmap*: [*None* | Colormap]** A `matplotlib.colors.Colormap` instance, eg. cm.jet. If *None*, default to rc `image.cmap` value.

cmap is ignored when *X* has RGB(A) information

***aspect*: [*None* | 'auto' | 'equal' | scalar]** 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:

Acceptable values are *None*, 'none', 'nearest', 'bilinear', 'bicubic', 'spline16', 'spline36', 'hanning', 'hamming', 'hermite', 'kaiser', 'quadric', 'catrom', 'gaussian', 'bessel', 'mitchell', 'sinc', 'lanczos'

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*: [*None* | Normalize]** An `matplotlib.colors.Normalize` instance; if *None*, default is `normalization()`. This scales luminance -> 0-1

norm is only used for an MxN float array.

***vmin*/*vmax*: [*None* | scalar]** 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 *norm* is not *None*, the settings for *vmin* and *vmax* will be ignored.

***alpha*: scalar** The alpha blending value, between 0 (transparent) and 1 (opaque) or *None*

***origin*: [*None* | 'upper' | 'lower']** 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*: [*None* | scalars (left, right, bottom, top)]** Data limits for the axes. The default assigns zero-based row, column indices to the *x*, *y* centers of the pixels.

***shape*: [*None* | scalars (columns, rows)]** For raw buffer images

***filternorm*:** 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*:** The filter radius for filters that have a radius parameter, i.e. when interpolation is one of: 'sinc', 'lanczos' or 'blackman'

Additional kwargs are `Artist` properties:

Property	Description
`agg_filter`	unknown
`alpha`	float (0.0 transparent through 1.0 opaque)
`animated`	[True | False]
`axes`	an `Axes` instance
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`contains`	a callable function
`figure`	a `matplotlib.figure.Figure` instance
`gid`	an id string
`label`	string or anything printable with '%s' conversion.
`lod`	[True | False]
`picker`	[None|float|boolean|callable]
`rasterized`	[True | False | None]
`snap`	unknown
`transform`	`Transform` instance
`url`	a url string
`visible`	[True | False]
`zorder`	any number

Example:

Additional kwargs: hold = [True|False] overrides default hold state

`matplotlib.pyplot.ioff()`
Turn interactive mode off.

`matplotlib.pyplot.ion()`
Turn interactive mode on.

`matplotlib.pyplot.ishold()`
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)`
Place a legend on the current axes.

Call signature:

```
legend(*args, **kwargs)
```

Places legend at location *loc*. Labels are a sequence of strings and *loc* can be a string or an integer specifying the legend location.

To make a legend with existing lines:

```
legend()
```

`legend()` by itself will try and build a legend using the label property of the lines/patches/collections. You can set the label of a line by doing:

```
plot(x, y, label='my data')
```

or:

```
line.set_label('my data').
```

If label is set to '_nolegend_', the item will not be shown in legend.

To automatically generate the legend from labels:

```
legend( ('label1', 'label2', 'label3') )
```

To make a legend for a list of lines and labels:

```
legend( (line1, line2, line3), ('label1', 'label2', 'label3') )
```

To make a legend at a given location, using a location argument:

```
legend( ('label1', 'label2', 'label3'), loc='upper left')
```

or:

```
legend( (line1, line2, line3),  ('label1', 'label2', 'label3'), loc=2)
```

The location codes are

Location String	Location Code
'best'	0
'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

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. For example,

> loc = 'upper right', bbox_to_anchor = (0.5, 0.5)

will place the legend so that the upper right corner of the legend at the center of the axes.

The legend location can be specified in other coordinate, by using the *bbox_transform* keyword.

The loc itslef can be a 2-tuple giving x,y of the lower-left corner of the legend in axes coords (*bbox_to_anchor* is ignored).

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.

fontsize: [size in points | 'xx-small' | 'x-small' | 'small' | 'medium' | 'large' | 'x-large' | 'xx-large']

> Set the font size. May be either a size string, relative to the default font size, or an absolute font size in points. This argument is only used if prop is not specified.

***numpoints*: integer** The number of points in the legend for line

***scatterpoints*: integer** The number of points in the legend for scatter plot

***scatteroffsets*: 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.

***frameon*: [*True* | *False*]** if *True*, draw a frame around the legend. The default is set by the rcParam 'legend.frameon'

***fancybox*: [*None* | *False* | *True*]** if *True*, draw a frame with a round fancybox. If *None*, use rc settings

***shadow*: [*None* | *False* | *True*]** If *True*, draw a shadow behind legend. If *None*, use rc settings.

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*)

bbox_to_anchor [an instance of BboxBase or a tuple of 2 or 4 floats] the bbox that the legend will be anchored.

bbox_transform [[an instance of Transform | *None*]] the transform for the bbox. transAxes if *None*.

title [string] the legend title

Padding and spacing between various elements use following keywords parameters. These values are measure 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.

Keyword	Description
borderpad	the fractional whitespace inside the legend border
labelspacing	the vertical space between the legend entries
handlelength	the length of the legend handles
handletextpad	the pad between the legend handle and text
borderaxespad	the pad between the axes and legend border
columnspacing	the spacing between columns

Note: Not all kinds of artist are supported by the legend command. See LINK (FIXME) for details.

Example:

See Also:

Legend guide.

`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.

Call signature:

```
loglog(*args, **kwargs)
```

`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 kwargs are `Line2D` properties:

Property	Description
`agg_filter`	unknown
`alpha`	float (0.0 transparent through 1.0 opaque)
`animated`	[True | False]
`antialiased` or aa	[True | False]
`axes`	an `Axes` instance
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color` or c	any matplotlib color
`contains`	a callable function
`dash_capstyle`	['butt' | 'round' | 'projecting']
`dash_joinstyle`	['miter' | 'round' | 'bevel']

Table 68.1

Property	Description
dashes	sequence of on/off ink in points
data	2D array (rows are x, y) or two 1D arrays
drawstyle	['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post']
figure	a matplotlib.figure.Figure instance
fillstyle	['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']
gid	an id string
label	string or anything printable with '%s' conversion.
linestyle or ls	['-' | '--' | '-.' | ':' | 'None' | ' ' | "] and any drawstyle in combina
linewidth or lw	float value in points
lod	[True | False]
marker	[7 | 4 | 5 | 6 | 'o' | 'D' | 'h' | 'H' | '_' | " | 'None' | None | ' ' | '8' | 'p
markeredgecolor or mec	any matplotlib color
markeredgewidth or mew	float value in points
markerfacecolor or mfc	any matplotlib color
markerfacecoloralt or mfcalt	any matplotlib color
markersize or ms	float
markevery	None | integer | (startind, stride)
picker	float distance in points or callable pick function fn(artist, event)
pickradius	float distance in points
rasterized	[True | False | None]
snap	unknown
solid_capstyle	['butt' | 'round' | 'projecting']
solid_joinstyle	['miter' | 'round' | 'bevel']
transform	a matplotlib.transforms.Transform instance
url	a url string
visible	[True | False]
xdata	1D array
ydata	1D array
zorder	any number

Example:

Additional kwargs: hold = [True|False] overrides default hold state

`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.

`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.

`matplotlib.pyplot.minorticks_off`()

Remove minor ticks from the current plot.

`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.

`matplotlib.pyplot.over`(*func*, **args*, ***kwargs*)

Call a function with hold(True).

Calls:

```
func(*args, **kwargs)
```

with `hold(True)` and then restores the hold state.

`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.

`matplotlib.pyplot.pcolor(*args, **kwargs)`

Create a pseudocolor plot of a 2-D array.

Call signatures:

```
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 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

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*).

Similarly for `meshgrid()`:

```
x = np.arange(5)
y = np.arange(3)
X, Y = meshgrid(x,y)
```

is equivalent to:

```
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:

```
C = rand( len(x), len(y))
```

