Matplotlib

Release 0.98

Darren Dale, Michael Droettboom, Eric Firing, John Hunter

October 15, 2008

CONTENTS

Part I

The Matplotlib 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™ graphics commands, it does not require 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 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. 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. 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. 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

2.1 Dependencies

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 sourceforge site.

python **2.4 (or later but not python3)** matplotlib requires python 2.4 or later (download)

numpy **1.1 (or later)** array support for python (download)

libpng 1.1 (or later) library for loading and saving *PNG* files (download). libpng requires zlib. If you are a windows user, you can ignore this since 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 wrappersfor the Qt4Agg backend

pygtk **2.2 or later** The python wrappers for the GTK widgets library for use with the GTK or GTKAgg backend

wxpython **2.6 or later** The python wrappers for the wx widgets library for use with the WXAgg backend

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

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

Required libraries that ship with matplotlib

If you are downloading matplotlib or installing from source or subversion, you can ignore this section. This is useful for matplotlib developers and packagers who may want to disable the matplotlib version and ship a packaged version.

***agg* 2.4** The antigrain C++ rendering engine

pytz 2007g or later timezone handling for python datetime objects

dateutil 1.1 or later extensions to python datetime handling

Optional libraries that ship with matplotlib

As above, if you are downloading matplotlib or installing from source or subversion, you can ignore this section. This is useful for matplotlib developers and packagers who may want to disable the matplotlib version and ship a packaged version.

enthought traits 2.6 The traits component of the Enthought Tool Suite used in the experimental matplotlib traits rc system. matplotlib has decided to stop installing this library so packagers should not distribute the version included with matplotlib. packagers do not need to list this as a requirement because the traits support is experimental and disabled by default.

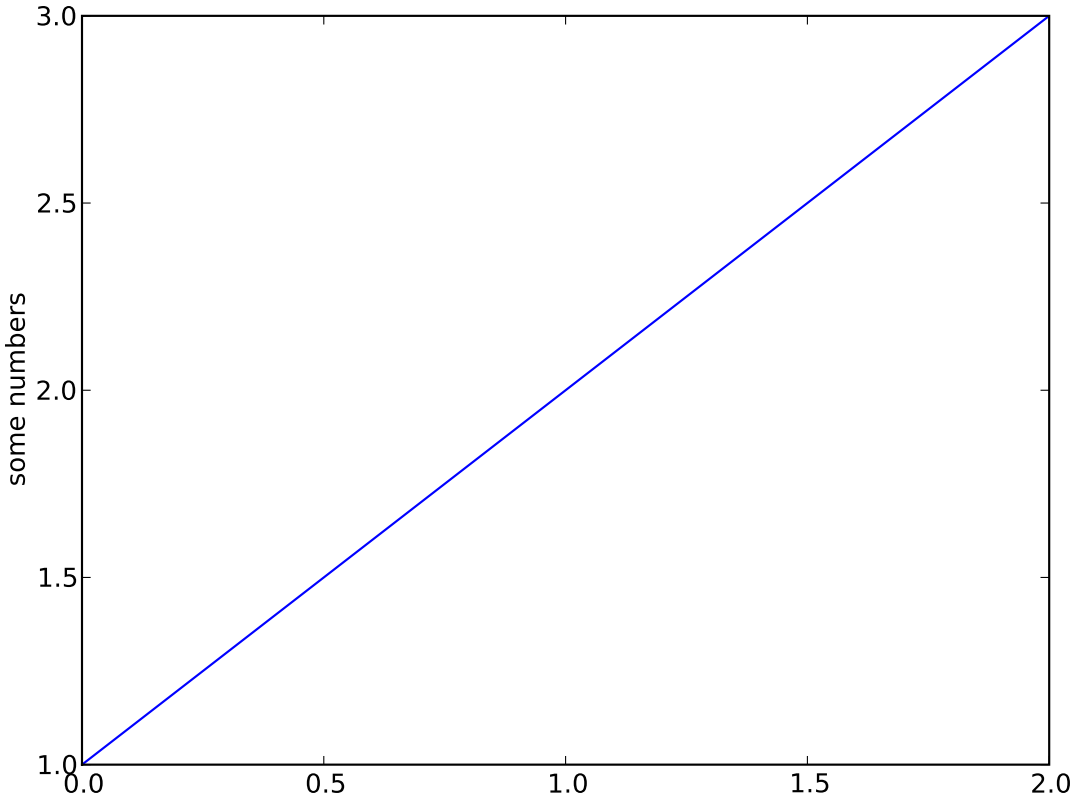
CHAPTER

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

```
import matplotlib.pyplot as plt
plt.plot([1,2,3])
plt.ylabel('some numbers')
plt.show()
```



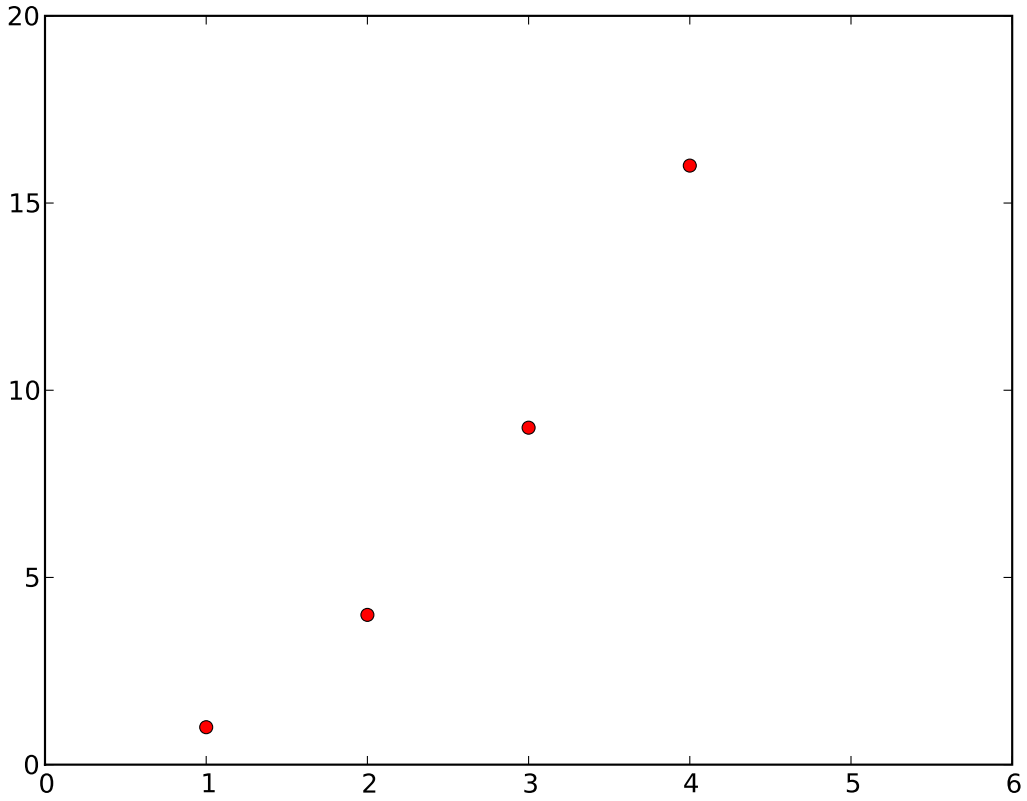
You may be wondering why the x-axis ranges from 0-3 and the y-axis from 1-4. If you provide a single list or array to the `plot()` command, matplotlib assumes it a sequence of y values, and automatically generates the x values for you. Since python ranges start with 0, the default x vector has the same 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 a 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

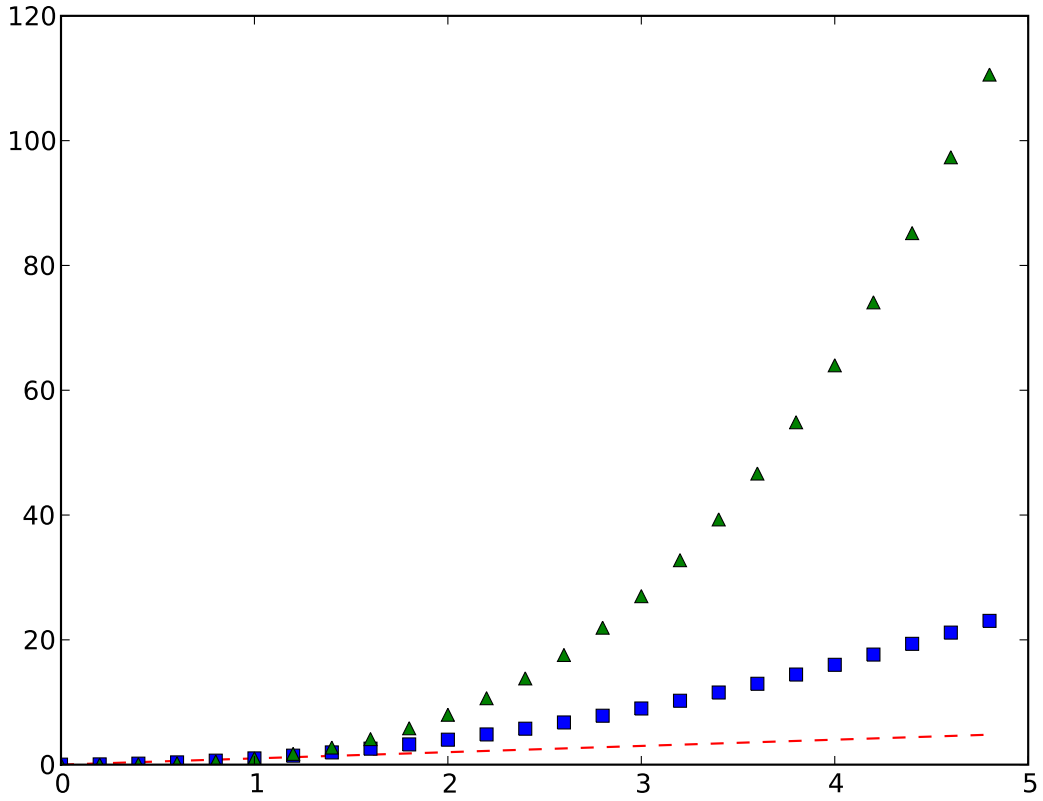
```
import matplotlib.pyplot as plt
plt.plot([1,2,3,4], [1,4,9,16], 'ro')
plt.axis([0, 6, 0, 20])
```



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.

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

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

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



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 of7 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, 'o')
line.set_antialiased(False) # turn off antialising
```

- Use the `setp()` command. The example below uses matlab handle graphics 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' | ...]
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
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

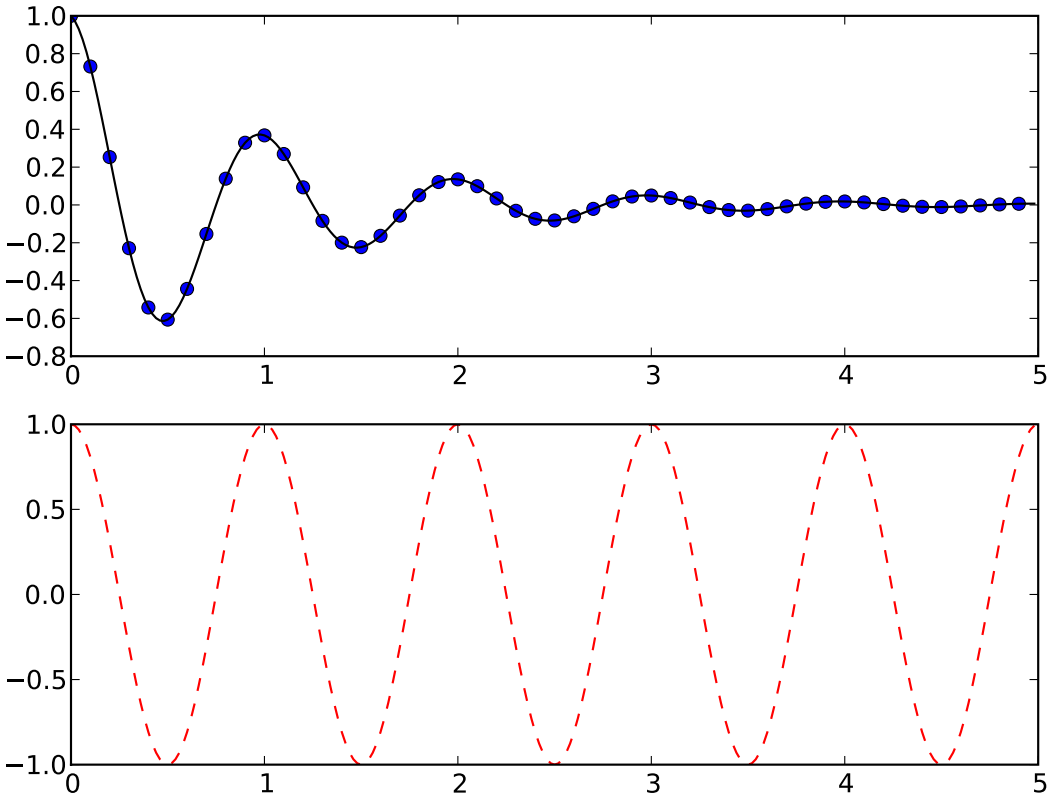
```
In [69]: lines = plot([1,2,3])

In [70]: 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 an script to create two subplots.

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

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

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

plt.figure(1)
plt.subplot(211)
plt.plot(t1, f(t1), 'bo', t2, f(t2), 'k')

plt.subplot(212)
plt.plot(t2, np.cos(2*np.pi*t2), 'r--')
```



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 axes_demo.py for an example of placing axes manually and line_styles.py 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
```

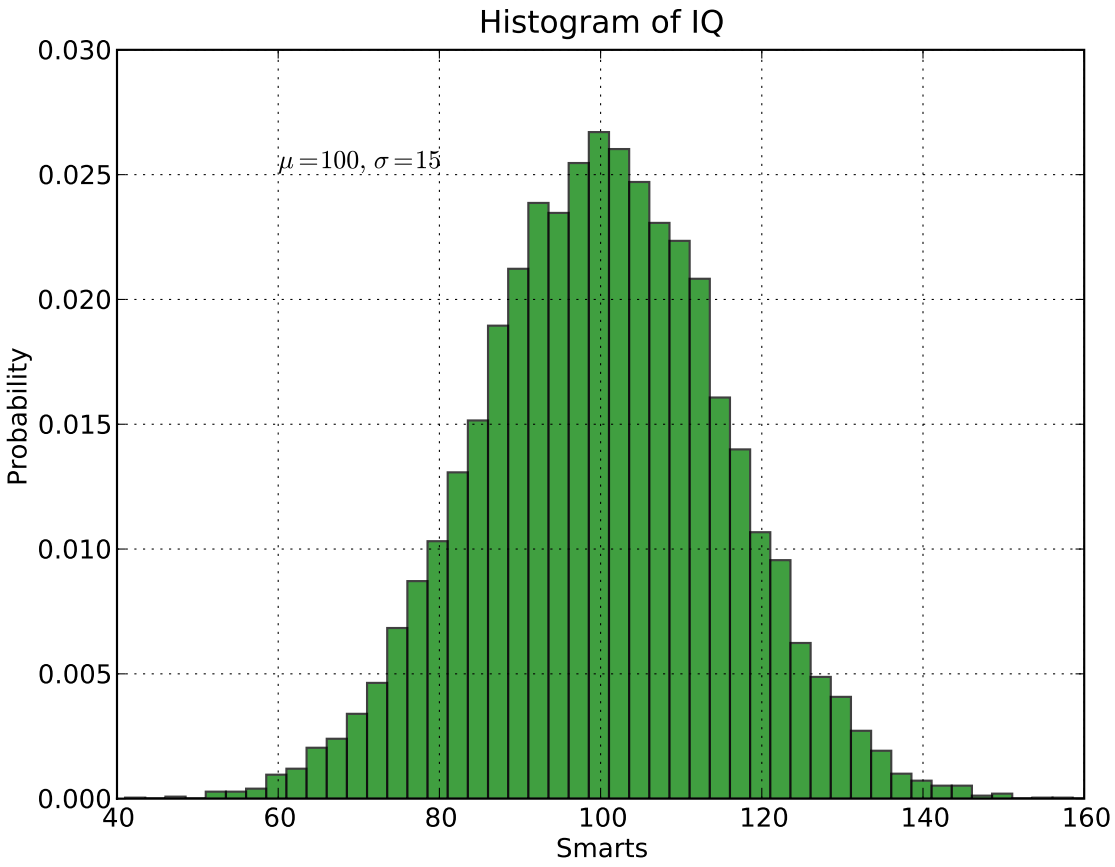
```
plt.figure(1)                # figure 1 current; subplot(212) still current
plt.subplot(211)             # make subplot(211) in figure1 current
plt.title('Easy as 1,2,3')   # subplot 211 title
```

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*)

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)

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

mu, sigma = 100, 15
x = mu + sigma * np.random.randn(10000)

# the histogram of the data
n, bins, patches = plt.hist(x, 50, normed=1, facecolor='g', alpha=0.75)


plt.xlabel('Smarts')
plt.ylabel('Probability')
plt.title('Histogram of IQ')
plt.text(60, .025, r'$\mu=100,\ \sigma=15$')
plt.axis([40, 160, 0, 0.03])
plt.grid(True)
```



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:

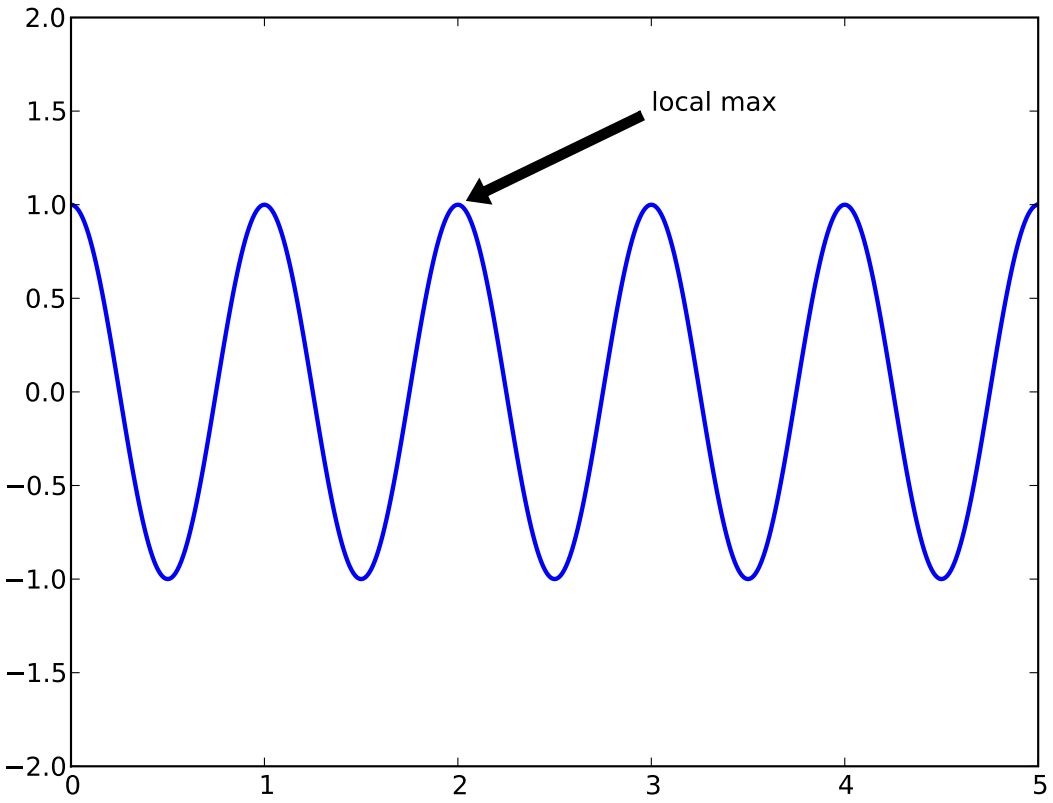
```
plt.title(r'$\sigma_i=15$')
```

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.

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

ax = plt.subplot(111)

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

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

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



In this basic example, both the `xy` (arrow tip) and `xytext` locations (text location) are in data coordinates. There are a variety of other coordinate systems one can choose – see *Annotating text* for details. More examples can be found in the annotations 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 contrained 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 yaxis 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 axes, the y axes, 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`.

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 *Where is my .matplotlib directory?*.
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 you `.matplotlib` directory.

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

Include file u'../mpl_data/matplotlibrc' not found or reading it failed

CHAPTER

SIX

Working with text

6.1 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.

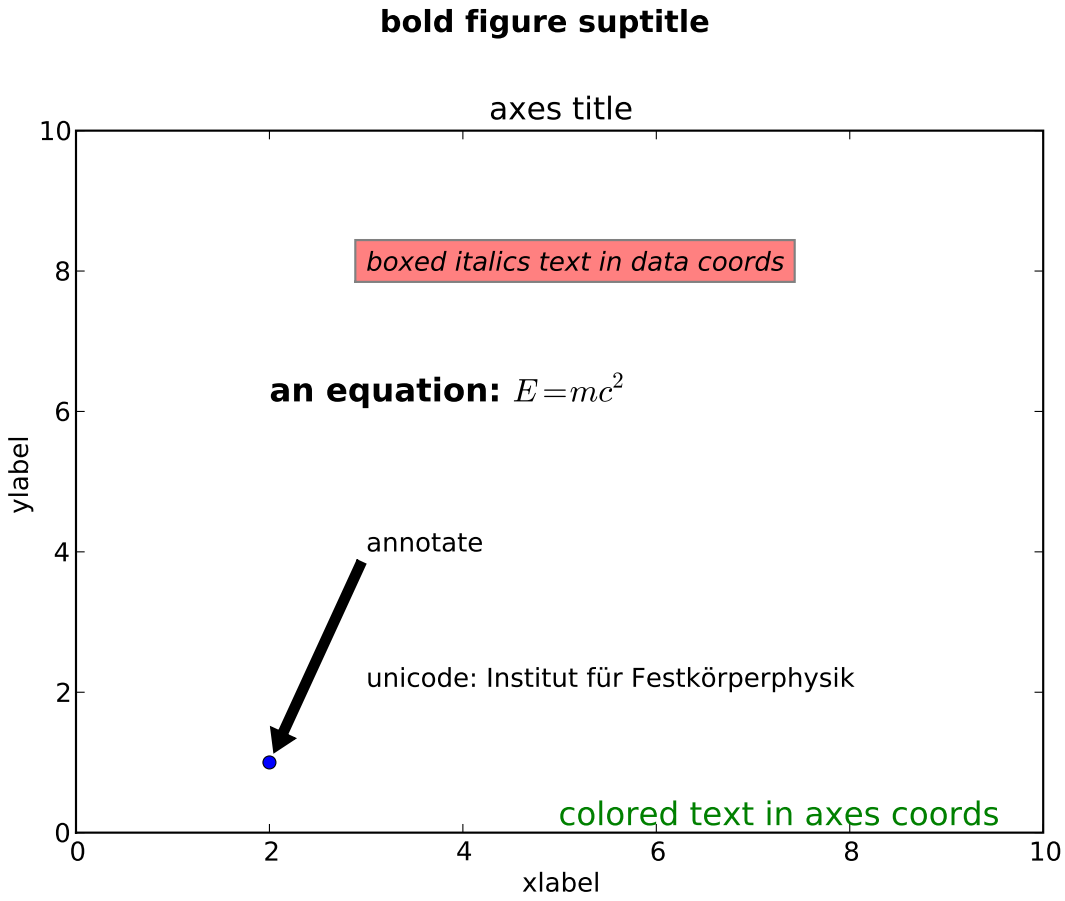
6.2 Basic text commands

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.