then you need:

```
pcolor(X, Y, C.T)
```

or:

```
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:

Property	Description
`agg_filter`	unknown
`alpha`	float or None
`animated`	[True | False]
`antialiased` or antialiaseds	Boolean or sequence of booleans
`array`	unknown
`axes`	an `Axes` instance
`clim`	a length 2 sequence of floats
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`cmap`	a colormap or registered colormap name
`color`	matplotlib color arg or sequence of rgba tuples
`colorbar`	unknown
`contains`	a callable function
`edgecolor` or edgecolors	matplotlib color arg or sequence of rgba tuples
`facecolor` or facecolors	matplotlib color arg or sequence of rgba tuples
`figure`	a `matplotlib.figure.Figure` instance
`gid`	an id string
`hatch`	['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
`label`	string or anything printable with '%s' conversion.
`linestyle` or linestyles or dashes	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
`linewidth` or lw or linewidths	float or sequence of floats
`lod`	[True | False]
`norm`	unknown
`offset_position`	unknown
`offsets`	float or sequence of floats
`paths`	unknown

Continued on next page

Table 68.15 – continued from previous page

Property	Description
picker	[None|float|boolean|callable]
pickradius	unknown
rasterized	[True | False | None]
snap	unknown
transform	Transform instance
url	a url string
urls	unknown
visible	[True | False]
zorder	any number

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.

Additional kwargs: hold = [True|False] overrides default hold state

`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:

Property	Description
`agg_filter`	unknown
`alpha`	float or None
`animated`	[True | False]
`antialiased` or antialiaseds	Boolean or sequence of booleans
`array`	unknown
`axes`	an `Axes` instance
`clim`	a length 2 sequence of floats
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`cmap`	a colormap or registered colormap name
`color`	matplotlib color arg or sequence of rgba tuples
`colorbar`	unknown
`contains`	a callable function
`edgecolor` or edgecolors	matplotlib color arg or sequence of rgba tuples
`facecolor` or facecolors	matplotlib color arg or sequence of rgba tuples
`figure`	a `matplotlib.figure.Figure` instance
`gid`	an id string
`hatch`	['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
`label`	string or anything printable with '%s' conversion.
`linestyle` or linestyles or dashes	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
`linewidth` or lw or linewidths	float or sequence of floats

Continued on next page

Table 68.16 – continued from previous page

Property	Description
lod	[True | False]
norm	unknown
offset_position	unknown
offsets	float or sequence of floats
paths	unknown
picker	[None|float|boolean|callable]
pickradius	unknown
rasterized	[True | False | None]
snap	unknown
transform	Transform instance
url	a url string
urls	unknown
visible	[True | False]
zorder	any number

See Also:

pcolor() For an explanation of the grid orientation and the expansion of 1-D *X* and/or *Y* to 2-D arrays.

Additional kwargs: hold = [True|False] overrides default hold state

matplotlib.pyplot.**pie**(*x*, *explode=None*, *labels=None*, *colors=None*, *autopct=None*, *pctdistance=0.6*, *shadow=False*, *labeldistance=1.1*, *startangle=None*, *radius=None*, *hold=None*)

Plot a pie chart.

Call signature:

```
pie(x, explode=None, labels=None,
    colors=('b', 'g', 'r', 'c', 'm', 'y', 'k', 'w'),
    autopct=None, pctdistance=0.6, shadow=False,
    labeldistance=1.1, startangle=None, radius=None)
```

Make 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.

Keyword arguments:

***explode*: [*None* | len(x) sequence]** If not *None*, is a `len(x)` array which specifies the fraction of the radius with which to offset each wedge.

***colors*: [*None* | color sequence]** A sequence of matplotlib color args through which the pie chart will cycle.

***labels*: [*None* | len(x) sequence of strings]** A sequence of strings providing the labels for each wedge

***autopct*: [*None* | format string | format function]** 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*: scalar** The ratio between the center of each pie slice and the start of the text generated by *autopct*. Ignored if *autopct* is *None*; default is 0.6.

***labeldistance*: scalar** The radial distance at which the pie labels are drawn

***shadow*: [*False* | *True*]** Draw a shadow beneath the pie.

***startangle*: [*None* | Offset angle]** If not *None*, rotates the start of the pie chart by *angle* degrees counterclockwise from the x-axis.

radius: [*None* | scalar] The radius of the pie, if *radius* is *None* it will be set to 1.

The pie chart will probably look best if the figure and axes are square. Eg.:

```
figure(figsize=(8,8))
ax = axes([0.1, 0.1, 0.8, 0.8])
```

Return value: If *autopct* is *None*, return the tuple (*patches*, *texts*):

- *patches* is a sequence of `matplotlib.patches.Wedge` instances
- *texts* is a list of the label `matplotlib.text.Text` instances.

If *autopct* is not *None*, return the tuple (*patches*, *texts*, *autotexts*), where *patches* and *texts* are as above, and *autotexts* is a list of `Text` instances for the numeric labels.

Additional kwargs: hold = [True|False] overrides default hold state

`matplotlib.pyplot.pink()`
set the default colormap to pink 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.

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 color specified by a 'color cycle'. To change this behavior, you can edit the axes.color_cycle rcParam. Alternatively, you can use `set_default_color_cycle()`.

The following format string characters are accepted to control the line style or marker:

character	description
`'-'`	solid line style
`'--'`	dashed line style
`'-.'`	dash-dot line style
`':'`	dotted line style
`'.'`	point marker
`','`	pixel marker
`'o'`	circle marker
`'v'`	triangle_down marker
`'^'`	triangle_up marker
`'<'`	triangle_left marker
`'>'`	triangle_right marker
`'1'`	tri_down marker
`'2'`	tri_up marker
`'3'`	tri_left marker
`'4'`	tri_right marker
`'s'`	square marker
`'p'`	pentagon marker
`'*'`	star marker
`'h'`	hexagon1 marker
`'H'`	hexagon2 marker
`'+'`	plus marker
`'x'`	x marker
`'D'`	diamond marker
`'d'`	thin_diamond marker
`'|'`	vline marker
`'_'`	hline marker

The following color abbreviations are supported:

character	color
‘b’	blue
‘g’	green
‘r’	red
‘c’	cyan
‘m’	magenta
‘y’	yellow
‘k’	black
‘w’	white

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:

```
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.:

```
plot(x1, y1, x2, y2, antialised=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 markercolor with:

```
plot(x, y, color='green', linestyle='dashed', marker='o',
     markerfacecolor='blue', markersize=12).
```

See `Line2D` for details.

The kwargs are `Line2D` properties:

Property	Description
`agg_filter`	unknown
`alpha`	float (0.0 transparent through 1.0 opaque)
`animated`	[True | False]
`antialiased` or aa	[True | False]

Table 68.1

<table>
<tr><th>Property</th><th>Description</th></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 | False]</td></tr>
<tr><td>clip_path</td><td>[(Path, Transform) | Patch | None]</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' | 'round' | 'projecting']</td></tr>
<tr><td>dash_joinstyle</td><td>['miter' | 'round' | 'bevel']</td></tr>
<tr><td>dashes</td><td>sequence of on/off ink in points</td></tr>
<tr><td>data</td><td>2D array (rows are x, y) or two 1D arrays</td></tr>
<tr><td>drawstyle</td><td>['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post']</td></tr>
<tr><td>figure</td><td>a matplotlib.figure.Figure instance</td></tr>
<tr><td>fillstyle</td><td>['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']</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>['-' | '--' | '-.' | ':' | 'None' | ' ' | "] and any drawstyle in combina</td></tr>
<tr><td>linewidth or lw</td><td>float value in points</td></tr>
<tr><td>lod</td><td>[True | False]</td></tr>
<tr><td>marker</td><td>[7 | 4 | 5 | 6 | 'o' | 'D' | 'h' | 'H' | '_' | " | 'None' | None | ' ' | '8' | 'p</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 | integer | (startind, stride)</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 | False | None]</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>
</table>

kwargs *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*.

Additional kwargs: hold = [True|False] overrides default hold state

matplotlib.pyplot.**plot_date**(*x*, *y*, *fmt='bo'*, *tz=None*, *xdate=True*, *ydate=False*, *hold=None*, ***kwargs*)

Plot with data with dates.

Call signature:

```
plot_date(x, y, fmt='bo', tz=None, xdate=True, ydate=False, **kwargs)
```

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.

Keyword arguments:

***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*: [*True* | *False*]** If *True*, the *x*-axis will be labeled with dates.

***ydate*: [*False* | *True*]** If *True*, the *y*-axis will be labeled with dates.

Note if you are using custom date tickers and formatters, it may be necessary to set the formatters/locators after the call to `plot_date()` since `plot_date()` will set the default tick locator to `matplotlib.dates.AutoDateLocator` (if the tick locator is not already set to a `matplotlib.dates.DateLocator` instance) and the default tick formatter to `matplotlib.dates.AutoDateFormatter` (if the tick formatter is not already set to a `matplotlib.dates.DateFormatter` instance).

Valid kwargs are `Line2D` properties:

Property	Description
`agg_filter`	unknown
`alpha`	float (0.0 transparent through 1.0 opaque)
`animated`	[True | False]
`antialiased` or aa	[True | False]
`axes`	an `Axes` instance
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color` or c	any matplotlib color
`contains`	a callable function
`dash_capstyle`	['butt' | 'round' | 'projecting']
`dash_joinstyle`	['miter' | 'round' | 'bevel']
`dashes`	sequence of on/off ink in points

Table 68.1

Property	Description
data	2D array (rows are x, y) or two 1D arrays
drawstyle	['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post']
figure	a matplotlib.figure.Figure instance
fillstyle	['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']
gid	an id string
label	string or anything printable with '%s' conversion.
linestyle or ls	['-' | '--' | '-.' | ':' | 'None' | ' ' | ''] and any drawstyle in combina
linewidth or lw	float value in points
lod	[True | False]
marker	[7 | 4 | 5 | 6 | 'o' | 'D' | 'h' | 'H' | '_' | '' | 'None' | None | ' ' | '8' | 'p
markeredgecolor or mec	any matplotlib color
markeredgewidth or mew	float value in points
markerfacecolor or mfc	any matplotlib color
markerfacecoloralt or mfcalt	any matplotlib color
markersize or ms	float
markevery	None | integer | (startind, stride)
picker	float distance in points or callable pick function fn(artist, event)
pickradius	float distance in points
rasterized	[True | False | None]
snap	unknown
solid_capstyle	['butt' | 'round' | 'projecting']
solid_joinstyle	['miter' | 'round' | 'bevel']
transform	a matplotlib.transforms.Transform instance
url	a url string
visible	[True | False]
xdata	1D array
ydata	1D array
zorder	any number

See Also:

dates for helper functions

date2num(), num2date() and drange() for help on creating the required floating point dates.

Additional kwargs: hold = [True|False] overrides default hold state

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 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, eg., 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:

```
# 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.

call signature:

```
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=256*, *Fs=2*, *Fc=0*, *detrend=<function detrend_none at 0x9bc0a3c>*, *window=<function window_hanning at 0x9bc0924>*, *noverlap=0*, *pad_to=None*, *sides='default'*, *scale_by_freq=None*, *hold=None*, ***kwargs*)

Plot the power spectral density.

Call signature:

```
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, **kwargs)
```

The power spectral density 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 $|\mathrm{fft}(i)|^2$ of each segment *i* are averaged to compute *Pxx*, with a scaling to correct for power loss due to windowing. *Fs* is the sampling frequency.

Keyword arguments:

***NFFT*: integer** The number of data points used in each block for the FFT. Must be even; 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.

***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.

***detrend*: 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.

***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.

***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 psd (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*

***sides*: ['default' | 'onesided' | 'twosided']** Specifies which sides of the PSD 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 PSD, while 'twosided' forces two-sided.

***scale_by_freq*: boolean** 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.

Returns the tuple (*Pxx*, *freqs*).

For plotting, the power is plotted as $10\log_{10}(P_{xx})$ for decibels, though *Pxx* itself is returned.

References: Bendat & Piersol – Random Data: Analysis and Measurement Procedures, John Wiley & Sons (1986)

kwargs control the `Line2D` properties:

Property	Description
`agg_filter`	unknown
`alpha`	float (0.0 transparent through 1.0 opaque)
`animated`	[True | False]
`antialiased` or aa	[True | False]
`axes`	an `Axes` instance
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color` or c	any matplotlib color
`contains`	a callable function
`dash_capstyle`	['butt' | 'round' | 'projecting']
`dash_joinstyle`	['miter' | 'round' | 'bevel']
`dashes`	sequence of on/off ink in points
`data`	2D array (rows are x, y) or two 1D arrays
`drawstyle`	['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post']
`figure`	a `matplotlib.figure.Figure` instance
`fillstyle`	['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']
`gid`	an id string
`label`	string or anything printable with '%s' conversion.
`linestyle` or ls	[`'-'` | `'--'` | `'-.'` | `':'` | `'None'` | `' '` | "] and any drawstyle in combina

Table 68.1

Property	Description
`linewidth` or lw	float value in points
`lod`	[True | False]
`marker`	[7 | 4 | 5 | 6 | 'o' | 'D' | 'h' | 'H' | '_' | '' | 'None' | None | ' ' | '8' | 'p
`markeredgecolor` or mec	any matplotlib color
`markeredgewidth` or mew	float value in points
`markerfacecolor` or mfc	any matplotlib color
`markerfacecoloralt` or mfcalt	any matplotlib color
`markersize` or ms	float
`markevery`	None | integer | (startind, stride)
`picker`	float distance in points or callable pick function `fn(artist, event)`
`pickradius`	float distance in points
`rasterized`	[True | False | None]
`snap`	unknown
`solid_capstyle`	['butt' | 'round' | 'projecting']
`solid_joinstyle`	['miter' | 'round' | 'bevel']
`transform`	a `matplotlib.transforms.Transform` instance
`url`	a url string
`visible`	[True | False]
`xdata`	1D array
`ydata`	1D array
`zorder`	any number

Example:

Additional kwargs: hold = [True|False] overrides default hold state

`matplotlib.pyplot.`**`quiver`**`(`*`*args`*`, `*`**kw`*`)`

Plot a 2-D field of arrows.

call signatures:

```
quiver(U, V, **kw)
quiver(U, V, C, **kw)
quiver(X, Y, U, V, **kw)
quiver(X, Y, U, V, C, **kw)
```

Arguments:

***X*, *Y*:** The x and y coordinates of the arrow locations (default is tail of arrow; see *pivot* kwarg)

***U*, *V*:** Give the x and y components of the arrow vectors

***C*:** An optional array used to map colors to the arrows

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:

***units*: ['width' | 'height' | 'dots' | 'inches' | 'x' | 'y' | 'xy']** Arrow units; the arrow dimensions *except for length* are in multiples of this unit.

- 'width' or 'height': the width or height of the axes
- 'dots' or 'inches': pixels or inches, based on the figure dpi
- 'x', 'y', or 'xy': *X*, *Y*, or sqrt(X^2+Y^2) 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 the window is resized; for 'dots' or 'inches', resizing does not change the arrows.

***angles*: ['uv' | 'xy' | array]** With the default 'uv', the arrow aspect ratio is 1, so that if *U*==*V* the angle of the arrow on the plot is 45 degrees CCW from the *x*-axis. With 'xy', the arrow points from (x,y) to (x+u, y+v). Alternatively, arbitrary angles may be specified as an array of values in degrees, CCW from the *x*-axis.

***scale*: [*None* | float]** Data units per arrow length unit, e.g. m/s per plot width; a smaller scale parameter makes the arrow longer. 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*: *None*, or any of the *units* options.** For example, if *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', then the vector will be half the width 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*:** 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** Head width as multiple of shaft width, default is 3

***headlength*: scalar** Head length as multiple of shaft width, default is 5

***headaxislength*: scalar** Head length at shaft intersection, default is 4.5

***minshaft*: scalar** 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** 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' | 'middle' | 'tip']** 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]** This is a synonym for the PolyCollection facecolor kwarg. If *C* has been set, *color* has no effect.

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.

linewidths and edgecolors can be used to customize the arrow outlines. Additional PolyCollection keyword arguments:

Property	Description
agg_filter	unknown
alpha	float or None
animated	[True | False]
antialiased or antialiaseds	Boolean or sequence of booleans
array	unknown
axes	an Axes instance
clim	a length 2 sequence of floats
clip_box	a matplotlib.transforms.Bbox instance
clip_on	[True | False]
clip_path	[(Path, Transform) | Patch | None]
cmap	a colormap or registered colormap name
color	matplotlib color arg or sequence of rgba tuples
colorbar	unknown
contains	a callable function
edgecolor or edgecolors	matplotlib color arg or sequence of rgba tuples
facecolor or facecolors	matplotlib color arg or sequence of rgba tuples
figure	a matplotlib.figure.Figure instance
gid	an id string
hatch	['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
label	string or anything printable with '%s' conversion.
linestyle or linestyles or dashes	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
linewidth or lw or linewidths	float or sequence of floats

Continued on next page

Table 68.20 – continued from previous page

Property	Description
`lod`	[True | False]
`norm`	unknown
`offset_position`	unknown
`offsets`	float or sequence of floats
`paths`	unknown
`picker`	[None|float|boolean|callable]
`pickradius`	unknown
`rasterized`	[True | False | None]
`snap`	unknown
`transform`	`Transform` instance
`url`	a url string
`urls`	unknown
`visible`	[True | False]
`zorder`	any number

Additional kwargs: hold = [True|False] overrides default hold state

`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*.

***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.

Additional kwargs: hold = [True|False] overrides default hold state

`matplotlib.pyplot.rc(*args, **kwargs)`

Set the current rc params. Group is the grouping for the rc, eg. 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, eg. (*xtick*, *ytick*). *kwargs* is a dictionary attribute name/value pairs, eg:

```
rc('lines', linewidth=2, color='r')
```

sets the current rc params and is equivalent to:

```
rcParams['lines.linewidth'] = 2
rcParams['lines.color'] = 'r'
```

The following aliases are available to save typing for interactive users:

Alias	Property
'lw'	'linewidth'
'ls'	'linestyle'
'c'	'color'
'fc'	'facecolor'
'ec'	'edgecolor'
'mew'	'markeredgewidth'
'aa'	'antialiased'

Thus you could abbreviate the above rc command as:

```
rc('lines', lw=2, c='r')
```

Note you can use python's kwargs dictionary facility to store dictionaries of default parameters. Eg, you can customize the font rc as follows:

```
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 `rcdefaults()` to restore the default rc params after changes.