```
# -*- coding: utf-8 -*-
import matplotlib.pyplot as plt

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

ax = fig.add_subplot(111)
fig.subplots_adjust(top=0.85)
ax.set_title('axes title')

ax.set_xlabel('xlabel')
ax.set_ylabel('ylabel')

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

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

ax.text(3, 2, unicode('unicode: Institut f\374r Festk\366rperphysik', 'latin-1'))

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


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

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

plt.show()
```



6.3 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()`).

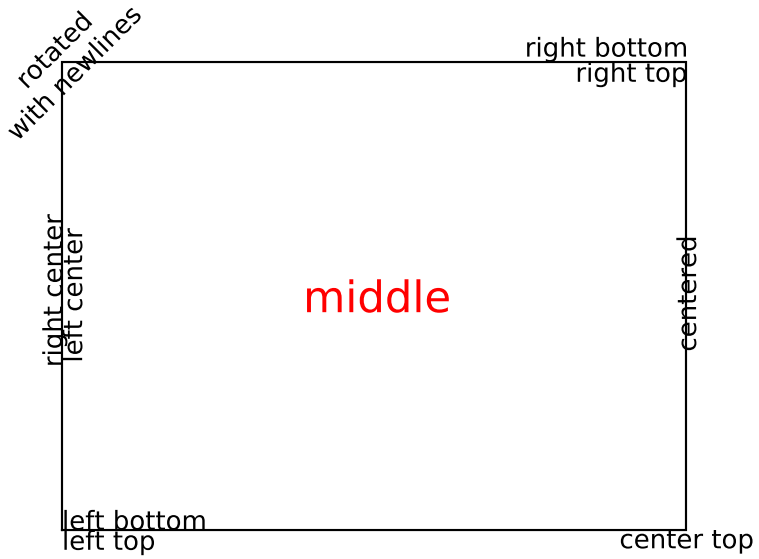
Property	Value Type
alpha	float
backgroundcolor	any matplotlib color
bbox	rectangle prop dict plus key 'pad' which is a pad in points
clip_box	a matplotlib.transform.Bbox instance
clip_on	[True | False]
clip_path	a Path instance and a Transform instance, a Patch
color	any matplotlib color
family	['serif' | '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.

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

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

fig = plt.figure()
```

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

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

ax.add_patch(p)

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

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

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

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

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

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

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

ax.text(0.5*(left+right), 0.5*(bottom+top), 'middle',
        horizontalalignment='center',
        verticalalignment='center',
        fontsize=20, color='red',
        transform=ax.transAxes)
```

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

ax.text(left, top, 'rotated\nwith newlines',
        horizontalalignment='center',
        verticalalignment='center',
        rotation=45,
        transform=ax.transAxes)

ax.set_axis_off()
plt.show()
```



6.4 Writing mathematical expressions

You can use TeX markup in any matplotlib text string. 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 need to use raw strings (preceed the quotes with an `'r'`), and surround the string 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*)

Here is a simple example:

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

produces "alpha > beta".

Whereas this:

```
# math text
plt.title(r'$\alpha > \beta$')
```

produces "$\alpha > \beta$".

6.4.1 Subscripts and superscripts

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

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

$$\alpha_i > \beta_i \tag{6.1}$$

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

$$\sum_{i=0}^{\infty} x_i \tag{6.2}$$

6.4.2 Fractions

Fractions can be created with the `\frac{}{}` command:

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

produces

$$\frac{3}{4} \tag{6.3}$$

Fractions can be arbitrarily nested:

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

produces

$$\frac{5 - \frac{1}{x}}{4} \tag{6.4}$$

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

$$(\frac{5 - \frac{1}{x}}{4}) \tag{6.5}$$

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

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

$$\left(\frac{5 - \frac{1}{x}}{4}\right) \tag{6.6}$$

6.4.3 Radicals

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

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

$$\sqrt{2} \tag{6.7}$$

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

$$\sqrt[3]{x} \tag{6.8}$$

6.4.4 Fonts

The default font is *italics* for mathematical symbols. To change fonts, eg, to write "sin" in a Roman font, enclose the text in a font command:

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

$$s(t) = \mathcal{A}\mathrm{sin}(2\omega t) \tag{6.9}$$

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

$$s(t) = \mathcal{A}\sin(2\omega t) \tag{6.10}$$

Here "s" and "t" are variable in italics font (default), "sin" is in Roman font, and the amplitude "A" is in calligraphy font.

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}`	$\mathbb{blackboard}$
`\mathrm{\mathbb{blackboard}}`	$\mathrm{\mathbb{blackboard}}$
`\mathfrak{Fraktur}`	$\mathfrak{Fraktur}$
`\mathsf{sansserif}`	sansserif
`\mathrm{\mathsf{sansserif}}`	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**

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

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.

6.4.5 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`	$\vec{a}$

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

$$\hat{i}\ \hat{\imath} \tag{6.11}$$

6.4.6 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	F \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	$\mho$ \mho
∇ \nabla					

Hebrew

$\aleph$ \aleph	$\beth$ \beth	$\daleth$ \daleth	$\gimel$ \gimel

Delimiters

/ /	[[	$\Downarrow$ \Downarrow	$\Uparrow$ \Uparrow	$\Vert$ \Vert	\ \backslash
$\downarrow$ \downarrow	$\langle$ \langle	$\lceil$ \lceil	$\lfloor$ \lfloor	$\llcorner$ \llcorner	$\lrcorner$ \lrcorner
$\rangle$ \rangle	$\rceil$ \rceil	$\rfloor$ \rfloor	$\ulcorner$ \ulcorner	$\uparrow$ \uparrow	$\urcorner$ \urcorner
$\vert$ \vert	{ \{	$\Vert$ \\|	} \}	]]	| |

Big symbols

$\bigcap$ \bigcap	$\bigcup$ \bigcup	$\bigodot$ \bigodot	$\bigoplus$ \bigoplus	$\bigotimes$ \bigotimes
$\biguplus$ \biguplus	$\bigvee$ \bigvee	$\bigwedge$ \bigwedge	$\coprod$ \coprod	$\int$ \int
$\oint$ \oint	$\prod$ \prod	$\sum$ \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	

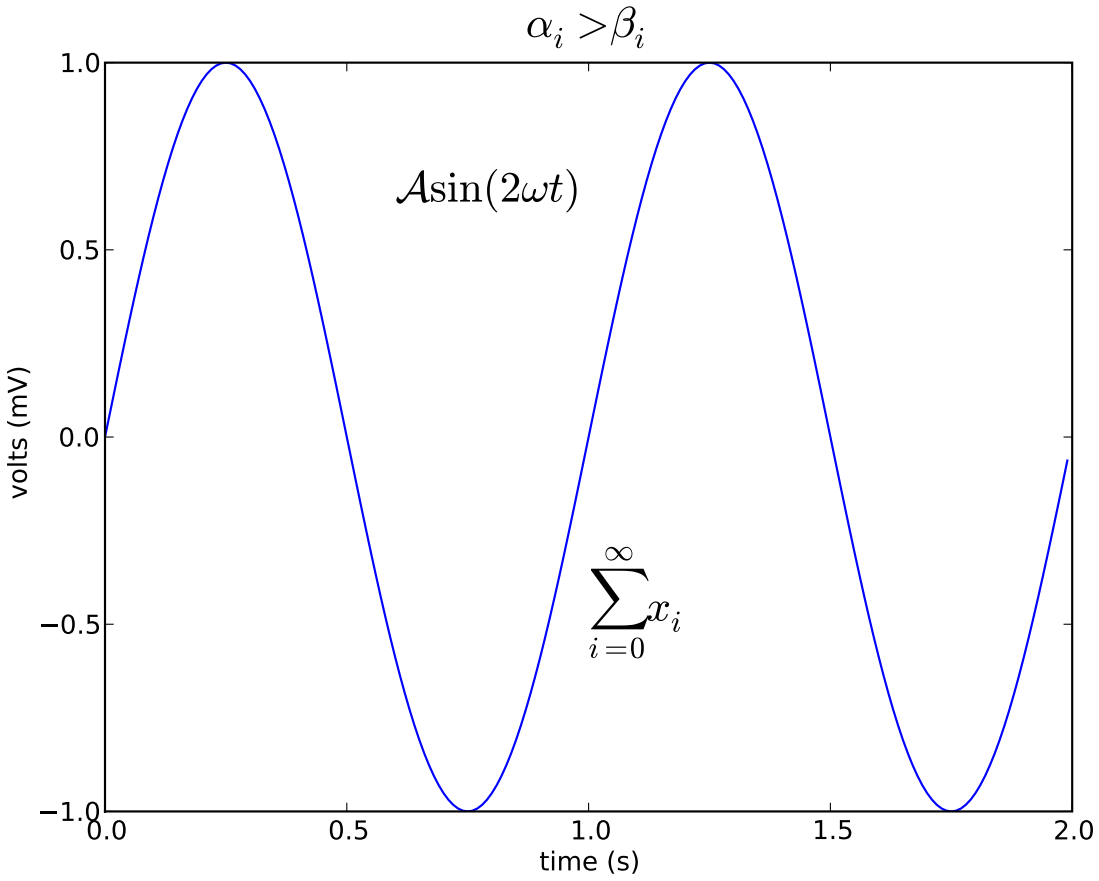
If a particular symbol does not have a name (as is true of many of the more obscure symbols in the STIX fonts), Unicode characters can also be used:

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

6.4.7 Example

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

```
import numpy as np
import matplotlib.pyplot as plt
t = np.arange(0.0, 2.0, 0.01)
s = np.sin(2*np.pi*t)

plt.plot(t,s)
plt.title(r'$\alpha_i > \beta_i$', fontsize=20)
plt.text(1, -0.6, r'$\sum_{i=0}^\infty x_i$', fontsize=20)
plt.text(0.6, 0.6, r'$\mathcal{A}\mathrm{sin}(2 \omega t)$',
         fontsize=20)
plt.xlabel('time (s)')
plt.ylabel('volts (mV)')
```



6.5 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.

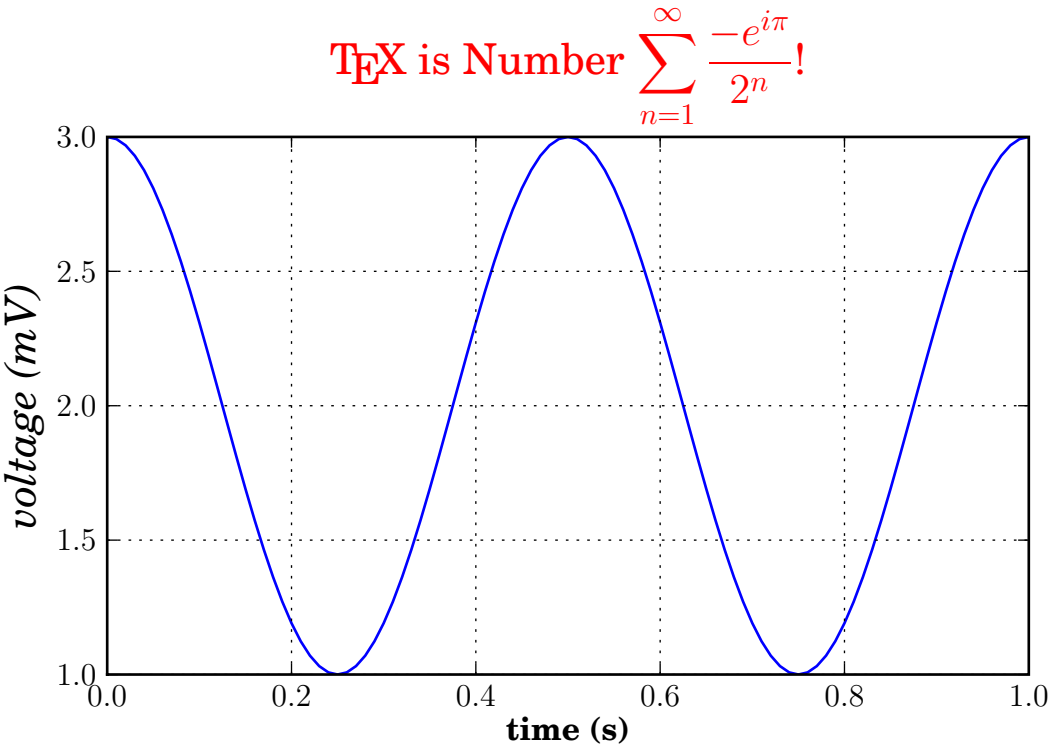
To use LaTeX and select Helvetica as the default font, without editing matplotlibrc use:

```
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*:

```
from matplotlib import rc
from numpy import arange, cos, pi
from matplotlib.pyplot import figure, axes, plot, xlabel, ylabel, title, \
     grid, savefig, show
```

```
rc('text', usetex=True)
rc('font', family='serif')
figure(1, figsize=(6,4))
ax = axes([0.1, 0.1, 0.8, 0.7])
t = arange(0.0, 1.0+0.01, 0.01)
s = cos(2*2*pi*t)+2
plot(t, s)

xlabel(r'\textbf{time (s)}')
ylabel(r'\textit{voltage (mV)}',fontsize=16)
title(r"\TeX\ is Number $\displaystyle\sum_{n=1}^\infty\frac{-e^{i\pi}}{2^n}$!",
      fontsize=16, color='r')
grid(True)
```



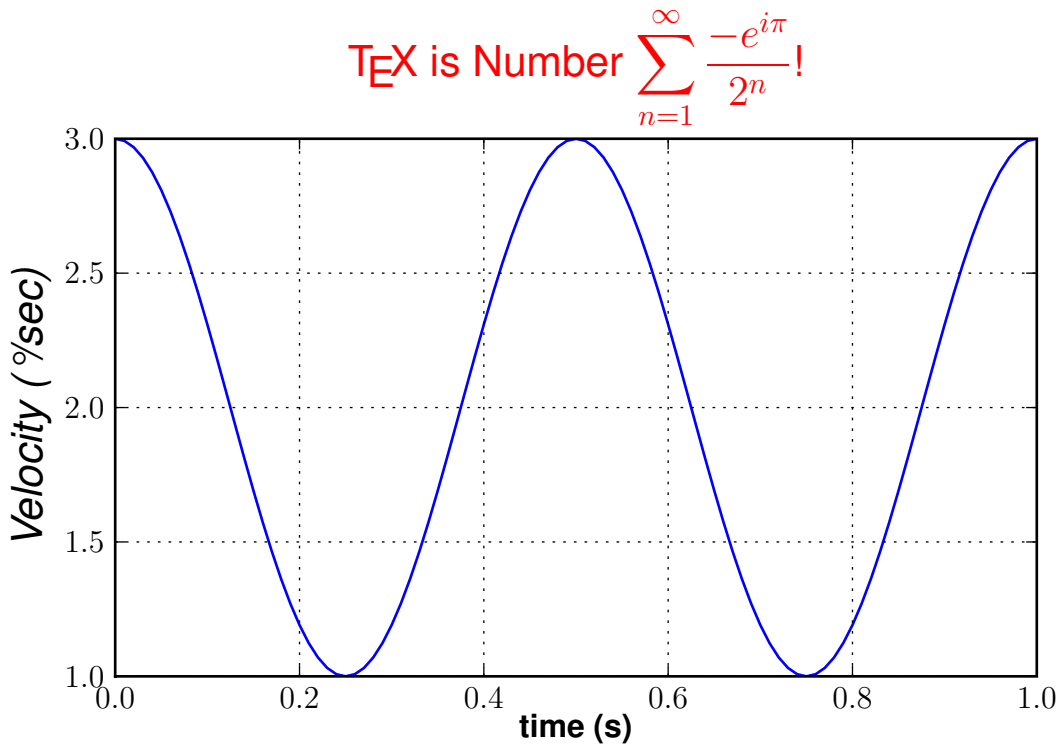
Note that display math mode (`$$ e=mc^2 $$`) is not supported, but adding the command `\displaystyle`, as in *tex_demo.py*, will produce the same results.

6.5.1 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*:

```
# -*- coding: latin-1 -*-
from matplotlib import rcParams
rcParams['text.usetex']=True
rcParams['text.latex.unicode']=True
from numpy import arange, cos, pi
```

```
from matplotlib.pyplot import figure, axes, plot, xlabel, ylabel, title, \
     grid, savefig, show

figure(1, figsize=(6,4))
ax = axes([0.1, 0.1, 0.8, 0.7])
t = arange(0.0, 1.0+0.01, 0.01)
s = cos(2*2*pi*t)+2
plot(t, s)

xlabel(r'\textbf{time (s)}')
ylabel(unicode('\\textit{Velocity (\xB0/sec)}','latin-1'),fontsize=16)
title(r"\TeX\ is Number $\displaystyle\sum_{n=1}^\infty\frac{-e^{i\pi}}{2^n}$!",
      fontsize=16, color='r')
grid(True)
```



6.5.2 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.

6.5.3 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.

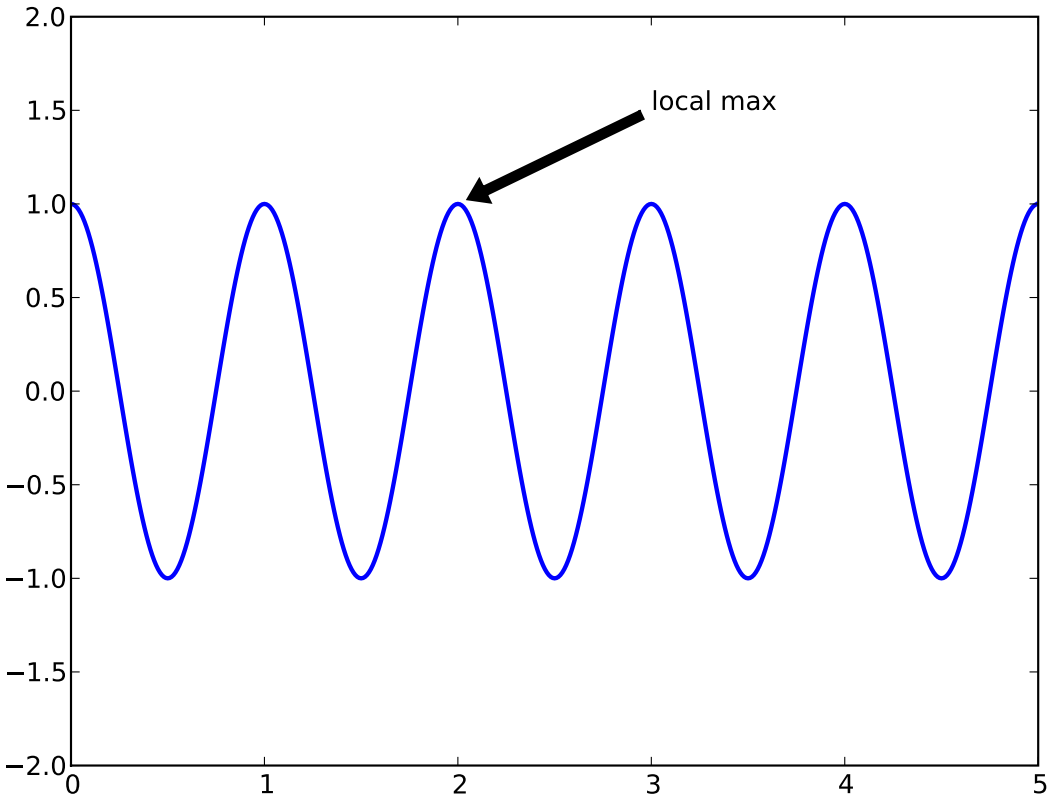
6.5.4 Troubleshooting

- Try deleting your `.matplotlib/tex.cache` directory. If you don't know where to find `.matplotlib`, see *Where is my .matplotlib directory?*.
- 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 *How do I report a problem?*

6.6 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.

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

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

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

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

ax.set_ylim(-2,2)
plt.show()
```



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.