`matplotlib.pyplot.rcdefaults()`
Restore the default rc params. These are not the params loaded by the rc file, but mpl's internal params. See rc_file_defaults for reloading the default params from the rc file

`matplotlib.pyplot.rgrids(*args, **kwargs)`
Get or set the radial gridlines on a polar plot.

call signatures:

```
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:

```
# 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' )
```

`matplotlib.pyplot.savefig(*args, **kwargs)`
Save the current figure.

Call signature:

```
savefig(fname, dpi=None, facecolor='w', edgecolor='w',
        orientation='portrait', papertype=None, format=None,
        transparent=False, bbox_inches=None, pad_inches=0.1)
```

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`]** The resolution in dots per inch. If *None* it will default to the value `savefig.dpi` in the matplotlibrc file.

***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.

***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.

`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=20*, *c='b'*, *marker='o'*, *cmap=None*, *norm=None*, *vmin=None*, *vmax=None*, *alpha=None*, *linewidths=None*, *faceted=True*, *verts=None*, *hold=None*, ***kwargs*)

Make a scatter plot.

Call signatures:

```
scatter(x, y, s=20, c='b', marker='o', cmap=None, norm=None,
        vmin=None, vmax=None, alpha=None, linewidths=None,
        verts=None, **kwargs)
```

Make a scatter plot of *x* versus *y*, where *x*, *y* are converted to 1-D sequences which must be of the same length, *N*.

Keyword arguments:

> ***s*:** size in points^2. It is a scalar or an array of the same length as *x* and *y*.
>
> ***c*:** 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.
>
> ***marker*:** can be one of:

marker	description
`7`	caretdown
`4`	caretleft
`5`	caretright
`6`	caretup
`'o'`	circle
`'D'`	diamond
`'h'`	hexagon1
`'H'`	hexagon2
`'_'`	hline
`''`	nothing
`'None'`	nothing
`None`	nothing
`' '`	nothing
`'8'`	octagon
`'p'`	pentagon
`','`	pixel
`'+'`	plus
`'.'`	point
`'s'`	square
`'*'`	star
`'d'`	thin_diamond
`3`	tickdown
`0`	tickleft
`1`	tickright
`2`	tickup
`'1'`	tri_down
`'3'`	tri_left
`'4'`	tri_right

Continued on next page

Table 68.21 – continued from previous page

marker	description
`'2'`	tri_up
`'v'`	triangle_down
`'<'`	triangle_left
`'>'`	triangle_right
`'^'`	triangle_up
`'|'`	vline
`'x'`	x
`'$...$'`	render the string using mathtext.
verts	a list of (x, y) pairs used for Path vertices.
path	a `Path` instance.
(*numsides*, *style*, *angle*)	see below

The marker can also be a tuple (*numsides*, *style*, *angle*), which will create a custom, regular symbol.

***numsides*:** the number of sides

***style*:** the style of the regular symbol:

Value	Description
0	a regular polygon
1	a star-like symbol
2	an asterisk
3	a circle (*numsides* and *angle* is ignored)

***angle*:** the angle of rotation of the symbol, in degrees

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.

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.

Other keyword arguments: the color mapping and normalization arguments will be used only if *c* is an array of floats.

***cmap*: [*None* | Colormap]** A `matplotlib.colors.Colormap` instance or registered name. If *None*, defaults to rc `image.cmap`. *cmap* is only used if *c* is an array of floats.

***norm*: [*None* | Normalize]** A `matplotlib.colors.Normalize` instance is used to scale luminance data to 0, 1. If *None*, use the default `normalize()`. *norm* is only used if *c* is an array of floats.

***vmin*/*vmax*:** *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 *C* is used. Note if you pass a *norm* instance, your settings for *vmin* and *vmax* will be ignored.

alpha*: `0 <= scalar <= 1` or *None The alpha value for the patches

***linewidths*: [*None* | scalar | sequence]** If *None*, defaults to (lines.linewidth,). Note that this is a tuple, and if you set the linewidths argument you must set it as a sequence of floats, as required by `RegularPolyCollection`.

Optional kwargs control the `Collection` properties; in particular:

***edgecolors*:** The string 'none' to plot faces with no outlines

***facecolors*:** The string 'none' to plot unfilled outlines

Here are the standard descriptions of all the `Collection` kwargs:

Property	Description
`agg_filter`	unknown
`alpha`	float or None
`animated`	[True | False]
`antialiased` or antialiaseds	Boolean or sequence of booleans
`array`	unknown
`axes`	an `Axes` instance
`clim`	a length 2 sequence of floats
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`cmap`	a colormap or registered colormap name
`color`	matplotlib color arg or sequence of rgba tuples
`colorbar`	unknown
`contains`	a callable function
`edgecolor` or edgecolors	matplotlib color arg or sequence of rgba tuples
`facecolor` or facecolors	matplotlib color arg or sequence of rgba tuples
`figure`	a `matplotlib.figure.Figure` instance
`gid`	an id string
`hatch`	['/' | '\\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
`label`	string or anything printable with '%s' conversion.
`linestyle` or linestyles or dashes	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
`linewidth` or lw or linewidths	float or sequence of floats
`lod`	[True | False]
`norm`	unknown
`offset_position`	unknown
`offsets`	float or sequence of floats
`paths`	unknown
`picker`	[None|float|boolean|callable]

Continued on next page

Table 68.22 – continued from previous page

Property	Description
pickradius	unknown
rasterized	[True | False | None]
snap	unknown
transform	Transform instance
url	a url string
urls	unknown
visible	[True | False]
zorder	any number

A Collection instance is returned.

Additional kwargs: hold = [True|False] overrides default hold state

`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.

Call signature:

```
semilogx(*args, **kwargs)
```

semilogx() supports all the keyword arguments of plot() and matplotlib.axes.Axes.set_xscale().

Notable keyword arguments:

***basex*: scalar > 1** Base of the *x* logarithm

***subsx*: [*None* | sequence]** 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*: ['mask' | 'clip']** Non-positive values in *x* can be masked as invalid, or clipped to a very small positive number

The remaining valid kwargs are Line2D properties:

Property	Description
agg_filter	unknown
alpha	float (0.0 transparent through 1.0 opaque)
animated	[True | False]

Table 68.2

Property	Description
`antialiased` or aa	[True | False]
`axes`	an `Axes` instance
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color` or c	any matplotlib color
`contains`	a callable function
`dash_capstyle`	['butt' | 'round' | 'projecting']
`dash_joinstyle`	['miter' | 'round' | 'bevel']
`dashes`	sequence of on/off ink in points
`data`	2D array (rows are x, y) or two 1D arrays
`drawstyle`	['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post']
`figure`	a `matplotlib.figure.Figure` instance
`fillstyle`	['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']
`gid`	an id string
`label`	string or anything printable with '%s' conversion.
`linestyle` or ls	[`'-'` | `'--'` | `'-.'` | `':'` | `'None'` | `' '` | ''] and any drawstyle in combina
`linewidth` or lw	float value in points
`lod`	[True | False]
`marker`	[`7` | `4` | `5` | `6` | `'o'` | `'D'` | `'h'` | `'H'` | `'_'` | '' | `'None'` | `None` | `' '` | `'8'` | `'p`
`markeredgecolor` or mec	any matplotlib color
`markeredgewidth` or mew	float value in points
`markerfacecolor` or mfc	any matplotlib color
`markerfacecoloralt` or mfcalt	any matplotlib color
`markersize` or ms	float
`markevery`	None | integer | (startind, stride)
`picker`	float distance in points or callable pick function `fn(artist, event)`
`pickradius`	float distance in points
`rasterized`	[True | False | None]
`snap`	unknown
`solid_capstyle`	['butt' | 'round' | 'projecting']
`solid_joinstyle`	['miter' | 'round' | 'bevel']
`transform`	a `matplotlib.transforms.Transform` instance
`url`	a url string
`visible`	[True | False]
`xdata`	1D array
`ydata`	1D array
`zorder`	any number

See Also:

`loglog()` For example code and figure

Additional kwargs: hold = [True|False] overrides default hold state

matplotlib.pyplot.**semilogy**(**args*, ***kwargs*)

Make a plot with log scaling on the *y* axis.

call signature:

```
semilogy(*args, **kwargs)
```

`semilogy()` supports all the keyword arguments of `plot()` and `matplotlib.axes.Axes.set_yscale()`.

Notable keyword arguments:

***basey*: scalar > 1** Base of the *y* logarithm

***subsy*: [*None* | sequence]** 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']** Non-positive values in *y* can be masked as invalid, or clipped to a very small positive number

The remaining valid kwargs are `Line2D` properties:

Property	Description
`agg_filter`	unknown
`alpha`	float (0.0 transparent through 1.0 opaque)
`animated`	[True | False]
`antialiased` or aa	[True | False]
`axes`	an `Axes` instance
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color` or c	any matplotlib color
`contains`	a callable function
`dash_capstyle`	['butt' | 'round' | 'projecting']
`dash_joinstyle`	['miter' | 'round' | 'bevel']
`dashes`	sequence of on/off ink in points
`data`	2D array (rows are x, y) or two 1D arrays
`drawstyle`	['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post']
`figure`	a `matplotlib.figure.Figure` instance
`fillstyle`	['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']
`gid`	an id string
`label`	string or anything printable with '%s' conversion.

Table 68.2

Property	Description
linestyle or ls	['-' | '--' | '-.' | ':' | 'None' | ' ' | "] and any drawstyle in combina
linewidth or lw	float value in points
lod	[True | False]
marker	[7 | 4 | 5 | 6 | 'o' | 'D' | 'h' | 'H' | '_' | " | 'None' | None | ' ' | '8' | 'p
markeredgecolor or mec	any matplotlib color
markeredgewidth or mew	float value in points
markerfacecolor or mfc	any matplotlib color
markerfacecoloralt or mfcalt	any matplotlib color
markersize or ms	float
markevery	None | integer | (startind, stride)
picker	float distance in points or callable pick function `fn(artist, event)`
pickradius	float distance in points
rasterized	[True | False | None]
snap	unknown
solid_capstyle	['butt' | 'round' | 'projecting']
solid_joinstyle	['miter' | 'round' | 'bevel']
transform	a matplotlib.transforms.Transform instance
url	a url string
visible	[True | False]
xdata	1D array
ydata	1D array
zorder	any number

See Also:

loglog() For example code and figure

Additional kwargs: hold = [True|False] overrides default hold state

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 `colors.Colormap` instance, or the name of a registered colormap.

See `register_cmap()` and `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:

```
>>> 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:

```
>>> 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:

```
>>> 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:

```
>>> 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:

```
>>> 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=256*, *Fs=2*, *Fc=0*, *detrend=<function detrend_none at 0x9bc0a3c>*, *window=<function window_hanning at 0x9bc0924>*, *noverlap=128*, *cmap=None*, *xextent=None*, *pad_to=None*, *sides='default'*, *scale_by_freq=None*, *hold=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, **kwargs)
```

Compute a spectrogram of data in *x*. Data are split into *NFFT* length segments and the PSD 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*.

Keyword arguments:

***NFFT*: integer** The number of data points used in each block for the FFT. Must be even; 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.

***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.

***detrend*: 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.

***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.

***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 psd (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

***sides*: ['default' | 'onesided' | 'twosided']** Specifies which sides of the PSD 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 PSD, while 'twosided' forces two-sided.

***scale_by_freq*: boolean** 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 128.

***Fc*: integer** The center frequency of *x* (defaults to 0), which offsets the y 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*:** The image extent along the x-axis. xextent = (xmin,xmax) The default is (0,max(bins)), where bins is the return value from `specgram()`

kwargs:

Additional kwargs are passed on to imshow which makes the specgram image

Return value is (*Pxx*, *freqs*, *bins*, *im*):

- *bins* are the time points the spectrogram is calculated over
- *freqs* is an array of frequencies
- *Pxx* is an array of shape `(len(times), len(freqs))` of power
- *im* is a `AxesImage` instance

Note: If *x* is real (i.e. non-complex), only the positive spectrum is shown. If *x* is complex, both positive and negative parts of the spectrum are shown. This can be overridden using the *sides* keyword argument.

Example:

Additional kwargs: hold = [True|False] overrides default hold state

`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'*, *hold=None*, ***kwargs*)

Plot the sparsity pattern on a 2-D array.

Call signature:

```
spy(Z, precision=0, marker=None, markersize=None,
    aspect='equal', **kwargs)
```

`spy(Z)` plots the sparsity pattern of the 2-D array *Z*.

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.

The array will be plotted as it would be printed, with the first index (row) increasing down and the second index (column) increasing to the right.

By default aspect is 'equal', so that each array element occupies a square space; set the aspect kwarg to 'auto' to allow the plot to fill the plot box, or to any scalar number to specify the aspect ratio of an array element directly.

Two plotting styles are available: image or marker. Both are available for full arrays, but only the marker style works for `scipy.sparse.spmatrix` instances.

If *marker* and *markersize* are *None*, an image will be returned and any remaining kwargs are passed to `imshow()`; else, a `Line2D` object will be returned with the value of marker determining the marker type, and any remaining kwargs passed to the `plot()` method.

If *marker* and *markersize* are *None*, useful kwargs include:

- *cmap*
- *alpha*

See Also:

`imshow()` For image options.

For controlling colors, e.g. cyan background and red marks, use:

```
cmap = mcolors.ListedColormap(['c','r'])
```

If *marker* or *markersize* is not *None*, useful kwargs include:

- *marker*

- *markersize*
- *color*

Useful values for *marker* include:

- 's' square (default)
- 'o' circle
- '.' point
- ',' pixel

See Also:

`plot()` For plotting options

Additional kwargs: hold = [True|False] overrides default hold state

`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:

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.

Additional kwargs: hold = [True|False] overrides default hold state

`matplotlib.pyplot.stem`(*x*, *y*, *linefmt='b-'*, *markerfmt='bo'*, *basefmt='r-'*, *bottom=None*, *label=None*, *hold=None*)

Create a stem plot.

Call signature:

```
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*.

Return value is a tuple (*markerline*, *stemlines*, *baseline*).

See Also:

This document for details.

Example:

Additional kwargs: hold = [True|False] overrides default hold state

`matplotlib.pyplot.step`(*x*, *y*, **args*, ***kwargs*)

Make a step plot.

Call signature:

```
step(x, y, *args, **kwargs)
```

Additional keyword args to `step()` are the same as those for `plot()`.

x and *y* must be 1-D sequences, and it is assumed, but not checked, that *x* is uniformly increasing.

Keyword arguments:

***where*: ['pre' | 'post' | 'mid']** If 'pre', 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.

Additional kwargs: hold = [True|False] overrides default hold state

`matplotlib.pyplot.streamplot`(*x*, *y*, *u*, *v*, *density=1*, *linewidth=None*, *color=None*, *cmap=None*, *norm=None*, *arrowsize=1*, *arrowstyle='-|>'*, *minlength=0.1*, *transform=None*, *hold=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 25x25 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.

arrowstyle [str] Arrow style specification. See FancyArrowPatch.

minlength [float] Minimum length of streamline in axes coordinates.

Returns:

stream_container [StreamplotSet]

Container object with attributes lines : matplotlib.collections.LineCollection of streamlines arrows : collection of matplotlib.patches.FancyArrowPatch objects

repesenting 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.

Additional kwargs: hold = [True|False] overrides default hold state

matplotlib.pyplot.**subplot**(**args*, ***kwargs*)

Create a new axes (subplot).

Creating axes with:

```
subplot(numRows, numCols, plotNum)
```

where *plotNum* = 1 is the first plot number and increasing *plotNums* fill rows first. max(*plotNum*) == *numRows* * *numCols*

You can leave out the commas if *numRows* <= *numCols* <= *plotNum* < 10, as in:

```
subplot(211)    # 2 rows, 1 column, first (upper) plot
```

`subplot(111)` is the default axis.

New subplots that overlap old will delete the old axes. If you do not want this behavior, use `add_subplot()` or the `axes()` command. Eg.:

```
from pylab import *
plot([1,2,3])  # implicitly creates subplot(111)
subplot(211)   # overlaps, subplot(111) is killed
plot(rand(12), rand(12))
subplot(212, axisbg='y') # creates 2nd subplot with yellow background
```

Keyword arguments:

***axisbg*:** 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/pylab_examples/polar_scatter.py` For an example

Example:

`matplotlib.pyplot.`**`subplot2grid`**(*shape*, *loc*, *rowspan=1*, *colspan=1*, ***kwargs*)

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[2])
subplotspec=gridspec.new_subplotspec(loc, rowspan, colspan)
subplot(subplotspec)
```

`matplotlib.pyplot.`**`subplot_tool`**(*targetfig=None*)

Launch a subplot tool window for a figure.

A `matplotlib.widgets.SubplotTool` instance is returned.

`matplotlib.pyplot.`**`subplots`**(*nrows=1*, *ncols=1*, *sharex=False*, *sharey=False*, *squeeze=True*, *subplot_kw=None*, ***fig_kw*)

Create a figure with a set of subplots already made.

This utility wrapper makes it convenient to create common layouts of subplots, including the enclosing figure object, in a single call.

Keyword arguments:

nrows [int] Number of rows of the subplot grid. Defaults to 1.

ncols [int] Number of columns of the subplot grid. Defaults to 1.

sharex [string or bool] If *True*, the X axis will be shared amongst all subplots. If *True* and you have multiple rows, the x tick labels on all but the last row of plots will have visible set to *False* If a string must be one of "row", "col", "all", or "none". "all" has the same effect as *True*, "none" has the same effect as *False*. If "row", each subplot row will share a X axis. If "col", each subplot column will share a X axis and the x tick labels on all but the last row will have visible set to *False*.

sharey [string or bool] If *True*, the Y axis will be shared amongst all subplots. If *True* and you have multiple columns, the y tick labels on all but the first column of plots will have visible set to *False* If a string must be one of "row", "col", "all", or "none". "all" has the same effect as *True*, "none" has the same effect as *False*. If "row", each subplot row will share a Y axis. If "col", each subplot column will share a Y axis and the y tick labels on all but the last row will have visible set to *False*.

squeeze [bool] If *True*, extra dimensions are squeezed out from the returned axis object:

- if only one subplot is constructed (nrows=ncols=1), the resulting single Axis object is returned as a scalar.
- for Nx1 or 1xN subplots, the returned object is a 1-d numpy object array of Axis objects are returned as numpy 1-d arrays.
- for NxM subplots with N>1 and M>1 are returned as a 2d array.

If *False*, no squeezing at all is done: the returned axis object is always a 2-d array containing Axis instances, even if it ends up being 1x1.

subplot_kw [dict] Dict with keywords passed to the `add_subplot()` call used to create each subplots.

fig_kw [dict] Dict with keywords passed to the `figure()` call. Note that all keywords not recognized above will be automatically included here.

Returns:

fig, ax : tuple

- *fig* is the `matplotlib.figure.Figure` object
- *ax* can be either a single axis object or an array of axis objects if more than one subplot was created. The dimensions of the resulting array can be controlled with the squeeze keyword, see above.

Examples:

```
x = np.linspace(0, 2*np.pi, 400)
y = np.sin(x**2)

# Just a figure and one subplot
f, ax = plt.subplots()
ax.plot(x, y)
ax.set_title('Simple plot')
```

```
# Two subplots, unpack the output 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)