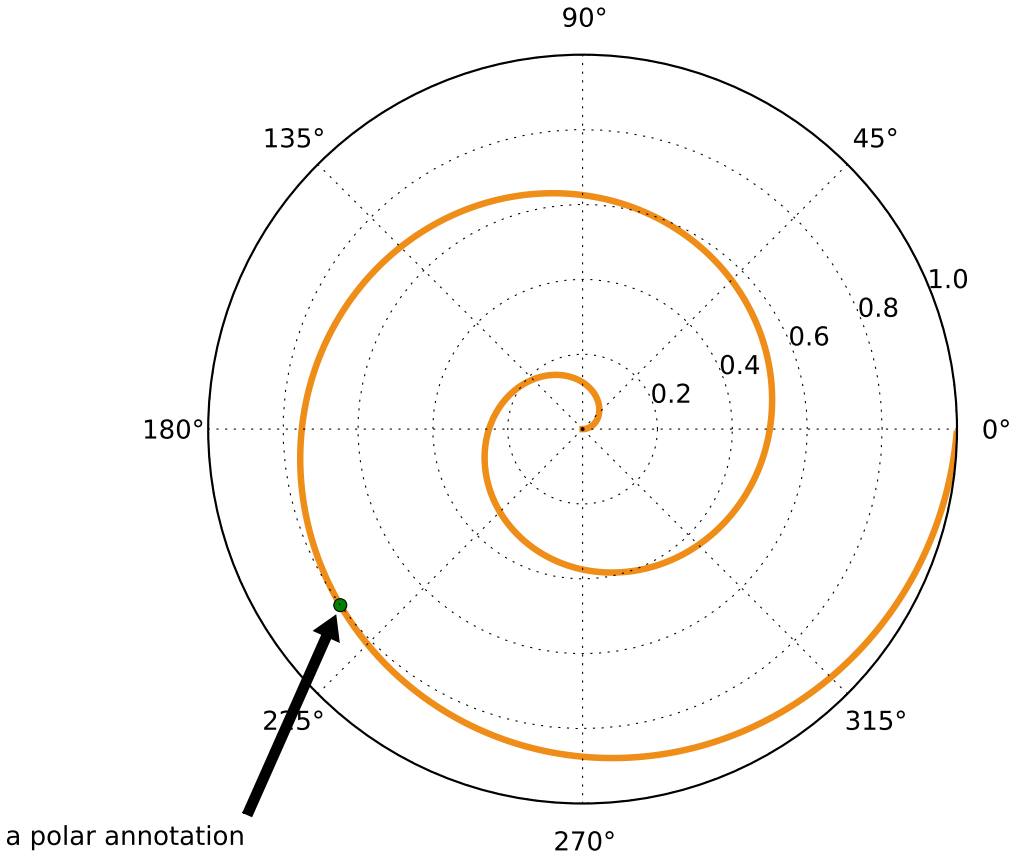
Optionally, you can specify arrow properties which draws an arrow from the text to the annotated point by giving a dictionary of arrow properties in the optional keyword argument `arrowprops`.

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`, eg `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

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

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

ind = 800
thisr, thistheta = r[ind], theta[ind]
ax.plot([thistheta], [thisr], 'o')
ax.annotate('a polar annotation',
            xy=(thistheta, thisr),  # theta, radius
            xytext=(0.05, 0.05),    # fraction, fraction
            textcoords='figure fraction',
```

```
            arrowprops=dict(facecolor='black', shrink=0.05),
            horizontalalignment='left',
            verticalalignment='bottom',
            )
plt.show()
```



See the annotations demo for more examples.

CHAPTER

SEVEN

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, and you can work directly with PostScript, PDF Gtk+, or wxPython `FigureCanvas` instances. For example, 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) 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 createsa `Line2D` instance and adds it the `Axes.lines` list. In the interactive ipython session below, you can see that `Axes.lines` list is length one and contains the same line that was returned by the `line, = ax.plot(x, y, 'o')` 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:

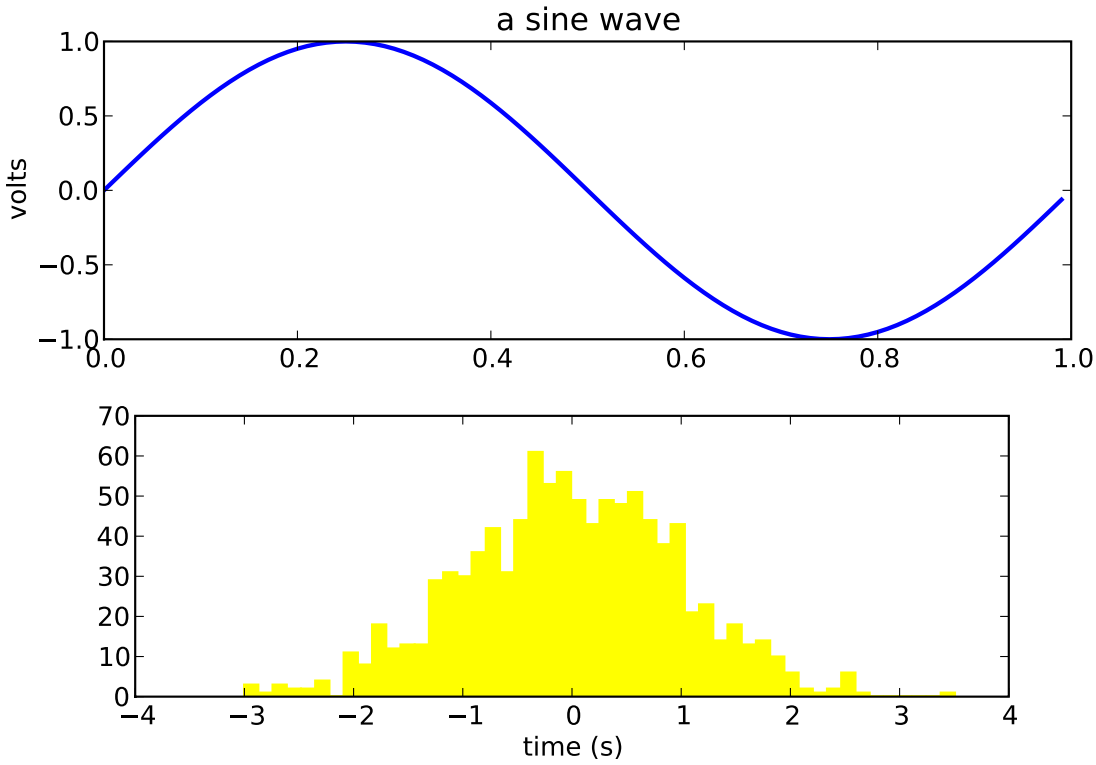
```
del ax.lines[0]
ax.lines.remove(line)  # one or the other, not both!
```

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



7.1 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 aritst 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", the online html docs at http://matplotlib.sourceforge.net/classdocs.html or PDF documentation at http://matplotlib.sourceforge.net/api.pdf for a listing of properties for a give object.

7.2 Object containers

Now that we know how to inspect 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.

7.3 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>]
```

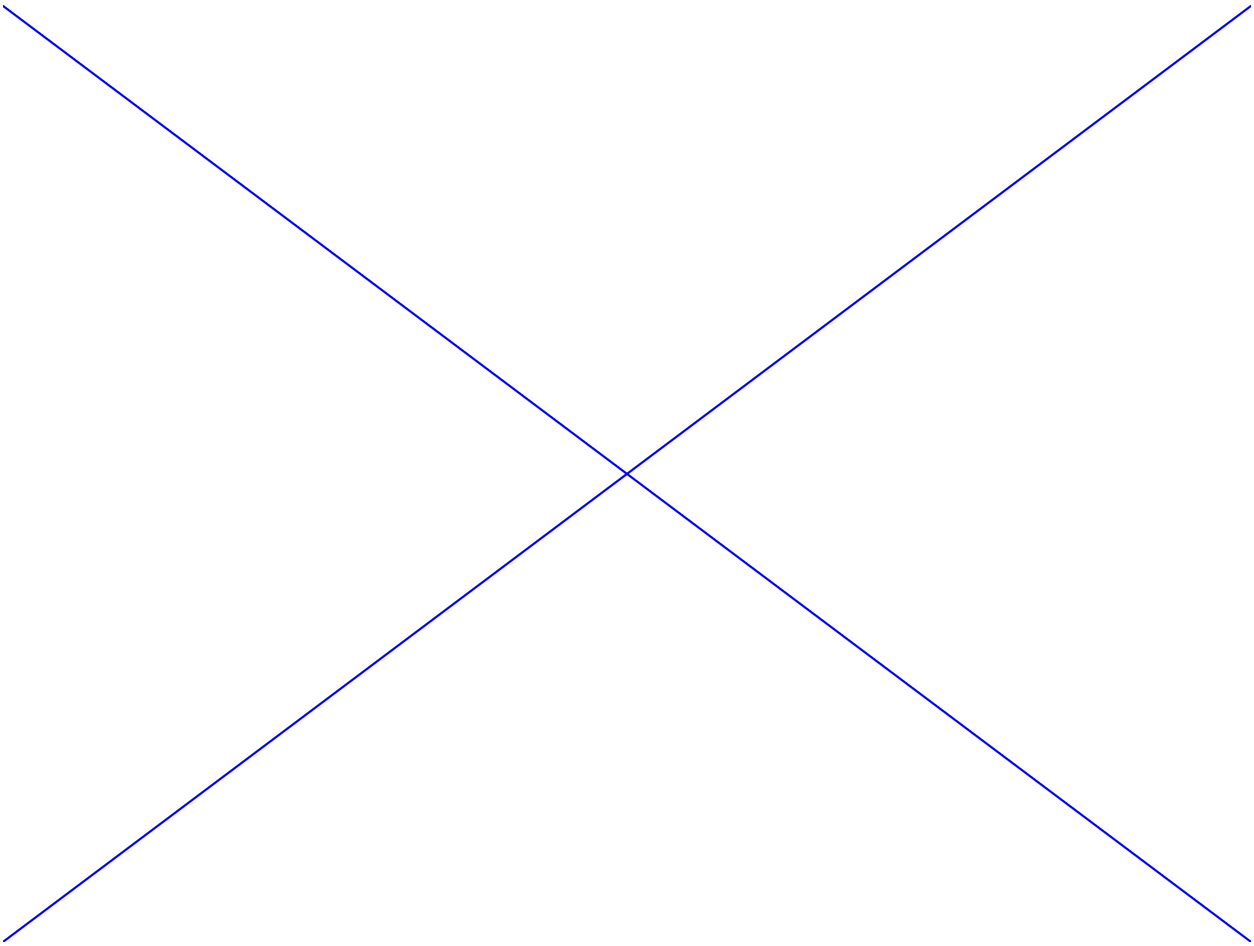
Because the figure maintains the concept of the "current axes" (see `Figure.gca` and `Figure.sca`) to support the pylab/pyplot state machine, you should not insert or remove axes directly from the axes list, but rather use the `add_subplot()` and `add_axes()` methods to insert, and the `delaxes()` method 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

7.4 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()
Subplot(49,81.25)

# 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.get_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

7.5 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 the `pylab` calls to `xlabel()` and `ylabel()` set) 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.:

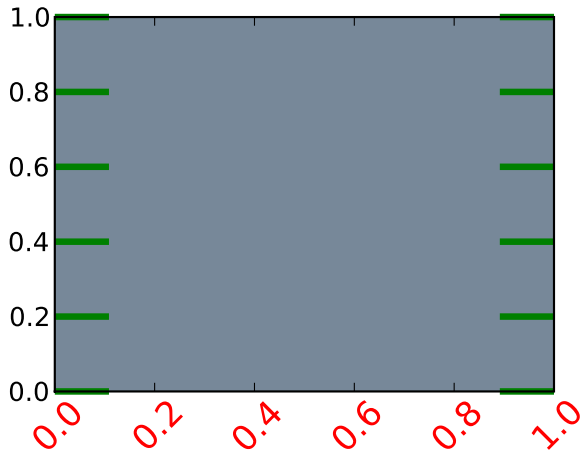
```
In [285]: axis = ax.xaxis

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

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

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

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


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

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



7.6 Tick containers

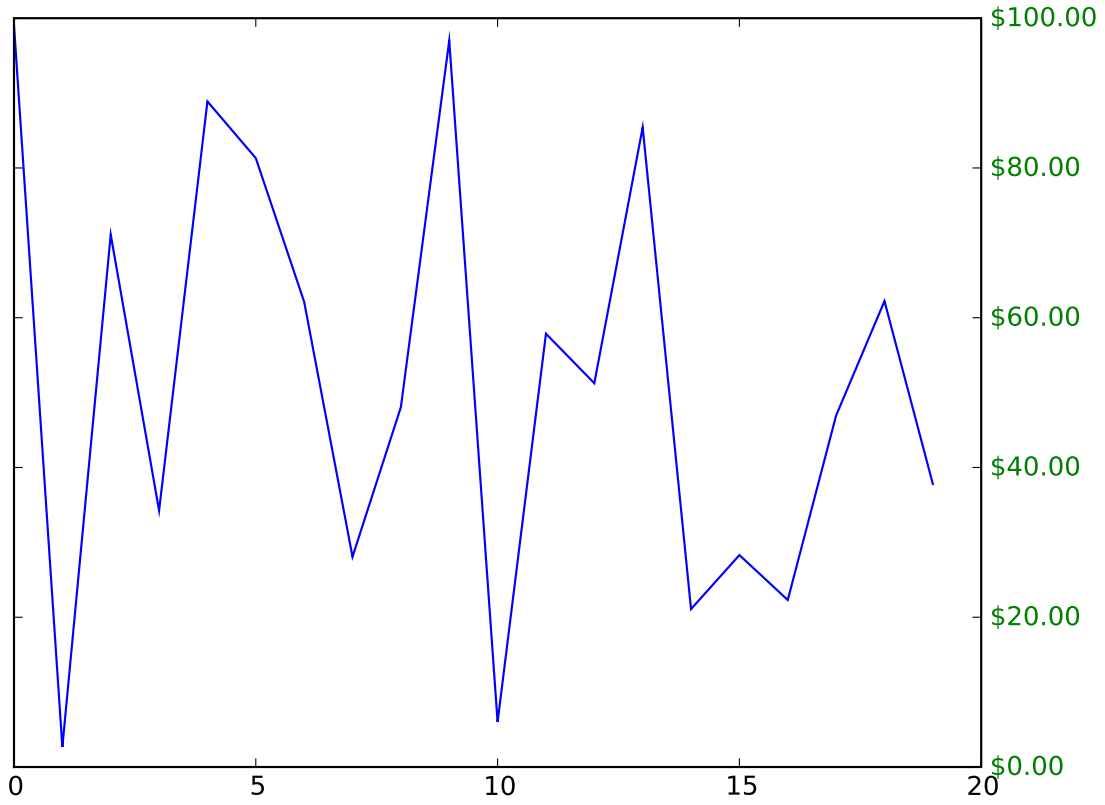
The `matplotlib.axis.Tick` is the final container object in our descent from the `Figure` to the `Axes` to the `Axis` to the `Tick`. The `Tick` contains the tick and grid line instances, as well as the label instances for the upper and lower ticks. Each of these is accessible directly as an attribute 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 upper ticks with dollar signs and colors them green on

the right side of the yaxis

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

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

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

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



CHAPTER

EIGHT

Event handling and picking

matplotlib works with 5 user interface toolkits (wxpython, tkinter, qt, gtk and fltk) 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.

8.1 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)
```

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

8.2 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:

```
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)
```

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, chr(range(255), 'shift', 'win', or 'control'

8.2.1 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 orginal 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, dx, x0+dx)
        self.rect.set_x(x0+dx)
        self.rect.set_y(y0+dy)

        self.rect.figure.canvas.draw()


    def on_release(self, event):
        'on release we reset the press data'
        self.press = None
        self.rect.figure.canvas.draw()

    def disconnect(self):
        'disconnect all the stored connection ids'
        self.rect.figure.canvas.mpl_disconnect(self.cidpress)
        self.rect.figure.canvas.mpl_disconnect(self.cidrelease)
        self.rect.figure.canvas.mpl_disconnect(self.cidmotion)

fig = plt.figure()
ax = fig.add_subplot(111)
rects = ax.bar(range(10), 20*np.random.rand(10))
drs = []
for rect in rects:
    dr = DraggableRectangle(rect)
    dr.connect()
    drs.append(dr)

plt.show()
```

Extra credit: use the animation blit techniques discussed in the animations recipe to make the animated drawing faster and smoother.

Extra credit solution:

```
# 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()
```

8.3 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)

8.3.1 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()
```

8.3.2 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()
```

Part II

The Matplotlib FAQ

CHAPTER

NINE

Installation

Contents

9.1 How do I report a compilation problem?

See *How do I report a problem?*.

9.2 matplotlib compiled fine, but I can't get anything to plot

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`,

or 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 *How do I report a problem?*.

9.3 How do I cleanly rebuild and reinstall everything?

The steps depend on your platform and installation method.

9.3.1 Easy Install

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

   ```
   easy_install -m PackageName
   ```

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

9.3.2 Windows installer

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

9.3.3 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 <locating-matplotlib-install>*

9.4 Backends

9.4.1 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 capabililities is called a backend (the "frontend" is the user facing code, ie the plotting code, whereas the "backend" does all the dirty work behind the scenes to make the figure. There are two types of backends: user interface backends (for use in pygtk, wxpython, tkinter, qt or fltk) and hardcopy backends to make image files (PNG, SVG, PDF, PS).

There are a two primary ways to configure your backend. One is to set the `backend` parameter in you `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`.

If you are unsure what to do, and just want to get cranking, just set your backend to `TkAgg`. This will do the right thing for 95% of the users. It gives you the option of running your scripts in batch or working interactively from the python shell, with the least amount of hassles, and is smart enough to do the right thing when you ask for postscript, or pdf, or other image formats.

If however, you want to write graphical user interfaces, or a web application server (*How do I use 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 antigrain C++ library to make a raster (pixel) image of the figure. All of the user interfaces can be used with agg rendering, eg `WXAgg`, `GTKAgg`, `QTAgg`, `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 represenation 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):

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:

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)
Qt4Agg	Agg rendering to a *Qt4* canvas (requires PyQt4)
FLTKAgg	Agg rendering to a *FLTK* canvas (requires pyFLTK)

9.4.2 How do I compile matplotlib with PyGTK-2.4?

There is a bug in PyGTK-2.4. You need to edit `pygobject.h` to add the `G_BEGIN_DECLS` and `G_END_DECLS` macros, and rename `typename` parameter to `typename_`:

```
-                       const char *typename,
+                       const char *typename_,
```

9.5 OS-X questions

9.5.1 How can I easy_install my egg?

I downloaded the egg for 0.98 from the matplotlib webpages, and I am trying to `easy_install` it, but I am getting an error:

```
> easy_install ./matplotlib-0.98.0-py2.5-macosx-10.3-fat.egg
Processing matplotlib-0.98.0-py2.5-macosx-10.3-fat.egg
removing '/Library/Python/2.5/site-packages/matplotlib-0.98.0-py2.5-
...snip...
Reading http://matplotlib.sourceforge.net
Reading http://cheeseshop.python.org/pypi/matplotlib/0.91.3
```

```
No local packages or download links found for matplotlib==0.98.0
error: Could not find suitable distribution for
Requirement.parse('matplotlib==0.98.0')
```

If you rename `matplotlib-0.98.0-py2.5-macosx-10.3-fat.egg` to `matplotlib-0.98.0-py2.5.egg`, `easy_install` will install it from the disk. Many Mac OS X eggs with cruft at the end of the filename, which prevents their installation through easy_install. Renaming is all it takes to install them; still, it's annoying.

9.6 Windows questions

9.6.1 Where can I get 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 sourceforge download site. Choose the files that match your version of python (eg `py2.5` if you installed Python 2.5) 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

TEN

Troubleshooting

Contents

10.1 What is my matplotlib version?

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

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

10.2 Where is matplotlib installed?

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'
```

10.3 Where is my .matplotlib directory?

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*.

10.4 How do I 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; on Linux/UNIX 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.

10.5 I am having trouble with a recent svn update, what should I do?

First make sure you have a clean build and install (see *How do I cleanly rebuild and reinstall everything?*), get the latest svn update, install it and run a simple test script in debug mode:

```
rm -rf build
rm -rf /path/to/site-packages/matplotlib*
svn up
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 svn 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 *How do I report a problem?*.

CHAPTER

ELEVEN

Howto

Contents

11.1 How do I find all the objects in my 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')
```

11.2 How do I 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 to affect the screen displayed version as well, you can set the alpha properties directly. The figure has a `matplotlib.patches.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)
```

11.3 How do I move the edge of my axes area over to make room for my 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 `matplotlib.figure.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 axes_demo.py for an example of placing axes manually.

11.4 How do I automatically make room for my tick labels?

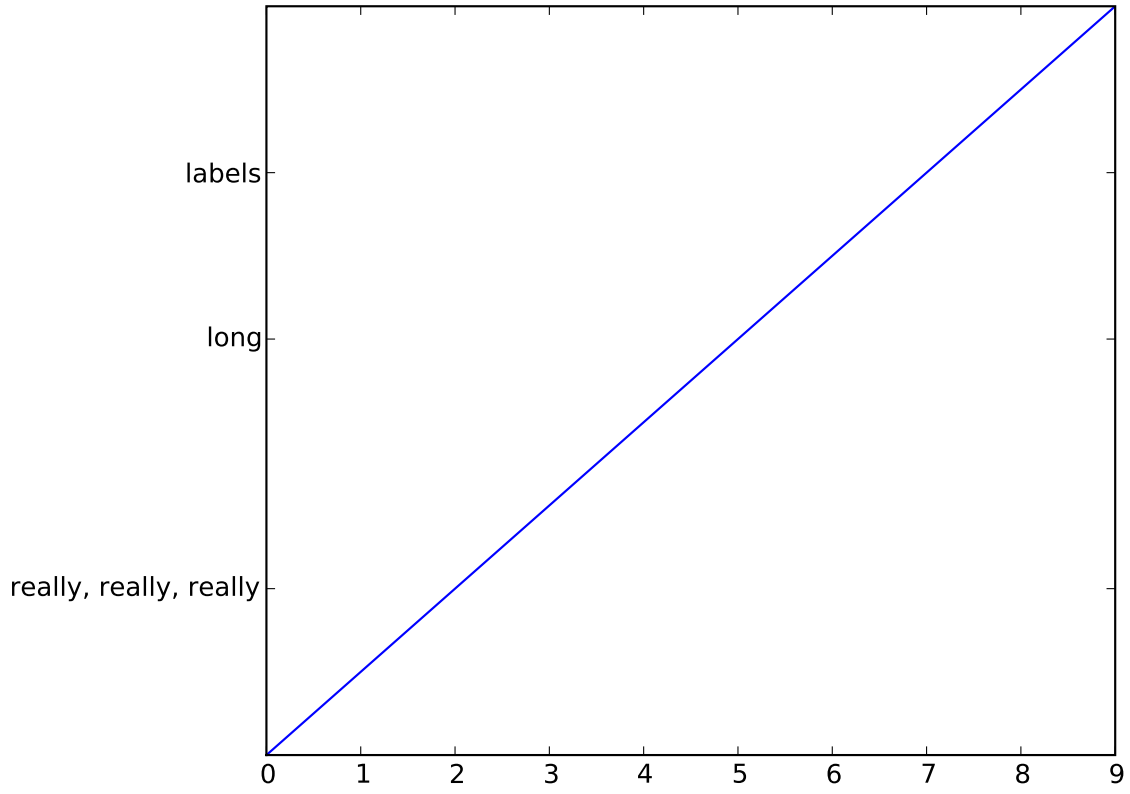
In most use cases, it is enought to simpy change the subplots adjust parameters as described in *How do I move the edge of my axes area over to make room for my tick labels?*. But in some cases, you don't know ahead of time what your tick labels will be, or how large they will be (data and labels outside 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 `matplotlib.text.Text` instance can report its extent in window coordinates (a negative x coordinate is outside the window), but there is a rub.

The `matplotlib.backend_bases.RendererBase` instance, which is used to calculate the text size, is not known until the figure is drawn (`matplotlib.figure.Figure.draw()`). After the window is drawn and the text instance knows its renderer, you can call `matplotlib.text.Text.get_window_extent‘()`. One way to solve this chicken and egg problem is to wait until the figure is draw by connecting (`matplotlib.backend_bases.FigureCanvasBase.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 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