# Four polar axes
plt.subplots(2, 2, subplot_kw=dict(polar=True))

# Share a X axis with each column of subplots
plt.subplots(2, 2, sharex='col')

# Share a Y axis with each row of subplots
plt.subplots(2, 2, sharey='row')

# Share a X and Y axis with all subplots
plt.subplots(2, 2, sharex='all', sharey='all')
# same as
plt.subplots(2, 2, sharex=True, sharey=True)
```

`matplotlib.pyplot.`**`subplots_adjust`**(**args*, ***kwargs*)

Tune the subplot layout.

call signature:

```
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
hspace = 0.2   # the amount of height reserved for white space between subplots
```

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

A `matplotlib.text.Text` instance is returned.

Example:

```
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. Eg, 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. 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:

Property	Description
agg_filter	unknown
alpha	float (0.0 transparent through 1.0 opaque)
animated	[True | False]
axes	an Axes instance
clip_box	a matplotlib.transforms.Bbox instance
clip_on	[True | False]
clip_path	[(Path, Transform) | Patch | None]
contains	a callable function
figure	a matplotlib.figure.Figure instance
fontsize	a float in points
gid	an id string
label	string or anything printable with '%s' conversion.
lod	[True | False]
picker	[None|float|boolean|callable]
rasterized	[True | False | None]
snap	unknown
transform	Transform instance
url	a url string
visible	[True | False]
zorder	any number

matplotlib.pyplot.**text**(*x*, *y*, *s*, *fontdict=None*, *withdash=False*, ***kwargs*)

Add text to the axes.

Call signature:

```
text(x, y, s, fontdict=None, **kwargs)
```

Add text in string *s* to axis at location *x*, *y*, data coordinates.

Keyword arguments:

***fontdict*:** A dictionary to override the default text properties. If *fontdict* is *None*, the defaults are determined by your rc parameters.

***withdash*: [*False* | *True*]** Creates a TextWithDash instance instead of a Text instance.

Individual keyword arguments can be used to override any given parameter:

```
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:

```
text(0.5, 0.5,'matplotlib',
     horizontalalignment='center',
```

```
    verticalalignment='center',
    transform = ax.transAxes)
```

You can put a rectangular box around the text instance (eg. to set a background color) by using the keyword *bbox*. *bbox* is a dictionary of `matplotlib.patches.Rectangle` properties. For example:

```
text(x, y, s, bbox=dict(facecolor='red', alpha=0.5))
```

Valid kwargs are `Text` properties:

Property	Description
`agg_filter`	unknown
`alpha`	float (0.0 transparent through 1.0 opaque)
`animated`	[True | False]
`axes`	an `Axes` instance
`backgroundcolor`	any matplotlib color
`bbox`	rectangle prop dict
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color`	any matplotlib color
`contains`	a callable function
`family` or fontfamily or fontname or name	[FONTNAME | 'serif' | 'sans-serif' | 'cursive' | 'fantasy' | 'mono
`figure`	a `matplotlib.figure.Figure` instance
`fontproperties` or font_properties	a `matplotlib.font_manager.FontProperties` instance
`gid`	an id string
`horizontalalignment` or ha	['center' | 'right' | 'left']
`label`	string or anything printable with '%s' conversion.
`linespacing`	float (multiple of font size)
`lod`	[True | False]
`multialignment`	['left' | 'right' | 'center']
`path_effects`	unknown
`picker`	[None|float|boolean|callable]
`position`	(x,y)
`rasterized`	[True | False | None]
`rotation`	[angle in degrees | 'vertical' | 'horizontal']
`rotation_mode`	unknown
`size` or fontsize	[size in points | 'xx-small' | 'x-small' | 'small' | 'medium' | 'large
`snap`	unknown
`stretch` or fontstretch	[a numeric value in range 0-1000 | 'ultra-condensed' | 'extra-conc
`style` or fontstyle	['normal' | 'italic' | 'oblique']
`text`	string or anything printable with '%s' conversion.

Table 68.25 – con

Property	Description
transform	Transform instance
url	a url string
variant or fontvariant	['normal' | 'small-caps']
verticalalignment or va or ma	['center' | 'top' | 'bottom' | 'baseline']
visible	[True | False]
weight or fontweight	[a numeric value in range 0-1000 | 'ultralight' | 'light' | 'normal' |
x	float
y	float
zorder	any number

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):

```
lines, labels = thetagrids()
```

Otherwise the syntax is:

```
lines, labels = thetagrids(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). Eg. 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:

```
# 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.

Keyword arguments:

axis [['x' | 'y' | 'both']] Axis on which to operate; default is 'both'.

reset [[True | False]] 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']] Puts ticks inside or outside the axes.

length Tick length in points.

width Tick width in points.

color Tick color; accepts any mpl color spec.

pad Distance in points between tick and label.

labelsize Tick label font size in points or as a string (e.g. 'large').

labelcolor Tick label color; mpl color spec.

colors Changes the tick color and the label color to the same value: mpl color spec.

zorder Tick and label zorder.

bottom, top, left, right [[bool | 'on' | 'off']] controls whether to draw the respective ticks.

labelbottom, labeltop, labelleft, labelright Boolean or ['on' | 'off'], controls whether to draw the respective tick labels.

Example:

```
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.pyplot.**ticklabel_format**(***kwargs*)

Change the ScalarFormatter used by default for linear axes.

Optional keyword arguments:

Keyword	Description
style	['sci' (or 'scientific') | 'plain'] plain turns off scientific notation
scilimits	(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.
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']
useLocale	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.

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 the title of the current axis.

Default font override is:

```
override = {'fontsize': 'medium',
            'verticalalignment': 'baseline',
            'horizontalalignment': 'center'}
```

See Also:

`text()` for information on how override and the optional args work.

`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 `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*

```
tricontourf(..., Z, V)
```

fill the (len(*V*)-1) regions between the values in *V*

```
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; eg 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

***nchunk*: [0 | integer]** If 0, no subdivision of the domain. Specify a positive integer to divide the domain into subdomains of roughly *nchunk* by *nchunk* points. This may never actually be advantageous, so this option may be removed. Chunking introduces artifacts at the chunk boundaries unless *antialiased* is *False*.

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:

Additional kwargs: hold = [True|False] overrides default hold state

`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 `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*

```
tricontourf(..., Z, V)
```

fill the (len(*V*)-1) regions between the values in *V*

```
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; eg 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

***nchunk*: [0 | integer]** If 0, no subdivision of the domain. Specify a positive integer to divide the domain into subdomains of roughly *nchunk* by *nchunk* points. This may never actually be advantageous, so this option may be removed. Chunking introduces artifacts at the chunk boundaries unless *antialiased* is *False*.

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:

Additional kwargs: hold = [True|False] overrides default hold state

`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:

```
tripcolor(triangulation, ...)
```

where triangulation is a `Triangulation` object, or

```
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. *shading* of 'faceted' is deprecated; please use *edgecolors* instead.

The remaining kwargs are the same as for `pcolor()`.

Example:

Additional kwargs: hold = [True|False] overrides default hold state

`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 `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 a explanation of these possibilities.

The remaining args and kwargs are the same as for `plot()`.

Example:

Additional kwargs: hold = [True|False] overrides default hold state

`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:

`examples/api_examples/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.vlines`(*x*, *ymin*, *ymax*, *colors='k'*, *linestyles='solid'*, *label=''*, *hold=None*, ***kwargs*)

Plot vertical lines.

Call signature:

```
vlines(x, ymin, ymax, color='k', linestyles='solid')
```

Plot vertical lines at each *x* from *ymin* to *ymax*. *ymin* or *ymax* can be scalars or len(*x*) numpy arrays. If they are scalars, then the respective values are constant, else the heights of the lines are determined by *ymin* and *ymax*.

***colors* :** A line collection's color args, either a single color or a `len(x)` list of colors *linestyles* : ['solid' | 'dashed' | 'dashdot' | 'dotted']

Returns the `matplotlib.collections.LineCollection` that was added.

kwargs are `LineCollection` properties:

Property	Description
`agg_filter`	unknown
`alpha`	float or None
`animated`	[True | False]
`antialiased` or antialiaseds	Boolean or sequence of booleans
`array`	unknown
`axes`	an `Axes` instance
`clim`	a length 2 sequence of floats
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`cmap`	a colormap or registered colormap name
`color`	matplotlib color arg or sequence of rgba tuples
`colorbar`	unknown
`contains`	a callable function
`edgecolor` or edgecolors	matplotlib color arg or sequence of rgba tuples
`facecolor` or facecolors	matplotlib color arg or sequence of rgba tuples
`figure`	a `matplotlib.figure.Figure` instance
`gid`	an id string
`hatch`	['/' | '\\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
`label`	string or anything printable with '%s' conversion.
`linestyle` or linestyles or dashes	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
`linewidth` or lw or linewidths	float or sequence of floats
`lod`	[True | False]
`norm`	unknown
`offset_position`	unknown
`offsets`	float or sequence of floats
`paths`	unknown
`picker`	[None|float|boolean|callable]
`pickradius`	unknown
`rasterized`	[True | False | None]
`segments`	unknown
`snap`	unknown
`transform`	`Transform` instance
`url`	a url string
`urls`	unknown
`verts`	unknown
`visible`	[True | False]

Continued on next page

Table 68.26 – continued from previous page

Property	Description
zorder	any number

Additional kwargs: hold = [True|False] overrides default hold state

matplotlib.pyplot.**waitforbuttonpress**(**args*, ***kwargs*)

Call signature:

```
waitforbuttonpress(self, timeout=-1)
```

Blocking call to interact with the figure.

This will return True is 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.

matplotlib.pyplot.**winter**()

set the default colormap to winter and apply to current image if any. See help(colormaps) for more information

matplotlib.pyplot.**xcorr**(*x*, *y*, *normed=True*, *detrend=<function detrend_none at 0x9bc0a3c>*, *usevlines=True*, *maxlags=10*, *hold=None*, ***kwargs*)

Plot the cross correlation between *x* and *y*.

Call signature:

```
xcorr(self, x, y, normed=True, detrend=mlab.detrend_none,
  usevlines=True, maxlags=10, **kwargs)
```

If *normed* = *True*, normalize the data by the cross correlation at 0-th lag. *x* and y are detrended by the *detrend* callable (default no normalization). *x* and *y* must be equal length.

Data are plotted as `plot(lags, c, **kwargs)`

Return value is a tuple (*lags*, *c*, *line*) 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()`.

The default *linestyle* is *None* and the default *marker* is 'o', though these can be overridden with keyword args. The cross correlation is performed with `numpy.correlate()` with *mode* = 2.

If *usevlines* is *True*:

vlines() rather than plot() is used to draw vertical lines from the origin to the xcorr. Otherwise the plotstyle is determined by the kwargs, which are Line2D properties.

The return value is a tuple (*lags*, *c*, *linecol*, *b*) where *linecol* is the matplotlib.collections.LineCollection instance and *b* is the *x*-axis.

maxlags is a positive integer detailing the number of lags to show. The default value of *None* will return all (2*len(x)-1) lags.

Example:

xcorr() is top graph, and acorr() is bottom graph.

Additional kwargs: hold = [True|False] overrides default hold state

matplotlib.pyplot.**xlabel**(*s*, **args*, ***kwargs*)

Set the *x* axis label of the current axis.

Default override is:

```
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.

```
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, eg.:

```
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:

```
xscale(scale, **kwargs)
```

The available scales are: 'linear' | 'log' | '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.

'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
xticks( arange(5), ('Tom', 'Dick', 'Harry', 'Sally', 'Sue') )
```

The keyword args, if any, are `Text` properties. For example, to rotate long labels:

```
xticks( arange(12), calendar.month_name[1:13], rotation=17 )
```

`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.

`matplotlib.pyplot.`**`ylim`**(**args*, ***kwargs*)

Get or set the *y*-limits of the current axes.

```
ymin, ymax = ylim()   # return the current ylim
ylim( (ymin, ymax) )  # set the ylim to ymin, ymax
ylim( ymin, ymax )    # set the ylim to ymin, ymax
```

If you do not specify args, you can pass the *ymin* and *ymax* as kwargs, eg.:

```
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.

`matplotlib.pyplot.`**`yscale`**(**args*, ***kwargs*)

Set the scaling of the *y*-axis.

call signature:

```
yscale(scale, **kwargs)
```

The available scales are: 'linear' | 'log' | '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.

'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
yticks( arange(5), ('Tom', 'Dick', 'Harry', 'Sally', 'Sue') )
```

The keyword args, if any, are `Text` properties. For example, to rotate long labels:

```
yticks( arange(12), calendar.month_name[1:13], rotation=45 )
```

CHAPTER

SIXTYNINE

SANKEY

69.1 matplotlib.sankey

Module for creating Sankey diagrams using matplotlib

class `matplotlib.sankey.`**`Sankey`**(*ax=None*, *scale=1.0*, *unit=''*, *format='%G'*, *gap=0.25*, *radius=0.1*, *shoulder=0.03*, *offset=0.15*, *head_angle=100*, *margin=0.4*, *tolerance=1e-06*, ***kwargs*)

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:

Field	Description
ax	axes onto which the data should be plotted If *ax* isn't provided, new axes will be created.
scale	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).
unit	string representing the physical unit associated with the flow quantities If *unit* is None, then none of the quantities are labeled.
format	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)
gap	space between paths that break in/break away to/from the top or bottom
radius	inner radius of the vertical paths
shoulder	size of the shoulders of output arrowS
offset	text offset (from the dip or tip of the arrow)
head_angle	angle of the arrow heads (and negative of the angle of the tails) [deg]
margin	minimum space between Sankey outlines and the edge of the plot area
tolerance	acceptable maximum of the magnitude of the sum of flows The magnitude of the sum of connected flows cannot be greater than *tolerance*.

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:

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:

Keyword	Description
patchlabel	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.
flows	array of flow values By convention, inputs are positive and outputs are negative.
orientations	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.
labels	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.
trunklength	length between the bases of the input and output groups
pathlengths	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.
prior	index of the prior diagram to which this diagram should be connected
connect	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.
rotation	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).

Valid kwargs are `matplotlib.patches.PathPatch()` arguments:

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
clip_box	a matplotlib.transforms.Bbox instance
clip_on	[True | False]
clip_path	[(Path, Transform) | Patch | None]
color	matplotlib color spec
contains	a callable function
edgecolor or ec	mpl color spec, or None for default, or 'none' for no color
facecolor or fc	mpl color spec, or None for default, or 'none' for no color
figure	a matplotlib.figure.Figure instance
fill	[True | False]
gid	an id string
hatch	['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
label	string or anything printable with '%s' conversion.
linestyle or ls	['solid' | 'dashed' | 'dashdot' | 'dotted']
linewidth or lw	float or None for default
lod	[True | False]
path_effects	unknown
picker	[None|float|boolean|callable]
rasterized	[True | False | None]
snap	unknown
transform	Transform instance
url	a url string
visible	[True | False]
zorder	any number

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 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:

Field	Description
patch	Sankey outline (an instance of `PathPatch`)
flows	values of the flows (positive for input, negative for output)
angles	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.
tips	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.
text	`Text` instance for the label of the diagram
texts	list of `Text` instances for the labels of flows

See Also:

add()

CHAPTER

SEVENTY

SPINES

70.1 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:

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
clip_box	a matplotlib.transforms.Bbox instance
clip_on	[True | False]
clip_path	[(Path, Transform) | Patch | None]
color	matplotlib color spec
contains	a callable function
edgecolor or ec	mpl color spec, or None for default, or 'none' for no color
facecolor or fc	mpl color spec, or None for default, or 'none' for no color
figure	a matplotlib.figure.Figure instance
fill	[True | False]
gid	an id string
hatch	['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
label	string or anything printable with '%s' conversion.
linestyle or ls	['solid' | 'dashed' | 'dashdot' | 'dotted']
linewidth or lw	float or None for default
lod	[True | False]
path_effects	unknown
picker	[None|float|boolean|callable]
rasterized	[True | False | None]
snap	unknown
transform	Transform instance
url	a url string
visible	[True | False]
zorder	any number

classmethod circular_spine(*axes*, *center*, *radius*, ***kwargs*)
(staticmethod) Returns a circular Spine.

cla()
Clear the current spine

draw(*artist*, *renderer*, **args*, ***kwargs*)

get_bounds()
Get the bounds of the spine.

get_patch_transform()

get_path()

get_position()
get the spine position

get_smart_bounds()
get whether the spine has smart bounds

get_spine_transform()
get the spine transform

is_frame_like()
return True if directly on axes frame

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

CHAPTER

SEVENTYONE

TICKER

71.1 `matplotlib.ticker`

71.1.1 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..

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, eg 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 (eg. where x = range(len(y)))

`LinearLocator` evenly spaced ticks from min to max

`LogLocator` logarithmically ticks from min to max

`MultipleLocator`

ticks and range are a multiple of base; either integer or float

`OldAutoLocator` choose a MultipleLocator and dyamically 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.

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:

```
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, eg 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

FormatStrFormatter use a sprintf format string

ScalarFormatter default formatter for scalars; autopick the fmt string

LogFormatter formatter for log axes

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:

```
ax.xaxis.set_major_formatter( xmajorFormatter )
ax.xaxis.set_minor_formatter( xminorFormatter )
```

```
ax.yaxis.set_major_formatter( ymajorFormatter )
ax.yaxis.set_minor_formatter( yminorFormatter )
```

See *pylab_examples-major_minor_demo1* for an example of setting major an 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`

Convert the tick location to a string

`fix_minus`(*s*)

some classes may want to replace a hyphen for minus with the proper unicode symbol as described here. The default is to do nothing

Note, if you use this method, eg in :meth'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*)

`format_data_short`(*value*)

return a short string version

`get_offset`()

`locs` = []

`set_locs`(*locs*)

class `matplotlib.ticker.`**`FixedFormatter`**(*seq*)

Bases: `matplotlib.ticker.Formatter`

Return fixed strings for tick labels

seq is a sequence of strings. For positions `i < len(seq)` return *seq[i]* regardless of *x*. Otherwise return ‘’

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`

User defined function for formatting

class `matplotlib.ticker.`**FormatStrFormatter**(*fmt*)
Bases: `matplotlib.ticker.Formatter`