```
import matplotlib.pyplot as plt
import matplotlib.transforms as mtransforms
fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(range(10))
ax.set_yticks((2,5,7))
labels = ax.set_yticklabels(('really, really, really', 'long', 'labels'))

def on_draw(event):
   bboxes = []
   for label in labels:
       bbox = label.get_window_extent()
```

```
        # the figure transform goes from relative coords->pixels and we
        # want the inverse of that
        bboxi = bbox.inverse_transformed(fig.transFigure)
        bboxes.append(bboxi)

    # this is the bbox that bounds all the bboxes, again in relative
    # figure coords
    bbox = mtransforms.Bbox.union(bboxes)
    if fig.subplotpars.left < bbox.width:
        # we need to move it over
        fig.subplots_adjust(left=1.1*bbox.width) # pad a little
        fig.canvas.draw()
    return False

fig.canvas.mpl_connect('draw_event', on_draw)

plt.show()
```



11.5 How do I 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()
```

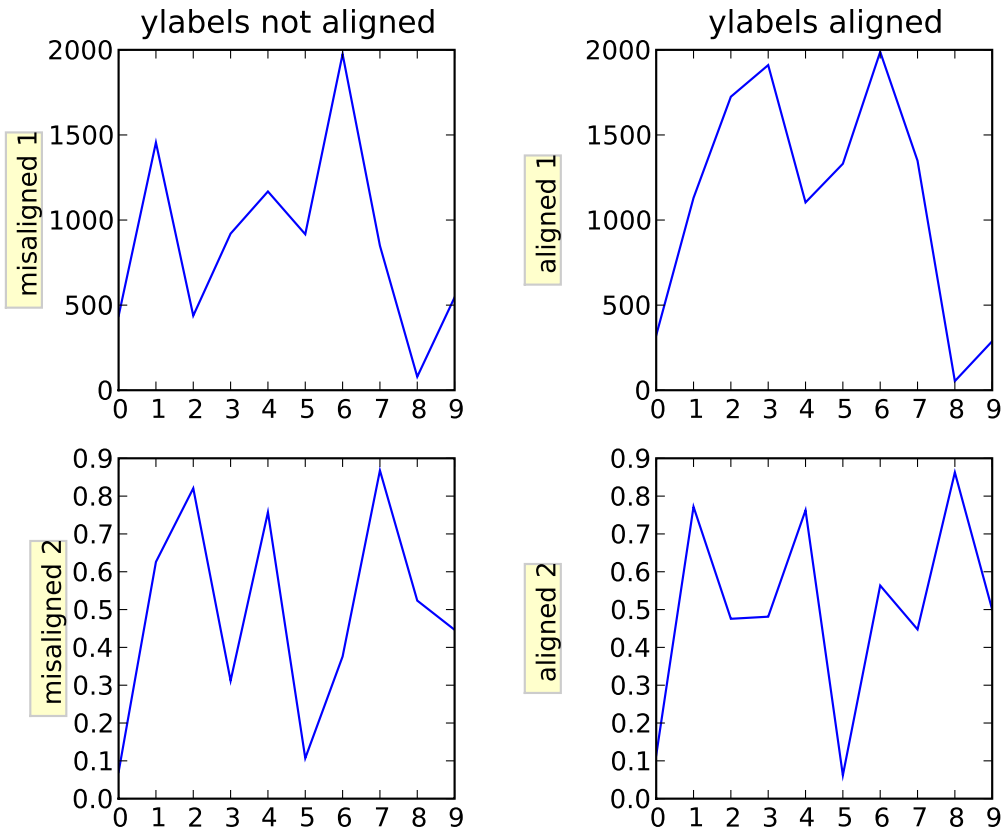
The other properties that control the tick marker, and all markers, are `markerfacecolor`, `markeredgecolor`, `markeredgewidth`, `markersize`. For more information on configuring ticks, see *Axis containers* and *Tick containers*.

11.6 How do I 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.

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

box = dict(facecolor='yellow', pad=5, alpha=0.2)

fig = plt.figure()
fig.subplots_adjust(left=0.2, wspace=0.6)


ax1 = fig.add_subplot(221)
ax1.plot(2000*np.random.rand(10))
ax1.set_title('ylabels not aligned')
ax1.set_ylabel('misaligned 1', bbox=box)
ax1.set_ylim(0, 2000)
ax3 = fig.add_subplot(223)
ax3.set_ylabel('misaligned 2',bbox=box)
ax3.plot(np.random.rand(10))


labelx = -0.3  # axes coords

ax2 = fig.add_subplot(222)
ax2.set_title('ylabels aligned')
```

```
ax2.plot(2000*np.random.rand(10))
ax2.set_ylabel('aligned 1', bbox=box)
ax2.yaxis.set_label_coords(labelx, 0.5)
ax2.set_ylim(0, 2000)

ax4 = fig.add_subplot(224)
ax4.plot(np.random.rand(10))
ax4.set_ylabel('aligned 2', bbox=box)
ax4.yaxis.set_label_coords(labelx, 0.5)


plt.show()
```



11.7 How do I use 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 altogeher, which will give you a little more control, by calling the API directly as shown in agg_oo.py .

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)
```

11.7.1 How do I use matplotlib with apache?

TODO

11.7.2 How do I use matplotlib with django?

TODO

11.7.3 How do I use matplotlib with zope?

TODO

11.8 How do I 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 indicies, 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()
```

CHAPTER

TWELVE

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 by default.

12.1 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
```

12.1.1 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.

12.1.2 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.

12.2 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's Guide

CHAPTER

THIRTEEN

Coding guide

13.1 Version control

13.1.1 svn checkouts

Checking out everything in the trunk (matplotlib and toolkits):

```
svn co https://matplotlib.svn.sourceforge.net/svnroot/matplotlib/trunk \
matplotlib --username=youruser --password=yourpass
```

Checking out the main source:

```
svn co https://matplotlib.svn.sourceforge.net/svnroot/matplotlib/trunk/\
matplotlib mpl --username=youruser --password=yourpass
```

Branch checkouts, eg the maintenance branch:

```
svn co https://matplotlib.svn.sourceforge.net/svnroot/matplotlib/branches/\
v0_91_maint mpl91 --username=youruser --password=yourpass
```

13.1.2 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 `API_CHANGES`, and consider posting to matplotlib-devel
- Are your changes python2.4 compatible? We still support 2.4, so avoid features new to 2.5
- Can you pass `examples/tests/backend_driver.py`? This is our poor man's unit test.
- If you have altered extension code, do you pass `unit/memleak_hawaii.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 branch and trunk in sync where it makes sense. If there is a bug on both that needs fixing, use svnmerge.py to keep them in sync. The basic procedure is:
  - install `svnmerge.py` in your PATH:

    ```
    > wget http://svn.collab.net/repos/svn/trunk/contrib/client-side/\
    svnmerge/svnmerge.py
    ```

  - get a svn copy of the maintenance branch and the trunk (see above)
  - Michael advises making the change on the branch and committing it. Make sure you svn upped on the trunk and have no local modifications, and then from the svn trunk do:

    ```
    > svnmerge.py merge
    ```

    If you wish to merge only specific revisions (in an unusual situation), do:

    ```
    > svnmerge.py merge -rNNN1-NNN2
    ```

    where the `NNN` are the revision numbers. Ranges are also acceptable.
    The merge may have found some conflicts (code that must be manually resolved). Correct those conflicts, build matplotlib and test your choices. If you have resolved any conflicts, you can let svn clean up the conflict files for you:

    ```
    > svn -R resolved .
    ```

    `svnmerge.py` automatically creates a file containing the commit messages, so you are ready to make the commit:

    ```
    > svn commit -F svnmerge-commit-message.txt
    ```

13.2 Style guide

13.2.1 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
```

13.2.2 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 checking changes into svn.

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 svn 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)))
```

13.2.3 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...
```

13.3 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.

13.4 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.

13.4.1 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 different 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 sagemath, linux, gcc and emacs.

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 two primary reasons why early matplotlib developers selected a BSD compatible license. 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 in general, who use matplotlib often make significant contributions, since they have the resources to get a job done, even a boring one, if they need it in their code. Two of the matplotlib backends (FLTK and WX) were contributed by private companies.

The other reason is licensing 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

FOURTEEN

Documenting matplotlib

14.1 Getting started

The documentation for matplotlib is generated from ReStructured Text using the Sphinx documentation generation tool. Sphinx-0.4 or later is required. Currently this means we need to install from the svn repository by doing:

```
svn co http://svn.python.org/projects/doctools/trunk sphinx
cd sphinx
python setup.py install
```

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/`.

14.2 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
```

14.3 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:

  $$\int_{-\infty}^{\infty}\frac{e^{i\phi}}{1+x^2\frac{e^{i\phi}}{1+x^2}} \tag{14.1}$$

- 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 release 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.

14.3.1 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.

14.4 Figures

14.4.1 Dynamically generated figures

The top level `doc` dir has a folder called `pyplots` in which you should include any pyplot plotting scripts that you want to generate figures for the documentation. It is not necessary to explicitly save the figure in the script, this will be done automatically at build time to insure that the code that is included runs and produces the advertised figure. Several figures will be saved with the same basnename as the filename when the documentation is generated (low and high res PNGs, a PDF). Matplotlib includes a Sphinx extension (`sphinxext/plot_directive.py`) for generating the images from the python script and including either a png copy for html or a pdf for latex:

```
.. plot:: pyplot_simple.py
   :include-source:
```

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. `:include-source:` will present the contents of the file, marked up as source code.

14.4.2 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 svn. 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:: tex_unicode_demo.py
   :include-source
```

14.5 Referring to mpl documents

In the documentation, you may want to include to a document in the matplotlib src, e.g. a license file, an image file from *mpl-data*, or an example. When you include these files, include them using a symbolic link from the documentation parent directory. This way, if we relocate the mpl documentation directory, all of the internal pointers to files will not have to change, just the top level symlinks. For example, In the top level doc directory we have symlinks pointing to the mpl *examples* and *mpl-data*:

```
home:~/mpl/doc2> ls -l mpl_*
mpl_data -> ../lib/matplotlib/mpl-data
mpl_examples -> ../examples
```

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:: ../mpl_data/matplotlibrc
```

14.6 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.

14.7 Section names, etc

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

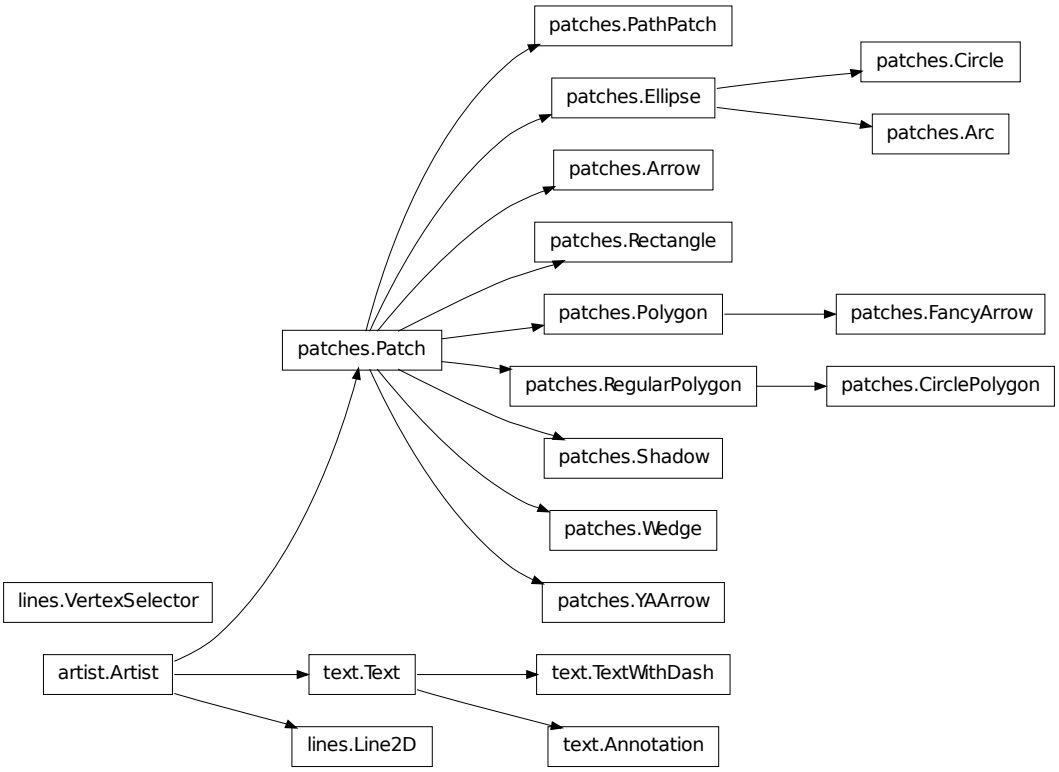
14.8 Inheritance diagrams

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
```



14.9 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

FIFTEEN

Doing a matplolib release

A guide for developers who are doing a matplotlib release

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

15.1 Testing

- Make sure `examples/tests/backend_driver.py` runs without errors and check the output of the PNG, PDF, PS and SVG backends
- 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

15.2 Packaging

- Make sure the `MANIFEST.in` us up to date and remove `MANIFEST` so it will be rebuilt by MANIFEST.in
- run svn-clean from in the mpl svn 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.
- on windows, unix2dos the rc file

15.3 Uploading

- Post the win32 and OS-X binaries for testing and make a request on matplotlib-devel for testing. Pester us if we don't respond

- ftp the source and binaries to the anonymous FTP site:

```
mpl> svn-clean
mpl> python setup.py sdist
mpl> cd dist/
dist> sftp jdh2358@frs.sourceforge.net
Connecting to frs.sourceforge.net...
sftp> cd uploads
sftp> ls
sftp> lls
matplotlib-0.98.2.tar.gz
sftp> put matplotlib-0.98.2.tar.gz
Uploading matplotlib-0.98.2.tar.gz to /incoming/j/jd/jdh2358/uploads/matplotlib-0.98.2.tar.gz
```

- go https://sourceforge.net/project/admin/?group_id=80706 and do a file release. Click on the "Admin" tab to log in as an admin, and then the "File Releases" tab. Go to the bottom and click "add release" and enter the package name but not the version number in the "Package Name" box. You will then be prompted for the "New release name" at which point you can add the version number, eg somepackage-0.1 and click "Create this release".

  You will then be taken to a fairly self explanatory page where you can enter the Change notes, the release notes, and select which packages from the incoming ftp archive you want to include in this release. For each binary, you will need to select the platform and file type, and when you are done you click on the "notify users who are monitoring this package link"

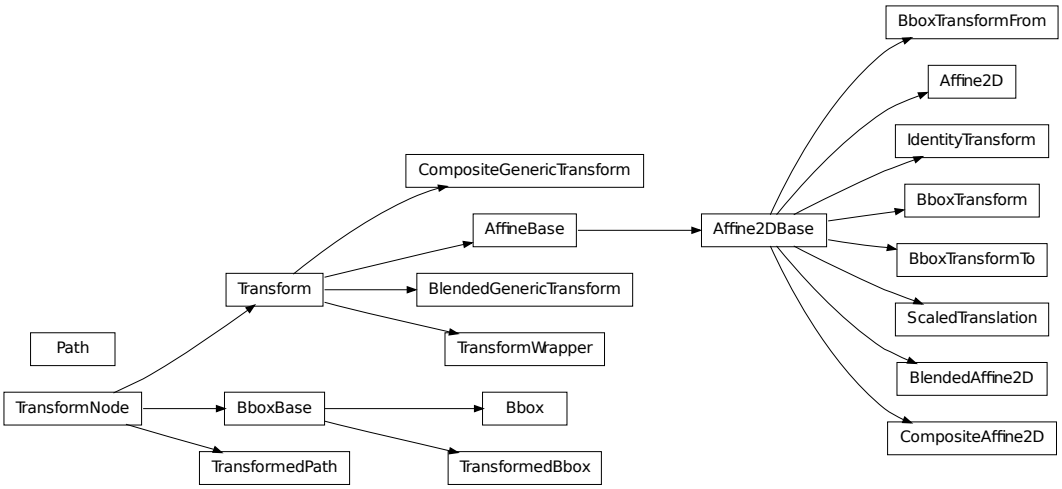
15.4 Announcing

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

SIXTEEN

Working with transformations

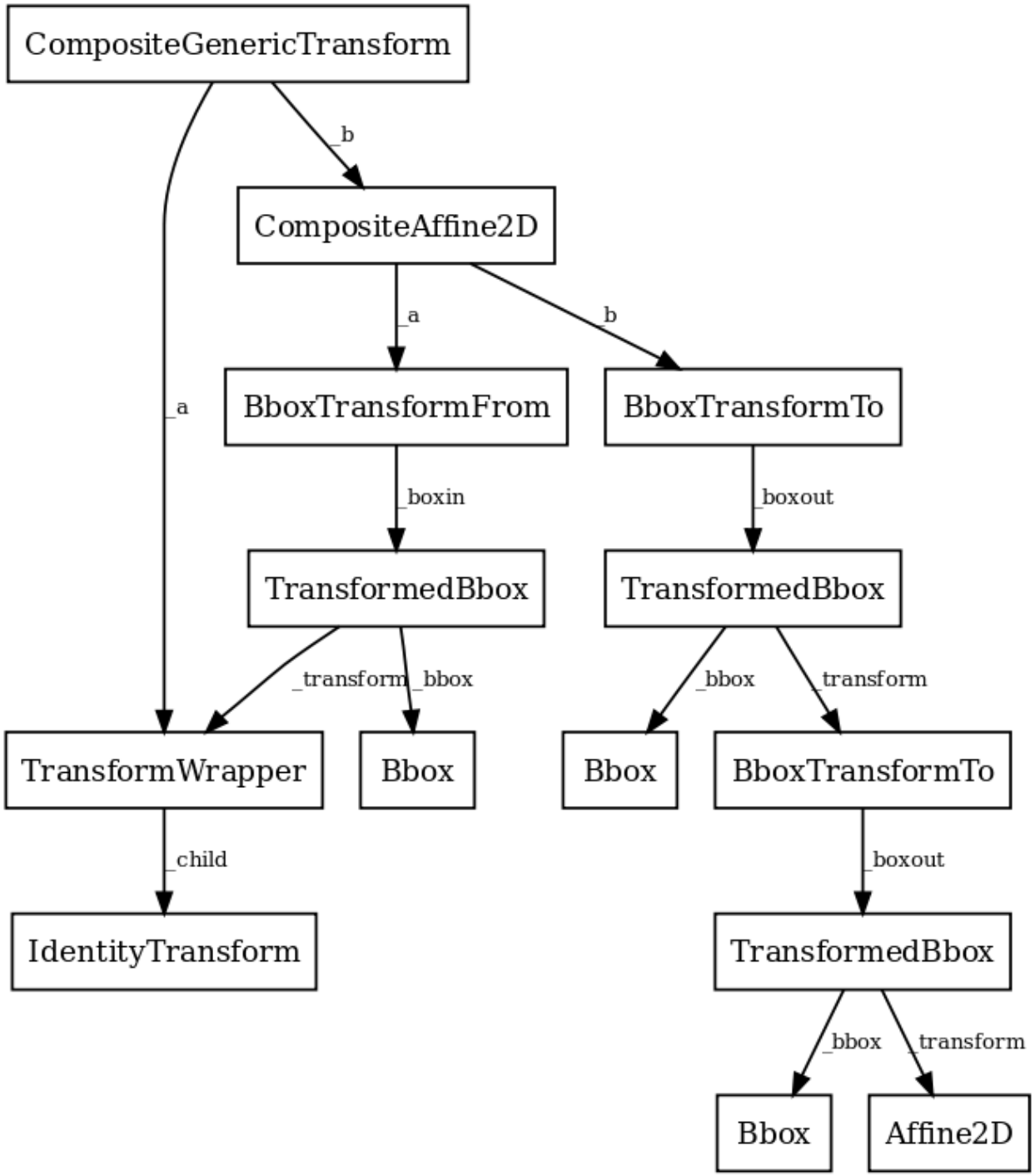


16.1 matplotlib.transforms

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 TransformNode()

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.

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 transform node and all of its ancestors. Should be called any time the transform changes.

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 BboxBase()

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 explicity.

Creates a new TransformNode.

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, 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`, shurnk 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.

union
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 Bbox(*points*)
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.

from_bounds
(staticmethod) Create a new Bbox from x0, y0, width and height.

width and height may be negative.

from_extents
(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.

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.

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.

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*)
Update the bounds of the Bbox based on the passed in data.

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().

class TransformedBbox(*bbox, transform*)
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 Transform()
Bases: matplotlib.transforms.TransformNode

The base class of all TransformNodes 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() can return True)

If the transform needs to do something non-standard with mathplotlib.path.Path objects, such as adding curves where there were once line segments, it should override:

- transform_path()

Creates a new TransformNode.

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(*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(values))`.

In non-affine transformations, this is generally a no-op. In affine transformations, this is equivalent to `transform(values)`.

Accepts a numpy array of shape (N x `input_dims`) and returns a numpy array of shape (N x `output_dims`).

transform_non_affine(*values*)
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(*path*)
Returns a transformed copy of 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 copy of path, transformed only by the affine part of this transform.

path: a Path instance

`transform_path(path)` is equivalent to `transform_path_affine(transform_path_non_affine(values`

transform_path_non_affine(*path*)
Returns a copy of path, transformed only by the non-affine part of this transform.

path: a Path instance

`transform_path(path)` is equivalent to `transform_path_affine(transform_path_non_affine(values`

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 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 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 AffineBase()

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.

get_matrix()

Get the underlying transformation matrix as a numpy array.

transform_non_affine(*points*)

Performs only the non-affine part of the transformation.

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

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

Accepts a numpy array of shape (N x `input_dims`) and returns a numpy array of shape (N x `output_dims`).

transform_path_affine(*path*)

Returns a copy of path, transformed only by the affine part of this transform.

path: a Path instance

`transform_path(path)` is equivalent to `transform_path_affine(transform_path_non_affine(values`

transform_path_non_affine(*path*)

Returns a copy of path, transformed only by the non-affine part of this transform.

path: a Path instance

`transform_path(path)` is equivalent to `transform_path_affine(transform_path_non_affine(values`

class Affine2DBase()

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))`

matrix_from_values
(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(*points*)
Performs only the affine part of this transformation on the given array of values.

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

In non-affine transformations, this is generally a no-op. In affine transformations, this is equivalent to `transform(values)`.

Accepts a numpy array of shape (N x `input_dims`) and returns a numpy array of shape (N x `output_dims`).

transform_affine(*points*)
Performs only the affine part of this transformation on the given array of values.

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

In non-affine transformations, this is generally a no-op. In affine transformations, this is equivalent to `transform(values)`.

Accepts a numpy array of shape (N x `input_dims`) and returns a numpy array of shape (N x `output_dims`).

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 Affine2D(*matrix=None*)
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.

from_values
(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
```

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, :meth:‘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, :meth:‘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, :meth:‘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, :meth:‘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, :meth:‘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, :meth:‘translate() and scale().

class IdentityTransform()
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 underlying transformation matrix as a numpy array.

inverted()
Return the corresponding inverse transformation.