Use a format string to format the tick

class `matplotlib.ticker.`**ScalarFormatter**(*useOffset=True*, *useMathText=None*, *useLocale=None*)
Bases: `matplotlib.ticker.Formatter`

Tick location is 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*)
use a unicode minus rather than hyphen

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.

e.g. `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 `set_scientific()`.

set_scientific(*b*)
True or False to turn scientific notation on or off see also `set_powerlimits()`

set_useLocale(*val*)

set_useOffset(*val*)

useLocale

useOffset

class `matplotlib.ticker.`**LogFormatter**(*base=10.0*, *labelOnlyBase=True*)
Bases: `matplotlib.ticker.Formatter`

Format values for log axis;

base is used to locate the decade tick, which will be the only one to be labeled if *labelOnlyBase* is `False`

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 on/off minor ticks labeling

pprint_val(*x*, *d*)

class `matplotlib.ticker.`**LogFormatterExponent**(*base=10.0*, *labelOnlyBase=True*)
Bases: `matplotlib.ticker.LogFormatter`

Format values for log axis; using `exponent = log_base(value)`

base is used to locate the decade tick, which will be the only one to be labeled if *labelOnlyBase* is `False`

class `matplotlib.ticker.`**LogFormatterMathtext**(*base=10.0*, *labelOnlyBase=True*)
Bases: `matplotlib.ticker.LogFormatter`

Format values for log axis; using `exponent = log_base(value)`

base is used to locate the decade tick, which will be the only one to be labeled if *labelOnlyBase* is `False`

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

`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 will be overridden.

`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, eg on every 5th point. It is assumed that you are doing index plotting; ie 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

`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.

`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

`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=15*)

Bases: `matplotlib.ticker.Locator`

Determine the tick locations for log axes

place ticks on the location= base**i*subs[j]

`base`(*base*)

set the base of the log scaling (major tick every base**i, i integer)

subs(*subs*)
set the minor ticks the log scaling every base**i*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

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.

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.

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. 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.

bin_boundaries(*vmin*, *vmax*)

default_params = {'trim': True, 'nbins': 10, 'steps': None, 'prune': None, 'integer': False, 'symme

set_params(***kwargs*)

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*)

CHAPTER

SEVENTYTWO

TIGHT_LAYOUT

72.1 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 and axes titles 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 calcuate 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.

`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).

CHAPTER

SEVENTYTHREE

UNITS

73.1 matplotlib.units

The classes here provide support for using custom classes with matplotlib, eg those that do not expose the array interface but know how to converter themselves to arrays. It also supoprts classes with units and units conversion. Use cases include converters for custom objects, eg 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:

```
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*)

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`**

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 ouput 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

CHAPTER

SEVENTYFOUR

WIDGETS

74.1 matplotlib.widgets

74.1.1 GUI Neutral widgets

Widgets that are designed to work for any of the GUI backends. All of these widgets require you to predefine an `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.

class `matplotlib.widgets.`**`AxesWidget`**`(`*ax*`)`

Bases: `matplotlib.widgets.Widget`

Widget that is connected to a single `Axes`.

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 call back ids for later clean up.

`disconnect_events``()`

Disconnect all events created by this widget.

`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.

class `matplotlib.widgets.Button`(*ax*, *label*, *image=None*, *color='0.85'*, *hovercolor='0.95'*)

Bases: `matplotlib.widgets.AxesWidget`

A GUI neutral button

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

ax The `matplotlib.axes.Axes` instance the button will be placed into.

label The button text. Accepts string.

image The image to place in the button, if not *None*. Can be any legal arg to imshow (numpy array, matplotlib Image instance, or PIL image).

color The color of the button when not activated

hovercolor The color of the button when the mouse is over it

disconnect(*cid*)

remove the observer with connection id *cid*

on_clicked(*func*)

When the button is clicked, call this *func* with event

A connection id is returned which can be used to disconnect

class `matplotlib.widgets.CheckButtons`(*ax*, *labels*, *actives*)

Bases: `matplotlib.widgets.AxesWidget`

A GUI neutral radio button

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

class matplotlib.widgets.**Cursor**(*ax*, *useblit=False*, ***lineprops*)
Bases: `matplotlib.widgets.AxesWidget`

A horizontal and vertical line span the axes that and move with the pointer. You can turn off the hline or vline spectively with the 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

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.**HorizontalSpanSelector**(*ax*, *onselect*, ***kwargs*)
Bases: `matplotlib.widgets.SpanSelector`

class matplotlib.widgets.**Lasso**(*ax*, *xy*, *callback=None*, *useblit=True*)
Bases: `matplotlib.widgets.AxesWidget`

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*)

Bases: `matplotlib.widgets.AxesWidget`

Selection curve of an arbitrary shape.

The selected path can be used in conjunction with **:function:'~matplotlib.path.Path.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:

```
ax = subplot(111)
ax.plot(x,y)

def onselect(verts):
    print verts
lasso = LassoSelector(ax, onselect)
```

ignore(*event*)

onmove(*event*)

onpress(*event*)

onrelease(*event*)

update_background(*event*)

class matplotlib.widgets.**LockDraw**

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 pr

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*, ***lineprops*)
Bases: matplotlib.widgets.Widget

Provide a vertical line cursor shared between multiple axes

Example usage:

```
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)
show()
```

clear(*event*)
clear the cursor

onmove(*event*)

class matplotlib.widgets.**RadioButtons**(*ax*, *labels*, *active=0*, *activecolor='blue'*)
Bases: matplotlib.widgets.AxesWidget

A GUI neutral radio button

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

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

class matplotlib.widgets.**RectangleSelector**(*ax*, *onselect*, *drawtype='box'*, *minspanx=None*, *minspany=None*, *useblit=False*, *lineprops=None*, *rectprops=None*, *spancoords='data'*, *button=None*)

Bases: `matplotlib.widgets.AxesWidget`

Select a min/max range of the x axes for a matplotlib Axes

Example usage:

```
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   : ', 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)

toggle_selector.RS = RectangleSelector(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:

```
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:

```
rectprops = dict(facecolor='red', edgecolor = 'black',
                 alpha=0.5, fill=False)
```

The line is drawn with *lineprops*; default:

```
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

`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.

Note, typically: 1 = left mouse button 2 = center mouse button (scroll wheel) 3 = right mouse button

get_active()
Get status of active mode (boolean variable)

ignore(*event*)
return *True* if *event* should be ignored

onmove(*event*)
on motion notify event if box/line is wanted

press(*event*)
on button press event

release(*event*)
on button release event

set_active(*active*)
Use this to activate / deactivate the RectangleSelector from your program with an boolean parameter *active*.

update()
draw using newfangled blit or oldfangled draw depending on useblit

update_background(*event*)
force an update of the background

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

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*

valinit The slider initial position

label The slider label

valfmt Used to format the slider value

closedmin* and *closedmax Indicate whether the slider interval is closed

slidermin* and *slidermax Used to constrain the value of this slider to the values of other sliders.

additional kwargs are passed on to `self.poly` which is the `matplotlib.patches.Rectangle` which draws the slider knob. See the `matplotlib.patches.Rectangle` documentation valid property names (e.g., *facecolor*, *edgecolor*, *alpha*, ...)

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*)
Bases: `matplotlib.widgets.AxesWidget`

Select a min/max range of the x or y axes for a matplotlib Axes

Example usage:

```
ax = subplot(111)
ax.plot(x,y)

def onselect(vmin, vmax):
    print vmin, vmax
span = SpanSelector(ax, onselect, 'horizontal')
```

***onmove_callback* is an optional callback that is called on mouse** move within the span range

Create a span selector in *ax*. When a selection is made, clear the span and call *onselect* with:

```
onselect(vmin, vmax)
```

and clear the span.

direction must be 'horizontal' or 'vertical'

If *minspan* is not *None*, ignore events smaller than *minspan*

The span rectangle is drawn with *rectprops*; default:: rectprops = dict(facecolor='red', alpha=0.5)

Set the visible attribute to *False* if you want to turn off the functionality of the span selector

ignore(*event*)
return *True* if *event* should be ignored

new_axes(*ax*)

onmove(*event*)
on motion notify event

press(*event*)
on button press event

release(*event*)
on button release event

update()
Draw using newfangled blit or oldfangled draw depending on *useblit*

update_background(*event*)
force an update of the background

class matplotlib.widgets.**SubplotTool**(*targetfig*, *toolfig*)
Bases: `matplotlib.widgets.Widget`

A tool to adjust to 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

funcbottom(*val*)

funchspace(*val*)

funcleft(*val*)

funcright(*val*)

functop(*val*)

funcwspace(*val*)

class matplotlib.widgets.**Widget**
Bases: `object`

Abstract base class for GUI neutral widgets

drawon = True

eventson = True

Part VIII

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)

FLTK FLTK (pronounced "fulltick") is a cross-platform C++ GUI toolkit for UNIX/Linux (X11), Microsoft Windows, and MacOS X

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 developerd 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

pyfltk pyfltk provides python wrappers for the *FLTK* widgets library for use with FLTKAgg

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 requied by the matplotlib QtAgg and Qt4Agg backends. Widely used on linux and windows; many linux distributions package this as 'python-qt3' 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 most recent 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 WXAgg 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.

PYTHON MODULE INDEX

m

INDEX

A

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z