The return value of this method should be treated as temporary. An update to ‘self’ does not cause a corresponding update to its inverted copy.

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

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

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

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

Accepts a numpy array of shape (N 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(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_non_affine(*points*)

Performs only the non-affine part of the transformation.

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

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

Accepts a numpy array of shape (N x `input_dims`) and returns a numpy array of shape (N x `output_dims`).

transform_path(*path*)

Returns a copy of path, transformed only by the non-affine part of this transform.

path: a Path instance

`transform_path(path)` is equivalent to `transform_path_affine(transform_path_non_affine(value`

transform_path_affine(*path*)

Returns a copy of path, transformed only by the non-affine part of this transform.

path: a Path instance

`transform_path(path)` is equivalent to `transform_path_affine(transform_path_non_affine(value`

transform_path_non_affine(*path*)

Returns a copy of path, transformed only by the non-affine part of this transform.

path: a Path instance

`transform_path(path)` is equivalent to `transform_path_affine(transform_path_non_affine(value`

class BlendedGenericTransform(*x_transform, y_transform*)

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(*points*)

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(*points*)

Performs only the affine part of this transformation on the given array of values.

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

In non-affine transformations, this is generally a no-op. In affine transformations, this is equivalent to `transform(values)`.

Accepts a numpy array of shape (N x `input_dims`) and returns a numpy array of shape (N x `output_dims`).

transform_non_affine(*points*)

Performs only the non-affine part of the transformation.

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

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

Accepts a numpy array of shape (N x `input_dims`) and returns a numpy array of shape (N x `output_dims`).

class BlendedAffine2D(*x_transform, y_transform*)

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 underlying transformation matrix as a numpy array.

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 CompositeGenericTransform(*a, b*)

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(*points*)
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(*points*)
Performs only the affine part of this transformation on the given array of values.
`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.
In non-affine transformations, this is generally a no-op. In affine transformations, this is equivalent to `transform(values)`.
Accepts a numpy array of shape (N x `input_dims`) and returns a numpy array of shape (N x `output_dims`).

transform_non_affine(*points*)
Performs only the non-affine part of the transformation.
`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.
In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.
Accepts a numpy array of shape (N x `input_dims`) and returns a numpy array of shape (N x `output_dims`).

transform_path(*path*)
Returns a transformed copy of 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 copy of path, transformed only by the affine part of this transform.
path: a Path instance
`transform_path(path)` is equivalent to `transform_path_affine(transform_path_non_affine(values`

transform_path_non_affine(*path*)
Returns a copy of path, transformed only by the non-affine part of this transform.
path: a Path instance
`transform_path(path)` is equivalent to `transform_path_affine(transform_path_non_affine(values`

class CompositeAffine2D(*a, b*)

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 underlying transformation matrix as a numpy array.

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 BboxTransform(*boxin, boxout*)

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 underlying transformation matrix as a numpy array.

class BboxTransformTo(*boxout*)

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 underlying transformation matrix as a numpy array.

class BboxTransformFrom(*boxin*)

Bases: matplotlib.transforms.Affine2DBase

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

get_matrix()

Get the underlying transformation matrix as a numpy array.

class ScaledTranslation(*xt, yt, scale_trans*)

Bases: matplotlib.transforms.Affine2DBase

A transformation that translates by xt and yt, after xt and yt have been transformaed by the given transform scale_trans.

get_matrix()
Get the underlying transformation matrix as a numpy array.

class 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.

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.

nonsingular(*vmin, vmax, expander=0.001, tiny=1.0000000000000001e-15, increasing=True*)
Ensure the endpoints of a range are not too close together.

"too close" means the interval is smaller than 'tiny' times the maximum absolute value.

If they are too close, each will be moved by the 'expander'. If 'increasing' is True and vmin > vmax, they will be swapped, regardless of whether they are too close.

16.2 matplotlib.path

Contains a class for managing paths (polylines).

class Path(*vertices, codes=None*)
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()`.

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, the resulting path will be compressed, with `MOVETO` codes inserted in the correct places to jump over the masked regions.

arc

(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.

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*)

Returns *True* if the path contains the given point.

If *transform* is not *None*, the path will be transformed before performing the test.

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.

interpolated(*steps*)

Returns a new path resampled to length N x steps. Does not currently handle interpolating curves.

intersects_bbox(*bbox*)

Returns *True* if this path intersects a given `Bbox`.

intersects_path(*other*)

Returns *True* if this path intersects another given path.

iter_segments()

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.

make_compound_path

(staticmethod) Make a compound path from a list of Path objects. Only polygons (not curves) are supported.

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 `matplotlib.transforms.TransformedPath` for a path that will cache the transformed result and automatically update when the transform changes.

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.

unit_rectangle

(staticmethod) Returns a `Path` of the unit rectangle from (0, 0) to (1, 1).

unit_regular_asterisk

(staticmethod) Returns a `Path` for a unit regular asterisk with the given numVertices and radius of 1.0, centered at (0, 0).

unit_regular_polygon

(staticmethod) Returns a `Path` for a unit regular polygon with the given *numVertices* and radius of 1.0, centered at (0, 0).

unit_regular_star

(staticmethod) Returns a `Path` for a unit regular star with the given numVertices and radius of 1.0, centered at (0, 0).

wedge

(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*.

get_path_collection_extents(**args*)

Given a sequence of `Path` objects, returns the bounding box that encapsulates all of them.

CHAPTER

SEVENTEEN

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.

17.1 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.

17.2 Creating a new projection

Adding a new projection consists of defining a subclass of `matplotlib.axes.Axes`, 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.
- 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 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_projection_example.py`. The polar plot functionality in `matplotlib.projections.polar` may also be of interest.

CHAPTER

EIGHTEEN

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	?
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
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
numerix/__init__		needs conversion
projections/__init__		needs conversion
projections/geo		needs conversion
projections/polar		needs conversion
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	needs conversion
fontconfig_pattern	Mike	needs conversion
image		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!

18.1 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.

18.1.1 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

The Matplotlib API

CHAPTER

NINETEEN

Matplotlib configuration

19.1 matplotlib

This is an object-orient plotting library.

A procedural interface is provided by the companion pylab module, which may be imported directly, e.g:

```
from pylab import *
```

or using ipython:

```
ipython -pylab
```

For the most part, direct use of the object-oriented library is encouraged when programming rather than working interactively. The exceptions are the pylab commands `figure()`, `subplot()`, `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

:mod'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 overridden 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 is written by John D. Hunter (jdh2358 at gmail.com) and a host of others.

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.

rcdefaults()

Restore the default rc params - the ones that were created at matplotlib load time.

use(*arg*)

IPython wrapper for matplotlib's backend switcher.

In interactive use, we can not allow switching to a different interactive backend, since thread conflicts will most likely crash the python interpreter. This routine does a safety check first, and refuses to perform a dangerous switch. It still allows switching to non-interactive backends.

CHAPTER

TWENTY

Matplotlib afm

20.1 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 = file('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 AFM(*fh*)

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

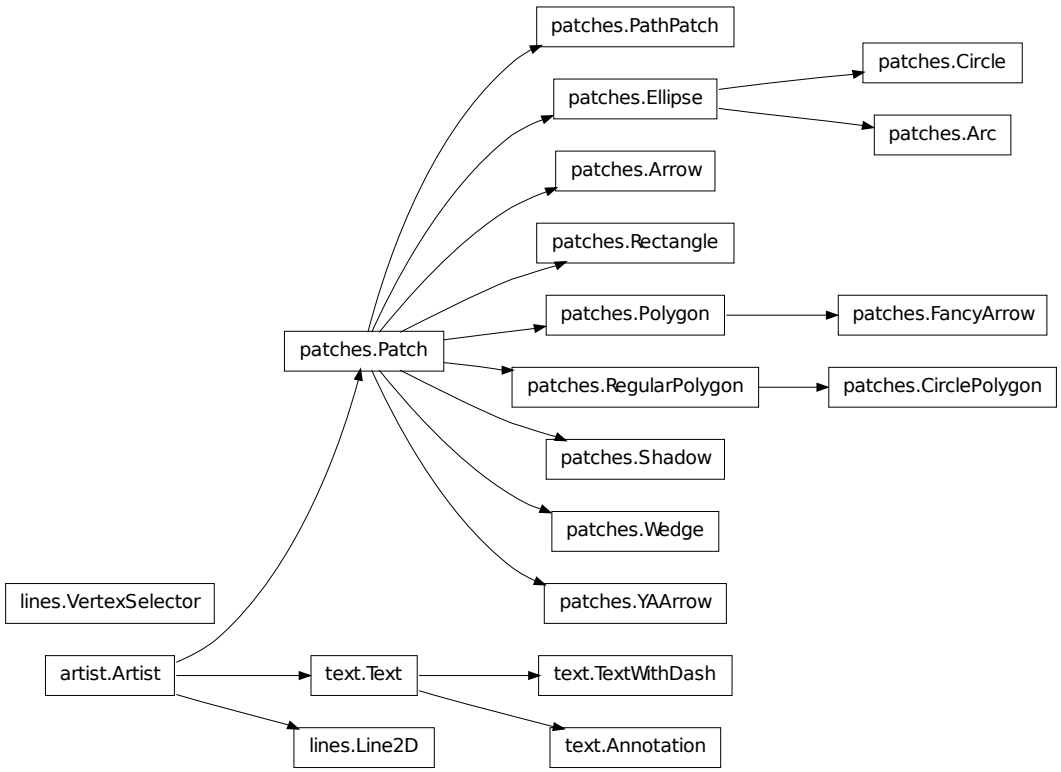
string_width_height(*s*)
Return the string width (including kerning) and string height as a (*w*, *h*) tuple.

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

TWENTYONE

Matplotlib artists



21.1 matplotlib.artist

class **Artist**()

Bases: `object`

Abstract base class for someone who renders into a `FigureCanvas`.

add_callback(*func*)

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 as units support, convert *x* using xaxis unit type

convert_yunits(*y*)

for artists in an axes, if the yaxis as units support, convert *y* using yaxis unit type

draw(*renderer, *args, **kwargs*)

Derived classes drawing method

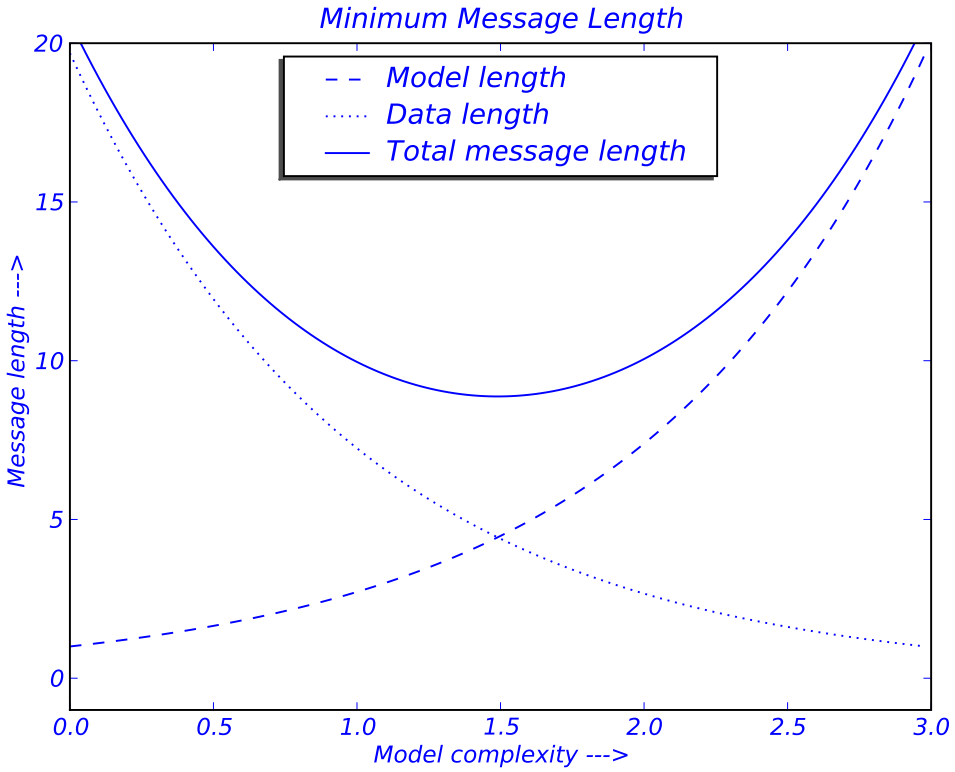
findobj(*match=None*) **pyplot signatur**

findobj(o=gcf(), match=None)

recursively find all :class:matplotlib.artist.Artist instances contained in self

match can be

- None: return all objects contained in artist (including artist)
- function with signature `boolean = match(artist)` used to filter matches
- class instance: eg Line2D. Only return artists of class type



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_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_label()

get_picker()
return the Pickeration instance used by this artist

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_visible()
return the artist's visiblity

get_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

is_figure_set()

is_transform_set()
Artist has transform explicity let

pchanged()
fire event when property changed

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 self is pickable

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*)

set(***kwargs*)
A tkstyle set command, pass *kwargs* to set properties

set_alpha(*alpha*)
Set the alpha value used for blending - not supported on all backends
ACCEPTS: float

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.transform.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: a Path instance and a Transform instance, a Patch instance, or *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.

set_figure(*fig*)
Set the Figure instance the artist belongs to.
ACCEPTS: a matplotlib.figure.Figure instance

set_label(*s*)
Set the line label to *s* for auto legend
ACCEPTS: any string

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_transform(*t*)
Set the `Transform` instance used by this artist.

set_visible(*b*)
set the artist's visiblity

ACCEPTS: [True | False]

set_zorder(*level*)
Set the zorder for the artist

ACCEPTS: any number

update(*props*)

update_from(*other*)
Copy properties from *other* to *self*.

class 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'

findobj(*match=None*)
recursively find all :class: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.

get(*o, *args, **kwargs*)
Return the value of handle property. property is an optional string for the property you want to return

Example usage:

```
getp(o)  # get all the object properties
getp(o, 'linestyle')  # get the linestyle property
```

o is a `Artist` instance, eg `Line2D` or an instance of a `Axes` or `matplotlib.text.Text`. If the *property* is 'somename', this function returns

o.get_somename()

getp can be used to query all the gettable properties with getp(o) Many properties have aliases for shorter typing, eg '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
```

getp(*o, property=None*)

Return the value of handle property. property is an optional string for the property you want to return

Example usage:

```
getp(o)  # get all the object properties
getp(o, 'linestyle')  # get the linestyle property
```

o is a `Artist` instance, eg `Line2D` or an instance of a `Axes` or `matplotlib.text.Text`. If the *property* is 'somename', this function returns

o.get_somename()

getp can be used to query all the gettable properties with getp(o) Many properties have aliases for shorter typing, eg '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
```

kwdoc(*a*)

setp(*h, *args, **kwargs*)

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(TM) 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
```

21.2 matplotlib.lines

This module contains all the 2D line class which can draw with a variety of line styles, markers and colors

class Line2D(*xdata, ydata, linewidth=None, linestyle=None, color=None, marker=None, markersize=None, markeredgewidth=None, markeredgecolor=None, markerfacecolor=None, antialiased=None, dash_capstyle=None, solid_capstyle=None, dash_joinstyle=None, solid_joinstyle=None, pickradius=5, **kwargs*)

Bases: matplotlib.artist.Artist

Create a Line2D instance with *x* and *y* data in sequences *xdata*, *ydata*.

The kwargs are Line2D properties:

Property	Description
alpha	float
animated	[True | False]
antialiased or aa	[True | False]
axes	unknown
clip_box	a `matplotlib.transform.Bbox` instance
clip_on	[True | False]
clip_path	a `Path` instance and a
color or c	any matplotlib color
contains	unknown
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' | 'steps-pre' | 'steps-mid' | 'steps-post' | 'None' | ' ' | '']
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
picker	unknown
pickradius	unknown
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

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()`/`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(*renderer*)

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_linestyle()

get_linewidth()

get_ls()
alias for get_linestyle

get_lw()
alias for get_linewidth

get_marker()

get_markeredgecolor()

get_markeredgewidth()

get_markerfacecolor()

get_markersize()

get_mec()
alias for get_markeredgecolor

get_mew()
alias for get_markeredgewidth

get_mfc()
alias for get_markerfacecolor

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

recache()

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_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: (np.array xdata, np.array ydata)

set_linestyle(*linestyle*)
Set the linestyle of the line

'steps' is equivalent to 'steps-pre' and is maintained for backward-compatibility.

ACCEPTS: ['-' | '–' | '-.' | ':' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post' | 'None' | ' ' | '']

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

ACCEPTS: ['+' | ',' | '.' | '1' | '2' | '3' | '4' '<' | '>' | 'D' | 'H' | '^' | '_' | 'd'
'h' | 'o' | 'p' | 's' | 'v' | 'x' | '|'
TICKUP | TICKDOWN | TICKLEFT | TICKRIGHT
'None' | ' ' | '']

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_markersize(*sz*)
Set the marker size in points

ACCEPTS: float

set_mec(*val*)
alias for set_markeredgecolor

set_mew(*val*)
alias for set_markeredgewidth

set_mfc(*val*)
alias for set_markerfacecolor

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: np.array

set_ydata(*y*)
Set the data np.array for y

ACCEPTS: np.array

update_from(*other*)
copy properties from other to self

class 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.

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.

unmasked_index_ranges(*mask, compressed=True*)

21.3 matplotlib.patches

class 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 cannot 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
alpha	float
animated	[True | False]
antialiased or aa	[True | False]
clip_box	a matplotlib.transform.Bbox instance
clip_on	[True | False]
edgecolor or ec	any matplotlib color
facecolor or fc	any matplotlib color
figure	a matplotlib.figure.Figure instance
fill	[True | False]
hatch	unknown
label	any string
linewidth or lw	float
lod	[True | False]
transform	a matplotlib.transform transformation instance
visible	[True | False]
zorder	any number

draw(*renderer*)

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 **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
alpha	float
animated	[True | False]
antialiased or aa	[True | False]
clip_box	a matplotlib.transform.Bbox instance
clip_on	[True | False]
edgecolor or ec	any matplotlib color
facecolor or fc	any matplotlib color
figure	a matplotlib.figure.Figure instance
fill	[True | False]
hatch	unknown
label	any string
linewidth or lw	float
lod	[True | False]
transform	a matplotlib.transform transformation instance
visible	[True | False]
zorder	any number

get_patch_transform()

get_path()

class `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
alpha	float
animated	[True | False]
antialiased or aa	[True | False]
clip_box	a matplotlib.transform.Bbox instance
clip_on	[True | False]
edgecolor or ec	any matplotlib color
facecolor or fc	any matplotlib color
figure	a matplotlib.figure.Figure instance
fill	[True | False]
hatch	unknown
label	any string
linewidth or lw	float
lod	[True | False]
transform	a matplotlib.transform transformation instance
visible	[True | False]
zorder	any number

class `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
alpha	float
animated	[True | False]
antialiased or aa	[True | False]
clip_box	a matplotlib.transform.Bbox instance
clip_on	[True | False]
edgecolor or ec	any matplotlib color
facecolor or fc	any matplotlib color
figure	a matplotlib.figure.Figure instance
fill	[True | False]
hatch	unknown
label	any string
linewidth or lw	float
lod	[True | False]
transform	a matplotlib.transform transformation instance
visible	[True | False]
zorder	any number

class `Ellipse`(*xy, width, height, angle=0.0, **kwargs*)

Bases: `matplotlib.patches.Patch`

A scale-free ellipse.

xy center of ellipse

width length of horizontal axis

height length of vertical axis

angle rotation in degrees (anti-clockwise)

Valid kwargs are:

Property	Description
alpha	float
animated	[True | False]
antialiased or aa	[True | False]
clip_box	a matplotlib.transform.Bbox instance
clip_on	[True | False]
edgecolor or ec	any matplotlib color
facecolor or fc	any matplotlib color
figure	a matplotlib.figure.Figure instance
fill	[True | False]
hatch	unknown
label	any string
linewidth or lw	float
lod	[True | False]
transform	a matplotlib.transform transformation instance
visible	[True | False]
zorder	any number

`contains`(*ev*)

`get_patch_transform`()

get_path()
Return the vertices of the rectangle

class 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

***length_includes_head*:** *True* if head is counted in calculating the length.

shape: ['full', 'left', 'right']

***overhang*:** distance that the arrow is swept back (0 overhang means triangular shape).

***head_starts_at_zero*:** If *True*, the head starts being drawn at coordinate 0 instead of ending at coordinate 0.

Valid kwargs are:

Property	Description
alpha	float
animated	[True | False]
antialiased or aa	[True | False]
clip_box	a matplotlib.transform.Bbox instance
clip_on	[True | False]
edgecolor or ec	any matplotlib color
facecolor or fc	any matplotlib color
figure	a matplotlib.figure.Figure instance
fill	[True | False]
hatch	unknown
label	any string
linewidth or lw	float
lod	[True | False]
transform	a matplotlib.transform transformation instance
visible	[True | False]
zorder	any number

class Patch(*edgecolor=None, facecolor=None, linewidth=None, linestyle=None, antialiased=None, hatch=None, fill=True, **kwargs*)
Bases: matplotlib.artist.Artist

A patch is a 2D thingy 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
alpha	float
animated	[True | False]
antialiased or aa	[True | False]
clip_box	a matplotlib.transform.Bbox instance
clip_on	[True | False]
edgecolor or ec	any matplotlib color
facecolor or fc	any matplotlib color
figure	a matplotlib.figure.Figure instance
fill	[True | False]
hatch	unknown
label	any string
linewidth or lw	float
lod	[True | False]
transform	a matplotlib.transform transformation instance
visible	[True | False]
zorder	any number

contains(*mouseevent*)
Test whether the mouse event occurred in the patch.
Returns T/F, {}

draw(*renderer*)

get_aa()

get_antialiased()

get_data_transform()

get_ec()

get_edgecolor()

get_extents()

get_facecolor()

get_fc()

get_fill()
return whether fill is set

get_hatch()
return the current hatching pattern

get_linestyle()

get_linewidth()

get_ls()

get_lw()

get_patch_transform()

get_path()
Return the path of this patch

get_transform()

get_verts()
Return a copy of the vertices used in this patch
If the patch contains Bézier 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*)
Set whether to use antialiased rendering
ACCEPTS: [True | False] or None for default

set_antialiased(*aa*)
Set whether to use antialiased rendering
ACCEPTS: [True | False] or None for default

set_ec(*color*)
Set the patch edge color
ACCEPTS: mpl color spec, or None for default, or 'none' for no color

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*)
Set the patch face color
ACCEPTS: mpl color spec, or None for default, or 'none' for no color

set_fill(*b*)
Set whether to fill the patch
ACCEPTS: [True | False]

set_hatch(*h*)
Set the hatching pattern
hatch can be one of:

```
/   - diagonal hatching
\   - back diagonal
|   - vertical
-   - horizontal
#   - crossed
x   - crossed diagonal
```

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching in that direction.
CURRENT LIMITATIONS:

1. Hatching is supported in the PostScript backend only.
2. Hatching is done with solid black lines of width 0.

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*)
Set the patch linestyle
ACCEPTS: ['solid' | 'dashed' | 'dashdot' | 'dotted']

set_lw(*w*)
Set the patch linewidth in points
ACCEPTS: float or None for default

update_from(*other*)

class PathPatch(*path, **kwargs*)
Bases: matplotlib.patches.Patch

A general polycurve path patch.

path is a matplotlib.path.Path object.

Valid kwargs are:

Property	Description
alpha	float
animated	[True | False]
antialiased or aa	[True | False]
clip_box	a matplotlib.transform.Bbox instance
clip_on	[True | False]
edgecolor or ec	any matplotlib color
facecolor or fc	any matplotlib color
figure	a matplotlib.figure.Figure instance
fill	[True | False]
hatch	unknown
label	any string
linewidth or lw	float
lod	[True | False]
transform	a matplotlib.transform transformation instance
visible	[True | False]
zorder	any number

See Patch documentation for additional kwargs

get_path()

class 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
alpha	float
animated	[True | False]
antialiased or aa	[True | False]
clip_box	a matplotlib.transform.Bbox instance
clip_on	[True | False]
edgecolor or ec	any matplotlib color
facecolor or fc	any matplotlib color
figure	a matplotlib.figure.Figure instance
fill	[True | False]
hatch	unknown
label	any string
linewidth or lw	float
lod	[True | False]
transform	a matplotlib.transform transformation instance
visible	[True | False]
zorder	any number

See Patch documentation for additional kwargs

get_closed()

get_path()

get_xy()

set_closed(*closed*)

set_xy(*vertices*)

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 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
alpha	float
animated	[True | False]
antialiased or aa	[True | False]
clip_box	a matplotlib.transform.Bbox instance
clip_on	[True | False]
edgecolor or ec	any matplotlib color
facecolor or fc	any matplotlib color
figure	a matplotlib.figure.Figure instance
fill	[True | False]
hatch	unknown
label	any string
linewidth or lw	float
lod	[True | False]
transform	a matplotlib.transform transformation instance
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_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_y(*y*)

Set the bottom coord of the rectangle

ACCEPTS: float

class 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
alpha	float
animated	[True | False]
antialiased or aa	[True | False]
clip_box	a matplotlib.transform.Bbox instance
clip_on	[True | False]
edgecolor or ec	any matplotlib color
facecolor or fc	any matplotlib color
figure	a matplotlib.figure.Figure instance
fill	[True | False]
hatch	unknown
label	any string
linewidth or lw	float
lod	[True | False]
transform	a matplotlib.transform transformation instance
visible	[True | False]
zorder	any number

get_patch_transform()

get_path()

numvertices

orientation

radius

xy

class 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
alpha	float
animated	[True | False]
antialiased or aa	[True | False]
clip_box	a matplotlib.transform.Bbox instance
clip_on	[True | False]
edgecolor or ec	any matplotlib color
facecolor or fc	any matplotlib color
figure	a matplotlib.figure.Figure instance
fill	[True | False]
hatch	unknown
label	any string
linewidth or lw	float
lod	[True | False]
transform	a matplotlib.transform transformation instance
visible	[True | False]
zorder	any number

`get_patch_transform`()

`get_path`()

class `Wedge`(*center, r, theta1, theta2, **kwargs*)

Bases: `matplotlib.patches.Patch`

Draw a wedge centered at *x*, *y* center with radius *r* that sweeps *theta1* to *theta2* (in degrees).

Valid kwargs are:

Property	Description
alpha	float
animated	[True | False]
antialiased or aa	[True | False]
clip_box	a matplotlib.transform.Bbox instance
clip_on	[True | False]
edgecolor or ec	any matplotlib color
facecolor or fc	any matplotlib color
figure	a matplotlib.figure.Figure instance
fill	[True | False]
hatch	unknown
label	any string
linewidth or lw	float
lod	[True | False]
transform	a matplotlib.transform transformation instance
visible	[True | False]
zorder	any number

`get_patch_transform`()

`get_path`()

class `YAArrow`(*figure, xytip, xybase, width=4, frac=0.10000000000000001, 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
alpha	float
animated	[True | False]
antialiased or aa	[True | False]
clip_box	a matplotlib.transform.Bbox instance
clip_on	[True | False]
edgecolor or ec	any matplotlib color
facecolor or fc	any matplotlib color
figure	a matplotlib.figure.Figure instance
fill	[True | False]
hatch	unknown
label	any string
linewidth or lw	float
lod	[True | False]
transform	a matplotlib.transform transformation instance
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*.

`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.

`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.

21.4 matplotlib.text

Figure and Axes text

class **Annotation**(*s, xy, xytext=None, xycoords='data', textcoords=None, arrowprops=None, **kwargs*)

Bases: matplotlib.text.Text

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. Valid keys 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`

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'
```

Additional kwargs are Text properties:

Property	Description
alpha	float
animated	[True | False]
axes	an axes instance
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
color	any matplotlib color
contains	unknown
family	['serif' | 'sans-serif' | 'cursive' | 'fantasy' | 'monospace']
figure	a `matplotlib.figure.Figure` instance
fontproperties	a matplotlib.font_manager.FontProperties instance
horizontalalignment or ha	['center' | 'right' | 'left']
label	any string
linespacing	float
lod	[True | False]
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	unknown
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

contains(*event*)

draw(*renderer*)

set_clip_box(*clipbox*)

Set the artist's clip Bbox

ACCEPTS: a matplotlib.transform.Bbox instance

set_figure(*fig*)

update_positions(*renderer*)

class Text(*x=0, y=0, text='', color=None, verticalalignment='bottom', horizontalalignment='left', multialignment=None, fontproperties=None, rotation=None, linespacing=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

alpha	float
animated	[True | False]
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]
color	any matplotlib color
family	['serif' | 'sans-serif' | 'cursive' | 'fantasy' | 'monospace']
figure	a matplotlib.figure.Figure instance
fontproperties	a matplotlib.font_manager.FontProperties instance
horizontalalignment or ha	['center' | 'right' | 'left']
label	any string
linespacing	float
lod	[True | False]
multialignment	['left' | 'right' | 'center']
name or fontname	string eg, ['Sans' | 'Courier' | 'Helvetica' ...]
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
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

contains(*mouseevent*)

Test whether the mouse event occurred in the patch.

Returns T/F, {}

draw(*renderer*)

get_color()

Return the color of the text

get_font_properties()

Return the font object

get_fontname()

alias for get_name

get_fontsize()
alias for get_size

get_fontstyle()
alias for get_style

get_fontweight()
alias for get_weight

get_ha()
alias for get_horizontalalignment

get_horizontalalignment()
Return the horizontal alignment as string

get_name()
Return the font name as string

get_position()
Return x, y as tuple

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

get_size()
Return the font size as integer

get_style()
Return the font style as string

get_text()
Get the text as string

get_va()
alias for getverticalalignment

get_verticalalignment()
Return the vertical alignment as string

get_weight()
Get the font weight as string

get_window_extent(*renderer=None*)

is_math_text(*s*)

set_backgroundcolor(*color*)
Set the background color of the text by updating the bbox (see set_bbox for more info)

ACCEPTS: any matplotlib color

set_bbox(*rectprops*)
Draw a bounding box around self. rect props are any settable properties for a rectangle, eg facecolor='red', alpha=0.5.

t.set_bbox(dict(facecolor='red', alpha=0.5))

ACCEPTS: rectangle prop dict plus key 'pad' which is a pad in points

set_color(*color*)
Set the foreground color of the text
ACCEPTS: any matplotlib color

set_family(*fontname*)
Set the font family
ACCEPTS: ['serif' | 'sans-serif' | 'cursive' | 'fantasy' | 'monospace']

set_fontname(*fontname*)
alias for set_name

set_fontproperties(*fp*)
Set the font properties that control the text
ACCEPTS: a matplotlib.font_manager.FontProperties instance

set_fontsize(*fontsize*)
alias for set_size

set_fontstyle(*fontstyle*)
alias for set_style

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

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*)
Set the font name,
ACCEPTS: string eg, ['Sans' | 'Courier' | 'Helvetica' ...]

set_position(*xy*)
Set the xy position of the text
ACCEPTS: (x,y)

set_rotation(*s*)
Set the rotation of the text
ACCEPTS: [angle in degrees 'vertical' | 'horizontal'

set_size(*fontsize*)
Set the font size, eg, 8, 10, 12, 14...
ACCEPTS: [size in points | relative size eg 'smaller', 'x-large']

set_style(*fontstyle*)
Set the font style
ACCEPTS: ['normal' | 'italic' | 'oblique']

set_text(*s*)
Set the text string s
ACCEPTS: string or anything printable with '%s' conversion

set_va(*align*)
alias for set_verticalalignment

set_variant(*variant*)
Set the font variant, eg,
ACCEPTS: ['normal' | 'small-caps']

set_verticalalignment(*align*)
Set the vertical alignment
ACCEPTS: ['center' | 'top' | 'bottom' | 'baseline']

set_weight(*weight*)
Set the font weight
ACCEPTS: ['normal' | 'bold' | 'heavy' | 'light' | 'ultrabold' | 'ultralight']

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_from(*other*)
Copy properties from other to self

class 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, xaxis=True*)
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 2: 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*)

get_dashdirection()

get_dashlength()

get_dashpad()

get_dashpush()

get_dashrotation()

get_figure()
return the figure instance

get_position()
Return x, y as tuple

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*)

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

set_dashlength(*dl*)
Set the length of the dash.

ACCEPTS: float

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

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

set_dashrotation(*dr*)
Set the rotation of the dash.
ACCEPTS: float

set_figure(*fig*)
Set the figure instance the artist belong to.
ACCEPTS: a matplotlib.figure.Figure instance

set_position(*xy*)
Set the xy position of the TextWithDash.
ACCEPTS: (x,y)

set_transform(*t*)
Set the Transformation instance used by this artist.
ACCEPTS: a matplotlib.transform transformation 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.

get_rotation(*rotation*)
return the text angle as float

CHAPTER

TWENTYTWO

Matplotlib figure

22.1 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 Figure(*figsize=None, dpi=None, facecolor=None, edgecolor=None, linewidth=1.0, frameon=True, subplotpars=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.

The figure patch is drawn by a the attribute

patch 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

add_axes(**args, **kwargs*)

Add an a axes with axes 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,

polar, rectilinear. Some of these projections support additional kwargs, which may be provided to add_axes():

```
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)   # add an Axes instance
```

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, eg. 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')
```

The Axes instance will be returned.

The following kwargs are supported:

Property	Description
adjustable	['box' | 'datalim']
alpha	float
anchor	unknown
animated	[True | False]
aspect	unknown
autoscale_on	unknown
axes	an axes instance
axis_bgcolor	any matplotlib color - see
axis_off	unknown
axis_on	unknown
axisbelow	[*True* | *False*]
clip_box	a `matplotlib.transform.Bbox` instance
clip_on	[True | False]
clip_path	a `Path` instance and a
color_cycle	unknown
contains	unknown
cursor_props	a (*float*, *color*) tuple
figure	unknown
frame_on	[*True* | *False*]
label	any string
lod	[True | False]
navigate	[True | False]
navigate_mode	unknown
picker	[None|float|boolean|callable]
position	unknown
title	unknown
transform	unknown
visible	[True | False]
xbound	unknown
xlabel	unknown
xlim	len(2) sequence of floats
xscale	['linear' | 'log' | 'symlog']
xticklabels	unknown
xticks	sequence of floats
ybound	unknown
ylabel	unknown
ylim	len(2) sequence of floats
yscale	['linear' | 'log' | 'symlog']
yticklabels	unknown
yticks	sequence of floats
zorder	any number

add_axobserver(*func*)

whenever the axes state change, func(self) will be called

add_subplot(**args, **kwargs*)

autofmt_xdate(*bottom=0.20000000000000001, 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

clear()

Clear the figure

clf()

Clear the figure

colorbar(*mappable, cax=None, ax=None, **kw*)

Create a colorbar for a ScalarMappable instance.

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 `matplotlib.Figure.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 `matplotlib.Figure.colorbar()` method but optional for the `matplotlib.pyplot.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

Additional keyword arguments are of two kinds:

axes properties:

Property	Description
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

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 metho
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 forma 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 eac 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.

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(*renderer*)

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=1.0, norm=None, cmap=None, vmin=None, vmax=None, origin=None*)

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:

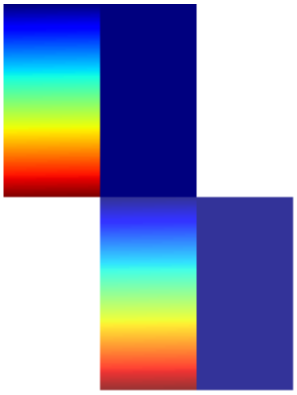
- If *X* is MxN, assume luminance (grayscale)
- If *X* is MxNx3, assume RGB
- If *X* is MxNx4, assume RGBA

Optional keyword arguments:

Keyword	Description
xo or yo	An integer, the *x* and *y* image offset in pixels
cmap	a `matplotlib.cm.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 1.0
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.



gca(***kwargs*)

Return the current axes, creating one if necessary

The following kwargs are supported

Property	Description
adjustable	['box' | 'datalim']
alpha	float
anchor	unknown
animated	[True | False]
aspect	unknown
autoscale_on	unknown
axes	an axes instance
axis_bgcolor	any matplotlib color - see
axis_off	unknown
axis_on	unknown
axisbelow	[*True* | *False*]
clip_box	a `matplotlib.transform.Bbox` instance
clip_on	[True | False]
clip_path	a `Path` instance and a
color_cycle	unknown
contains	unknown
cursor_props	a (*float*, *color*) tuple
figure	unknown
frame_on	[*True* | *False*]
label	any string
lod	[True | False]
navigate	[True | False]
navigate_mode	unknown
picker	[None|float|boolean|callable]
position	unknown
title	unknown
transform	unknown
visible	[True | False]
xbound	unknown
xlabel	unknown
xlim	len(2) sequence of floats
xscale	['linear' | 'log' | 'symlog']
xticklabels	unknown
xticks	sequence of floats
ybound	unknown
ylabel	unknown
ylim	len(2) sequence of floats
yscale	['linear' | 'log' | 'symlog']
yticklabels	unknown
yticks	sequence of floats
zorder	any number

get_axes()

get_children()
get a list of artists contained in the figure

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_window_extent(**args, **kwargs*)
get the figure bounding box in display space; kwargs are void

ginput(*n=1, timeout=30, show_clicks=True*)
call signature:

```
ginput(self, n=1, timeout=30, show_clicks=True)
```

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 negative, does not timeout.

If *n* is negative, accumulate clicks until a middle click terminates the input.

Right clicking cancels last input.

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.

The legend instance is returned. The following kwargs are supported

loc the location of the legend

numpoints the number of points in the legend line

prop a `matplotlib.font_manager.FontProperties` instance

pad the fractional whitespace inside the legend border

markerscale the relative size of legend markers vs. original

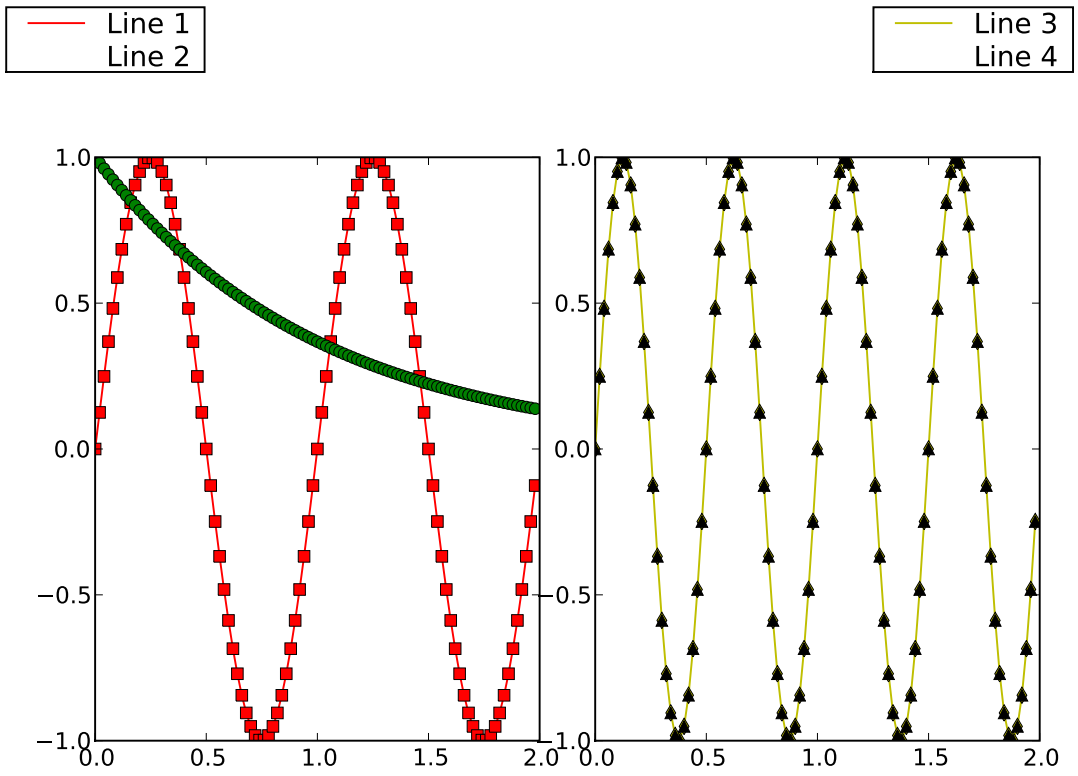
shadow if True, draw a shadow behind legend

labelsep the vertical space between the legend entries

handlelen the length of the legend lines

handletextsep the space between the legend line and legend text

axespad the border between the axes and legend edge



savefig(**args, **kwargs*)

call signature:

```
savefig(fname, dpi=None, facecolor='w', edgecolor='w',
        orientation='portrait', papertype=None, format=None,
        transparent=False):
```

Save the current figure.

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.

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 figure patch and axes patches will all be transparent. 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.

`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_figsize_inches`(**args, **kwargs*)

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(**args, **kwargs*)
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

WARNING: forward=True is broken on all backends except GTK* and WX*

ACCEPTS: a w,h tuple with w,h in inches

subplots_adjust(**args, **kwargs*)
fig.subplots_adjust(left=None, bottom=None, right=None, wspace=None, hspace=None)

Update the SubplotParams with *kwargs* (defaulting to rc where 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 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.subtitle('this is the figure title', fontsize=12)
```

text(*x, y, s, *args, **kwargs*)
Call signature:

```
figtext(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
alpha	float
animated	[True | False]
axes	an axes instance
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
color	any matplotlib color
contains	unknown
family	['serif' | 'sans-serif' | 'cursive' | 'fantasy' | 'monospace']
figure	a `matplotlib.figure.Figure` instance
fontproperties	a matplotlib.font_manager.FontProperties instance
horizontalalignment or ha	['center' | 'right' | 'left']
label	any string
linespacing	float
lod	[True | False]
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	unknown
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

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 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

validate make sure the params are in a legal state (*left*<**right*, etc)

`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

`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

TWENTYTHREE

Matplotlib axes

23.1 matplotlib.axes

class Axes(*fig, rect, axisbg=None, frameon=True, sharex=None, sharey=None, label='', **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*)

call signature:

```
acorr(x, normed=False, detrend=mlab.detrend_none, usevlines=False,
    maxlags=None, **kwargs)
```

Plot the autocorrelation of *x*. If *normed* = *True*, normalize the data but 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. The return value is a tuple (*lags*, *c*, *linecol*, *b*) where

- *linecol* is the LineCollection
- *b* is the *x*-axis.

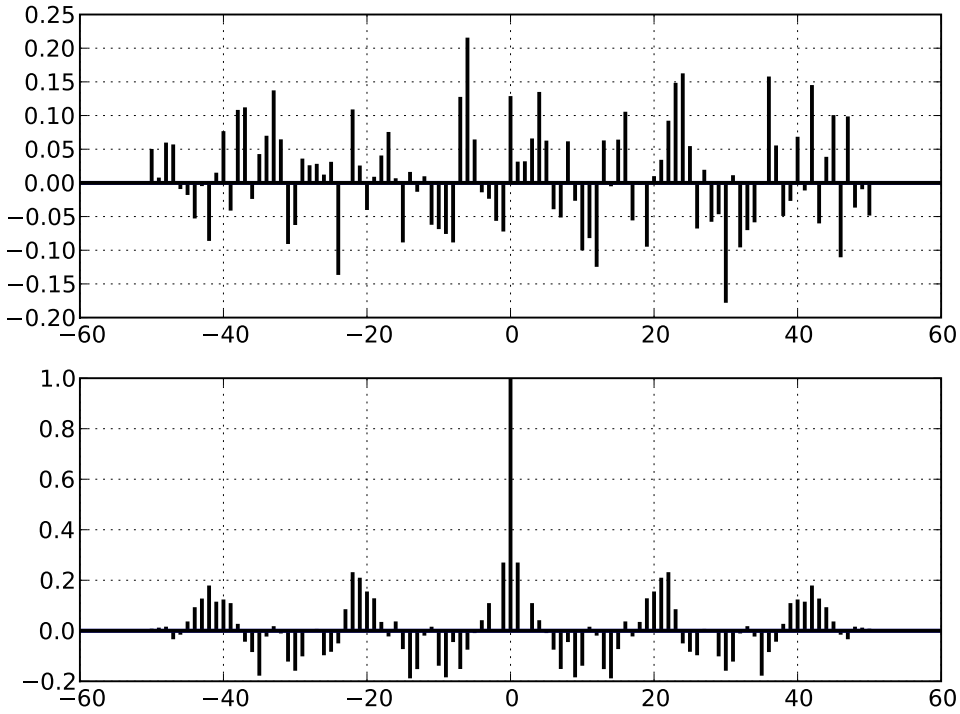
maxlags is a positive integer detailing the number of lags to show. The default value of *None* will return all `(2*len(x)-1)` lags.

See the respective plot() or vlines() functions for documentation on valid kwargs.

Example:

xcorr() above, and acorr() below.

Example:



add_artist(*a*)

Add any Artist to the axes

add_collection(*collection, autolim=True*)

add a Collection instance to the axes

add_line(*line*)

Add a Line2D to the list of plot lines

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.

add_table(*tab*)

Add a Table instance to the list of axes tables

annotate(**args, **kwargs*)

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. Valid keys 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`

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'
```

Additional kwargs are Text properties:

Property	Description
alpha	float
animated	[True | False]
axes	an axes instance
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
color	any matplotlib color
contains	unknown
family	['serif' | 'sans-serif' | 'cursive' | 'fantasy' | 'monospace']
figure	a `matplotlib.figure.Figure` instance
fontproperties	a matplotlib.font_manager.FontProperties instance
horizontalalignment or ha	['center' | 'right' | 'left']
label	any string
linespacing	float
lod	[True | False]
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	unknown
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

apply_aspect(*position=None*)

Use `_aspect()` and `_adjustable()` to modify the axes box or the view limits.

arrow(*x, y, dx, dy, **kwargs*)

call signature:

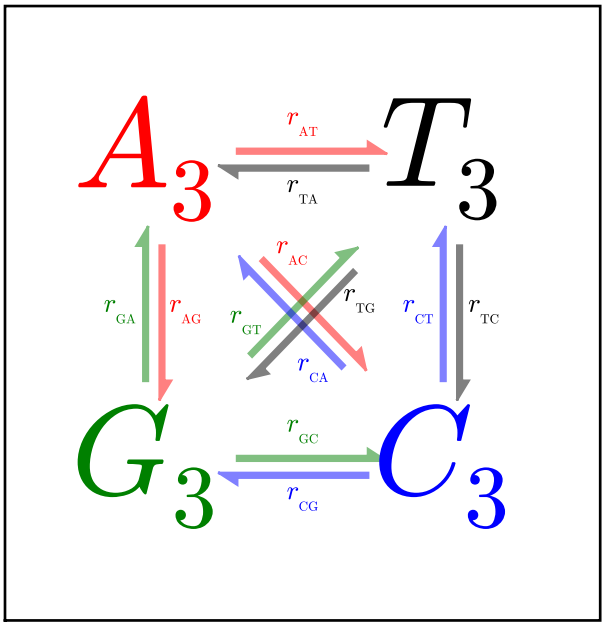
```
arrow(x, y, dx, dy, **kwargs)
```

Draws arrow on specified axis from (*x*, *y*) to (*x* + *dx*, *y* + *dy*).

Optional kwargs control the arrow properties:

Property	Description
aa	[True | False] or None for default
alpha	float
animated	[True | False]
antialiased	[True | False] or None for default
axes	an axes instance
clip_box	a `matplotlib.transform.Bbox` instance
clip_on	[True | False]
clip_path	a `Path` instance and a
contains	unknown
ec	mpl color spec, or None for default, or 'none' for no color
edgecolor	mpl color spec, or None for default, or 'none' for no color
facecolor	mpl color spec, or None for default, or 'none' for no color
fc	mpl color spec, or None for default, or 'none' for no color
figure	a `matplotlib.figure.Figure` instance
fill	[True | False]
hatch	unknown
label	any string
linestyle	['solid' | 'dashed' | 'dashdot' | 'dotted']
linewidth	float or None for default
lod	[True | False]
ls	['solid' | 'dashed' | 'dashdot' | 'dotted']
lw	float or None for default
picker	[None|float|boolean|callable]
transform	unknown
visible	[True | False]
zorder	any number

Example:



autoscale_view(*tight=False, 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.

axhline(*y=0, xmin=0, xmax=1, **kwargs*)

call signature:

```
axhline(y=0, xmin=0, xmax=1, **kwargs)
```

Axis Horizontal Line

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:

Property	Description
alpha	float
animated	[True | False]
antialiased or aa	[True | False]
axes	unknown
clip_box	a `matplotlib.transform.Bbox` instance
clip_on	[True | False]
clip_path	a `Path` instance and a
color or c	any matplotlib color
contains	unknown
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' | 'steps-pre' | 'steps-mid' | 'steps-post' | 'None' | ' ' | '']
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
picker	unknown
pickradius	unknown
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

See `axhspan()` for example plot and source code

axhspan(*ymin, ymax, xmin=0, xmax=1, **kwargs*)

call signature:

```
axhspan(ymin, ymax, xmin=0, xmax=1, **kwargs)
```

Axis Horizontal Span.

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 span 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:

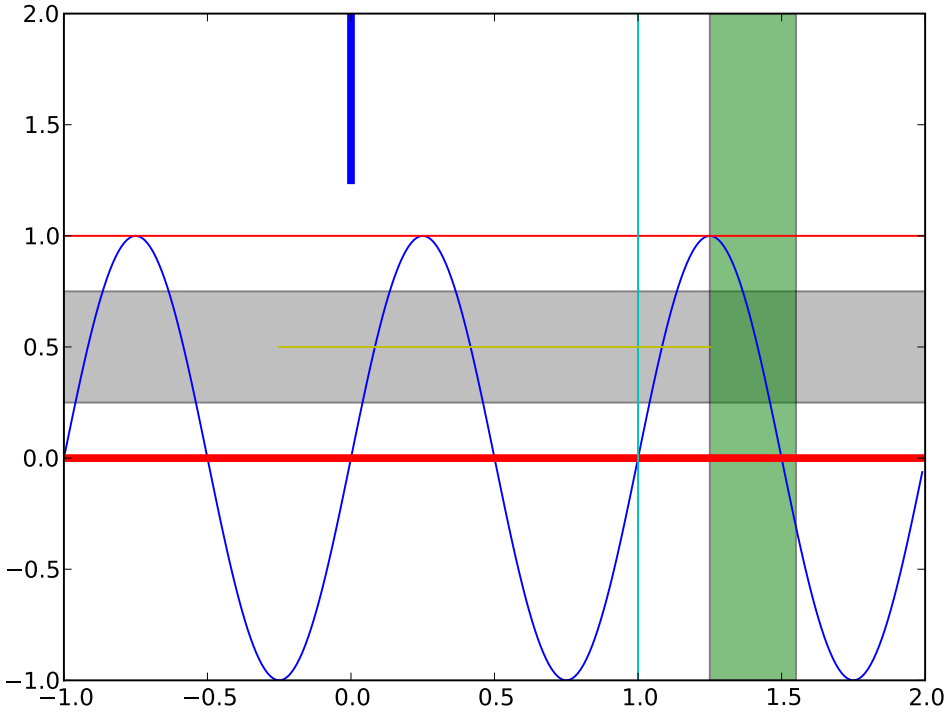
- draw a gray rectangle from y = 0.25-0.75 that spans the horizontal extent of the axes

```
>>> axhspan(0.25, 0.75, facecolor='0.5', alpha=0.5)
```

Valid kwargs are Polygon properties:

Property	Description
aa	[True | False] or None for default
alpha	float
animated	[True | False]
antialiased	[True | False] or None for default
axes	an axes instance
clip_box	a matplotlib.transform.Bbox instance
clip_on	[True | False]
clip_path	a Path instance and a
contains	unknown
ec	mpl color spec, or None for default, or 'none' for no color
edgecolor	mpl color spec, or None for default, or 'none' for no color
facecolor	mpl color spec, or None for default, or 'none' for no color
fc	mpl color spec, or None for default, or 'none' for no color
figure	a matplotlib.figure.Figure instance
fill	[True | False]
hatch	unknown
label	any string
linestyle	['solid' | 'dashed' | 'dashdot' | 'dotted']
linewidth	float or None for default
lod	[True | False]
ls	['solid' | 'dashed' | 'dashdot' | 'dotted']
lw	float or None for default
picker	[None|float|boolean|callable]
transform	unknown
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. *kwargs* are passed on to `set_xlim()` and `set_ylim()`

axvline(*x=0, ymin=0, ymax=1, **kwargs*)

call signature:

```
axvline(x=0, ymin=0, ymax=1, **kwargs)
```

Axis Vertical Line

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 xlim settings, even if you change them, eg. with the `set_xlim()` 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:

Property	Description
alpha	float
animated	[True | False]
antialiased or aa	[True | False]
axes	unknown
clip_box	a matplotlib.transform.Bbox instance
clip_on	[True | False]
clip_path	a Path instance and a
color or c	any matplotlib color
contains	unknown
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' | 'steps-pre' | 'steps-mid' | 'steps-post' | 'None' | ' ' | '']
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
picker	unknown
pickradius	unknown
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

See axhspan() for example plot and source code

axvspan(*xmin, xmax, ymin=0, ymax=1, **kwargs*)

call signature:

```
axvspan(xmin, xmax, ymin=0, ymax=1, **kwargs)
```

Axis Vertical Span.

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 span 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
aa	[True | False] or None for default
alpha	float
animated	[True | False]
antialiased	[True | False] or None for default
axes	an axes instance
clip_box	a `matplotlib.transform.Bbox` instance
clip_on	[True | False]
clip_path	a `Path` instance and a
contains	unknown
ec	mpl color spec, or None for default, or 'none' for no color
edgecolor	mpl color spec, or None for default, or 'none' for no color
facecolor	mpl color spec, or None for default, or 'none' for no color
fc	mpl color spec, or None for default, or 'none' for no color
figure	a `matplotlib.figure.Figure` instance
fill	[True | False]
hatch	unknown
label	any string
linestyle	['solid' | 'dashed' | 'dashdot' | 'dotted']
linewidth	float or None for default
lod	[True | False]
ls	['solid' | 'dashed' | 'dashdot' | 'dotted']
lw	float or None for default
picker	[None|float|boolean|callable]
transform	unknown
visible	[True | False]
zorder	any number

See `axhspan()` for example plot and source code

bar(*left, height, width=0.80000000000000004, bottom=None, color=None, edgecolor=None, linewidth=None, yerr=None, xerr=None, ecolor=None, capsize=3, align='edge', orientation='vertical', log=False, **kwargs*)
call signature:

```
bar(left, height, width=0.8, bottom=0,
    color=None, edgecolor=None, linewidth=None,
```

```
yerr=None, xerr=None, ecolor=None, capsize=3,
align='edge', orientation='vertical', log=False)
```

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
align	'edge' (default) | 'center'
orientation	'vertical' | 'horizontal'
log	[False|True] False (default) leaves the orientation axis as-is; True sets it to log scale

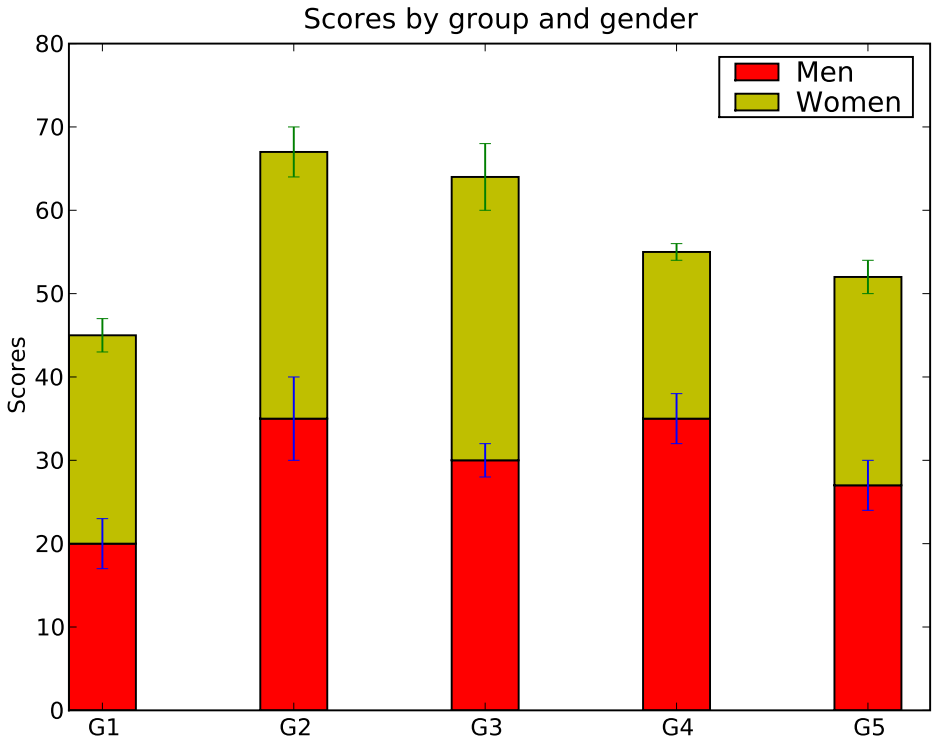
For vertical bars, *align* = 'edge' aligns bars by their left edges in left, while *align* = 'center' interprets these values as the *x* coordinates of the bar centers. For horizontal bars, *align* = 'edge' aligns bars by their bottom edges in bottom, while *align* = 'center' interprets these values 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.

Other optional kwargs:

Property	Description
aa	[True | False] or None for default
alpha	float
animated	[True | False]
antialiased	[True | False] or None for default
axes	an axes instance
clip_box	a `matplotlib.transform.Bbox` instance
clip_on	[True | False]
clip_path	a `Path` instance and a
contains	unknown
ec	mpl color spec, or None for default, or 'none' for no color
edgecolor	mpl color spec, or None for default, or 'none' for no color
facecolor	mpl color spec, or None for default, or 'none' for no color
fc	mpl color spec, or None for default, or 'none' for no color
figure	a `matplotlib.figure.Figure` instance
fill	[True | False]
hatch	unknown
label	any string
linestyle	['solid' | 'dashed' | 'dashdot' | 'dotted']
linewidth	float or None for default
lod	[True | False]
ls	['solid' | 'dashed' | 'dashdot' | 'dotted']
lw	float or None for default
picker	[None|float|boolean|callable]
transform	unknown
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:

Unexpected indentation.

'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.

Unexpected indentation.

'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
alpha	float
animated	[True | False]
antialiased	Boolean or sequence of booleans
antialiaseds	Boolean or sequence of booleans
array	unknown
axes	an axes instance
clim	a length 2 sequence of floats
clip_box	a matplotlib.transform.Bbox instance
clip_on	[True | False]
clip_path	a Path instance and a
cmap	a colormap
color	matplotlib color arg or sequence of rgba tuples
colorbar	unknown
contains	unknown
dashes	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
edgecolor	matplotlib color arg or sequence of rgba tuples
edgecolors	matplotlib color arg or sequence of rgba tuples
facecolor	matplotlib color arg or sequence of rgba tuples
facecolors	matplotlib color arg or sequence of rgba tuples
figure	a matplotlib.figure.Figure instance
label	any string
linestyle	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
linestyles	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
linewidth	float or sequence of floats
linewidths	float or sequence of floats
lod	[True | False]
lw	float or sequence of floats
norm	unknown
offsets	float or sequence of floats
picker	[None|float|boolean|callable]
pickradius	unknown
transform	unknown
visible	[True | False]
zorder	any number

barh(*bottom, width, height=0.80000000000000004, left=None, **kwargs*)

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
aa	[True | False] or None for default
alpha	float
animated	[True | False]
antialiased	[True | False] or None for default
axes	an axes instance
clip_box	a `matplotlib.transform.Bbox` instance
clip_on	[True | False]
clip_path	a `Path` instance and a
contains	unknown
ec	mpl color spec, or None for default, or 'none' for no color
edgecolor	mpl color spec, or None for default, or 'none' for no color
facecolor	mpl color spec, or None for default, or 'none' for no color
fc	mpl color spec, or None for default, or 'none' for no color
figure	a `matplotlib.figure.Figure` instance
fill	[True | False]
hatch	unknown
label	any string
linestyle	['solid' | 'dashed' | 'dashdot' | 'dotted']
linewidth	float or None for default
lod	[True | False]
ls	['solid' | 'dashed' | 'dashdot' | 'dotted']
lw	float or None for default
picker	[None|float|boolean|callable]
transform	unknown
visible	[True | False]
zorder	any number

boxplot(*x, notch=0, sym='b+', vert=1, whis=1.5, positions=None, widths=None*)

call signature:

```
boxplot(x, notch=0, sym='+', vert=1, whis=1.5,
        positions=None, widths=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.

- *notch* = 0 (default) produces a rectangular box plot.
- *notch* = 1 will produce a notched box plot

sym (default 'b+') is the default symbol for flier points. Enter an empty string ('') if you don't want to show fliers.

- *vert* = 1 (default) makes the boxes vertical.
- *vert* = 0 makes horizontal boxes. This seems goofy, but that's how Matlab did it.

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.

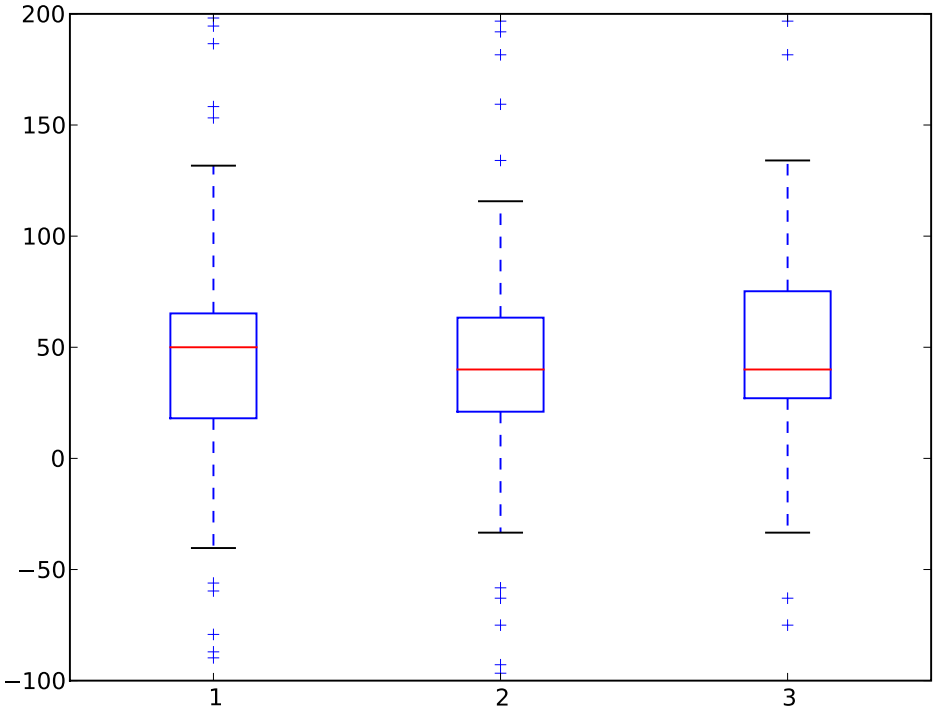
positions (default 1,2,...,n) sets the horizontal positions of the boxes. The ticks and limits are automatically set to match the positions.

widths is 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.

x is an array or a sequence of vectors.

Returns a list of the `matplotlib.lines.Line2D` instances added.

Example:



broken_barh(*xranges, yrange, **kwargs*)

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
alpha	float
animated	[True | False]
antialiased	Boolean or sequence of booleans
antialiaseds	Boolean or sequence of booleans
array	unknown
axes	an axes instance
clim	a length 2 sequence of floats
clip_box	a `matplotlib.transform.Bbox` instance
clip_on	[True | False]
clip_path	a `Path` instance and a
cmap	a colormap
color	matplotlib color arg or sequence of rgba tuples
colorbar	unknown
contains	unknown
dashes	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
edgecolor	matplotlib color arg or sequence of rgba tuples
edgecolors	matplotlib color arg or sequence of rgba tuples
facecolor	matplotlib color arg or sequence of rgba tuples
facecolors	matplotlib color arg or sequence of rgba tuples
figure	a `matplotlib.figure.Figure` instance
label	any string
linestyle	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
linestyles	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
linewidth	float or sequence of floats
linewidths	float or sequence of floats
lod	[True | False]
lw	float or sequence of floats
norm	unknown
offsets	float or sequence of floats
picker	[None|float|boolean|callable]
pickradius	unknown
transform	unknown
visible	[True | False]
zorder	any number

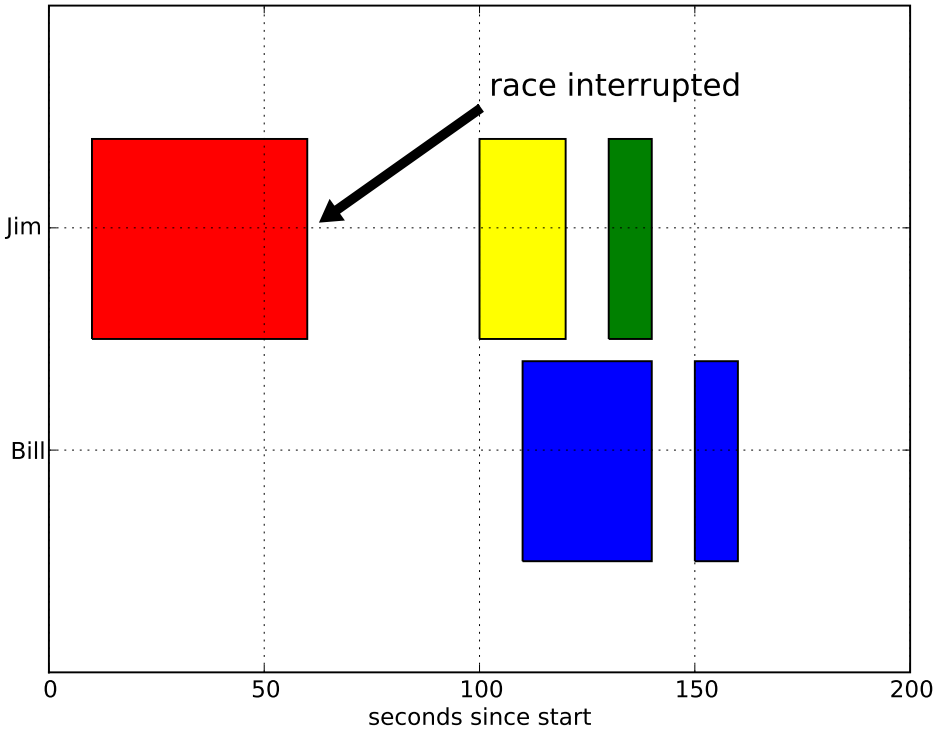
these can either be a single argument, ie:

```
facecolors = 'black'
```

or a sequence of arguments for the various bars, ie:

```
facecolors = ('black', 'red', 'green')
```

Example:



can_zoom()
Return *True* if this axes support the zoom box

cla()
Clear the current axes

clabel(*CS, *args, **kwargs*)
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*:** See http://matplotlib.sf.net/fonts.html

clear()
clear the axes

cohere(*x, y, NFFT=256, Fs=2, Fc=0, detrend=<function detrend_none at 0x8d3d7d4>, window=<function window_hanning at 0x8d3d4c4>, noverlap=0, **kwargs*)
call signature:

```
cohere(x, y, NFFT=256, Fs=2, Fc=0, detrend = mlab.detrend_none,
       window = mlab.window_hanning, noverlap=0, **kwargs)
```

cohere the coherence between x and y. Coherence is the normalized cross spectral density:

$$C_{xy} = \frac{|P_{xy}|^2}{P_{xx} * P_{yy}} \tag{23.1}$$

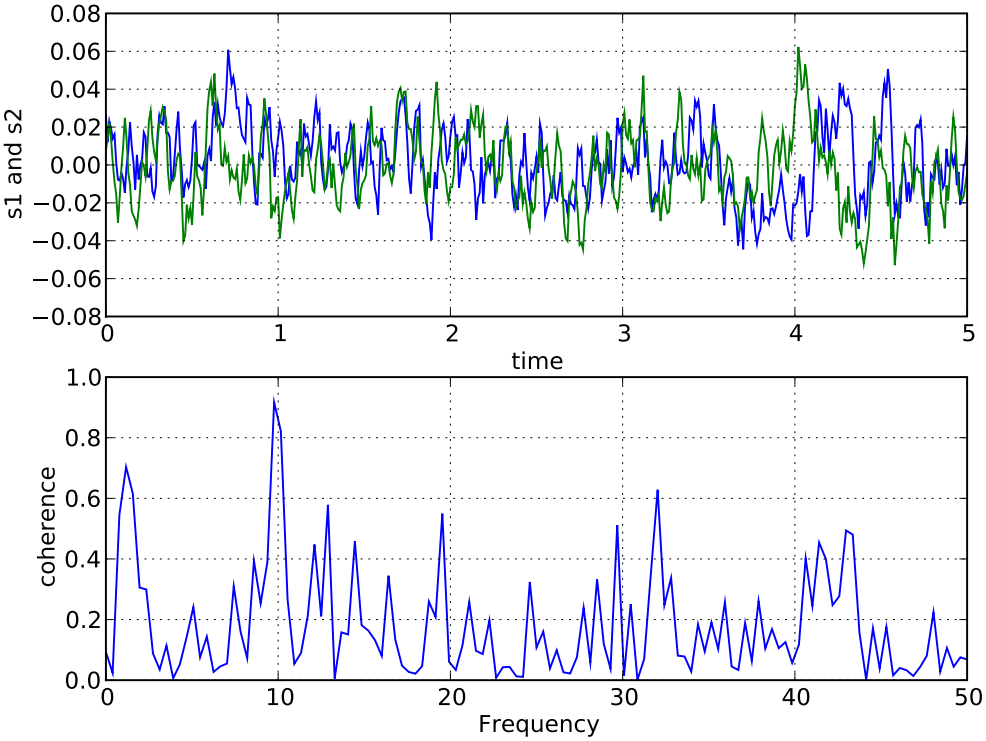
The return value is a tuple (Cxy, f), where f are the frequencies of the coherence vector.
See the psd() for a description of the optional parameters.
kwargs are applied to the lines.
References:

- Bendat & Piersol – Random Data: Analysis and Measurement Procedures, John Wiley & Sons (1986)

kwargs control the Line2D properties of the coherence plot:

Property	Description
alpha	float
animated	[True | False]
antialiased or aa	[True | False]
axes	unknown
clip_box	a matplotlib.transform.Bbox instance
clip_on	[True | False]
clip_path	a Path instance and a
color or c	any matplotlib color
contains	unknown
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' | 'steps-pre' | 'steps-mid' | 'steps-post' | 'None' | ' ' | '']
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
picker	unknown
pickradius	unknown
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

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 T/F, {}

contour(**args, **kwargs*)

`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 (TM) version in that it does not draw the polygon edges, because the contouring engine yields simply connected regions with branch cuts. To draw the 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, *Y*, and *Z* must be arrays with the same dimensions.

Z may be a masked array, but filled contouring may not handle internal masked regions correctly.

`C = contour(...)` returns a `ContourSet` 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.

***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.cm.Colormap.set_under()` and `matplotlib.cm.Colormap.set_over()` methods.

contour-only keyword arguments:

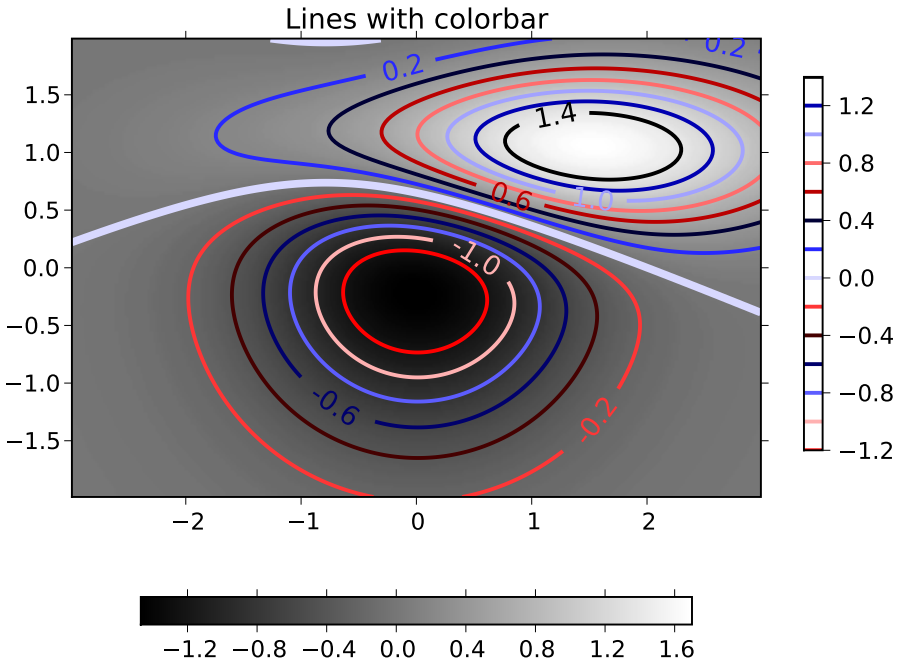
***linewidths*: [None | number | tuple of numbers]** If *linewidths* is *None*, the default width in `lines.linewidth` in `matplotlibrc` is used
If a number, all levels will be plotted with this linewidth.
If a tuple, different levels will be plotted with different linewidths in the order specified

contourf-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*.

Example:



contourf(**args, **kwargs*)

`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 (TM) version in that it does not draw the polygon edges, because the contouring engine yields simply connected regions with branch cuts. To draw the 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, *Y*, and *Z* must be arrays with the same dimensions.

Z may be a masked array, but filled contouring may not handle internal masked regions correctly.

`C = contour(...)` returns a `ContourSet` 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.

***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.cm.Colormap.set_under()` and `matplotlib.cm.Colormap.set_over()` methods.

contour-only keyword arguments:

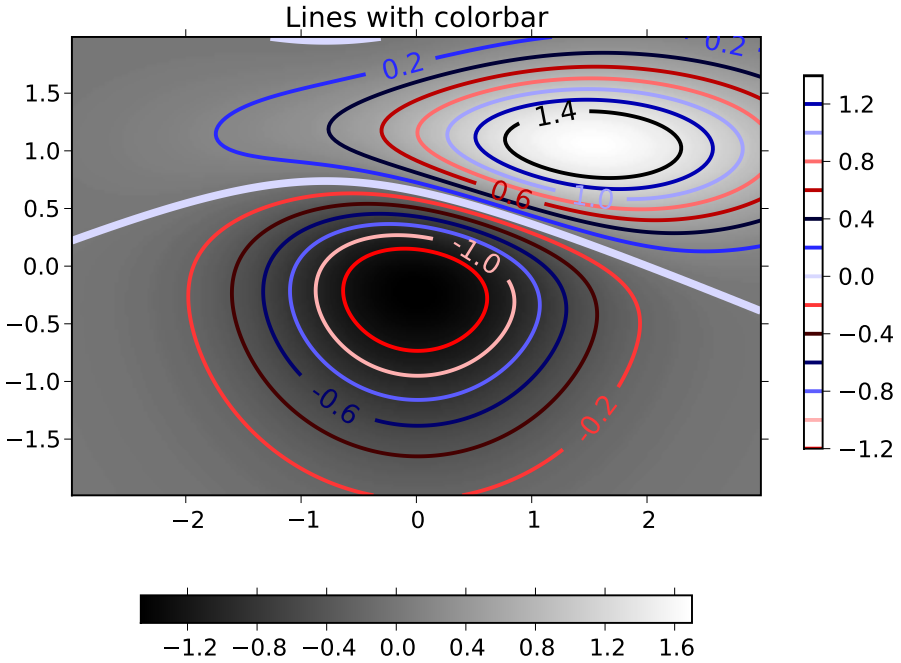
***linewidths*: [None | number | tuple of numbers]** If *linewidths* is *None*, the default width in `lines.linewidth` in `matplotlibrc` is used
If a number, all levels will be plotted with this linewidth.
If a tuple, different levels will be plotted with different linewidths in the order specified

contourf-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*.

Example:



csd(*x, y, NFFT=256, Fs=2, Fc=0, detrend=<function detrend_none at 0x8d3d7d4>, window=<function window_hanning at 0x8d3d4c4>, noverlap=0, **kwargs*)
call signature:

```
csd(x, y, NFFT=256, Fs=2, Fc=0, detrend=mlab.detrend_none,
    window=window_hanning, noverlap=0, **kwargs)
```

The cross spectral density P_{xy} by Welches 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.

See `psd()` for a description of the optional parameters.

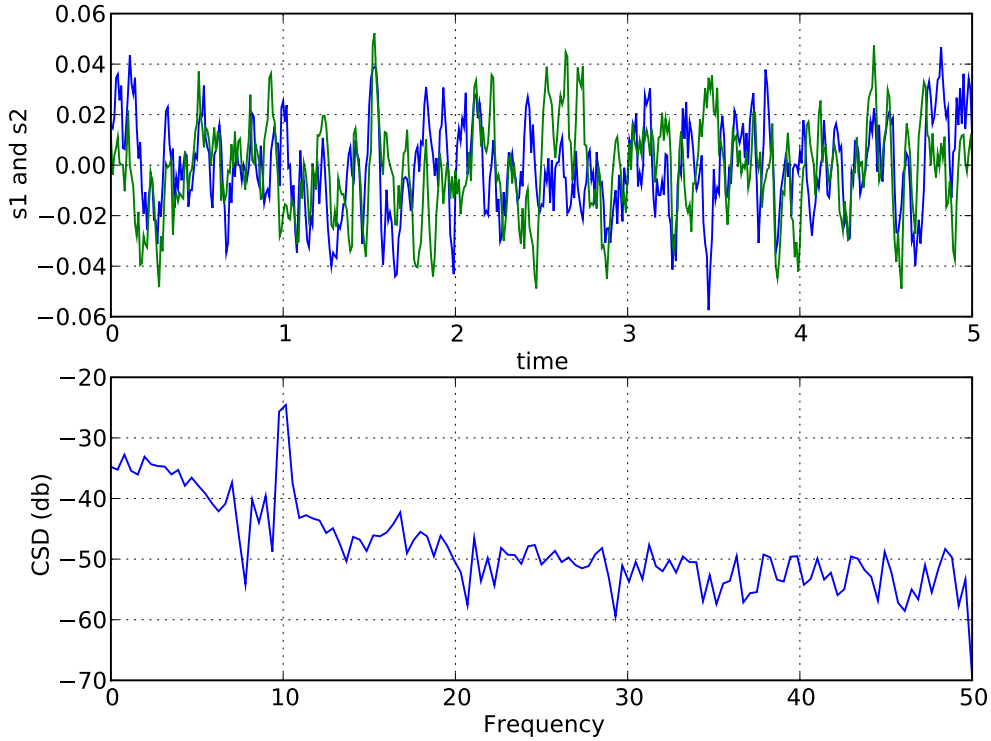
Returns the tuple (*Pxy*, *freqs*). P is the cross spectrum (complex valued), and $10\log_{10}|P_{xy}|$ is plotted.

References: Bendat & Piersol – Random Data: Analysis and Measurement Procedures, John Wiley & Sons (1986)

kwargs control the Line2D properties:

Property	Description
alpha	float
animated	[True | False]
antialiased or aa	[True | False]
axes	unknown
clip_box	a `matplotlib.transform.Bbox` instance
clip_on	[True | False]
clip_path	a `Path` instance and a
color or c	any matplotlib color
contains	unknown
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' | 'steps-pre' | 'steps-mid' | 'steps-post' | 'None' | ' ' | '']
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
picker	unknown
pickradius	unknown
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

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(*renderer=None, inframe=False*)
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, **kwargs*)
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)
```

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, Nx2 array-like]** If a scalar number, len(N) array-like object, or an Nx1 array-like object, errorbars are drawn +/- value.
If a rank-1, Nx2 Numpy array, errorbars are drawn at -column1 and +column2

***fmt*: '-'** The plot format symbol for *y*. If *fmt* is *None*, just plot the errorbars with no line symbols. This can be useful for creating a bar plot with errorbars.

***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 size of the error bar caps in points

***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*.

All other keyword arguments are passed on to the plot command for the markers, so you can add additional key=value pairs to control the errorbar 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)
```

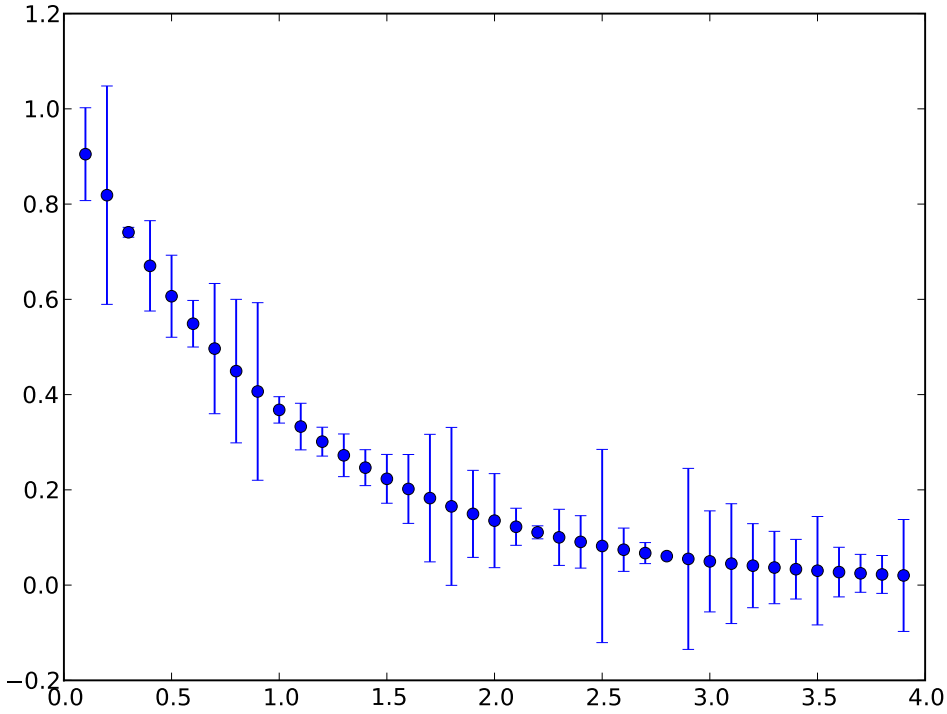
where *mfc*, *mec*, *ms* and *mew* are aliases for the longer property names, *markerfacecolor*, *markeredgecolor*, *markersize* and *markeredgewith*.

valid kwargs for the marker properties are

Property	Description
alpha	float
animated	[True | False]
antialiased or aa	[True | False]
axes	unknown
clip_box	a `matplotlib.transform.Bbox` instance
clip_on	[True | False]
clip_path	a `Path` instance and a
color or c	any matplotlib color
contains	unknown
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' | 'steps-pre' | 'steps-mid' | 'steps-post' | 'None' | ' ' | '']
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
picker	unknown
pickradius	unknown
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

Return value is a length 3 tuple. The first element is the `Line2D` instance for the *y* symbol lines. The second element is a list of error bar cap lines, the third element is a list of `LineCollection` instances for the horizontal and vertical error ranges.

Example:



fill(**args, **kwargs*)

call signature:

```
fill(*args, **kwargs)
```

Plot filled polygons. *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 `poly_between()`, eg.:

```
xs, ys = poly_between(x, 0, y)
axes.fill(xs, ys, facecolor='red', alpha=0.5)
```

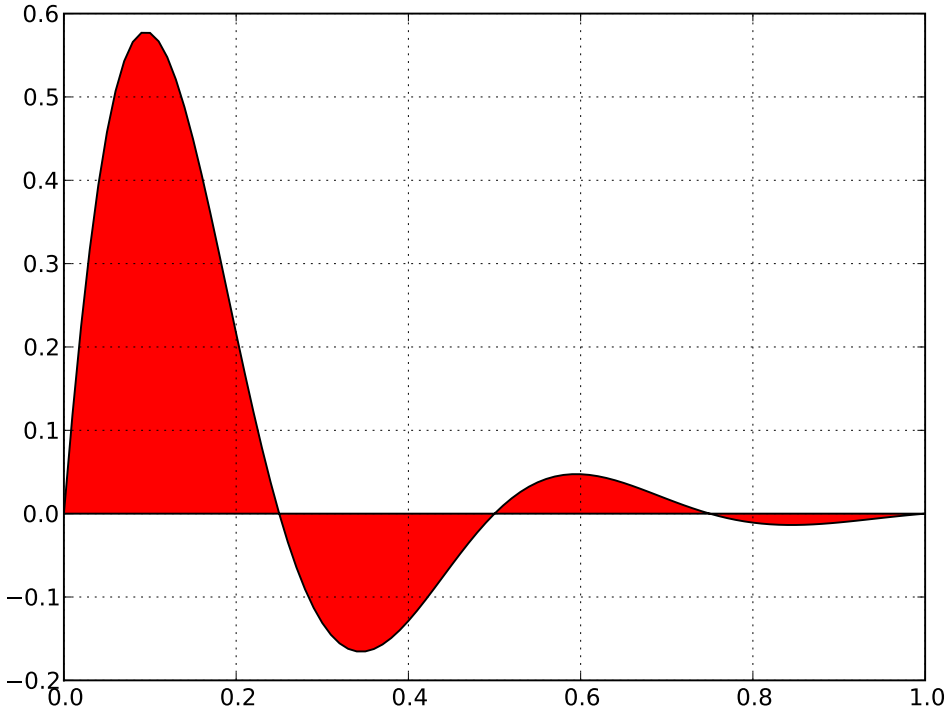
See `examples/pylab_examples/fill_between.py` for more examples.

The *closed* kwarg will close the polygon when *True* (default).

kwargs control the Polygon properties:

Property	Description
aa	[True | False] or None for default
alpha	float
animated	[True | False]
antialiased	[True | False] or None for default
axes	an axes instance
clip_box	a `matplotlib.transform.Bbox` instance
clip_on	[True | False]
clip_path	a `Path` instance and a
contains	unknown
ec	mpl color spec, or None for default, or 'none' for no color
edgecolor	mpl color spec, or None for default, or 'none' for no color
facecolor	mpl color spec, or None for default, or 'none' for no color
fc	mpl color spec, or None for default, or 'none' for no color
figure	a `matplotlib.figure.Figure` instance
fill	[True | False]
hatch	unknown
label	any string
linestyle	['solid' | 'dashed' | 'dashdot' | 'dotted']
linewidth	float or None for default
lod	[True | False]
ls	['solid' | 'dashed' | 'dashdot' | 'dotted']
lw	float or None for default
picker	[None|float|boolean|callable]
transform	unknown
visible	[True | False]
zorder	any number

Example:



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

get_adjustable()

get_anchor()

get_aspect()

get_autoscale_on()
Get whether autoscaling is applied on plot commands

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 release 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_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_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_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_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()
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 [*xmin*, *xmax*]

get_xmajorticklabels()
Get the xtick labels as a list of Text instances

get_xminorticklabels()
Get the xtick labels as a list of Text instances

get_xscale()

get_xticklabels(*minor=False*)
Get the xtick 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()
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()
Returns the 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 [*ymin*, *ymax*]

get_ymajorticklabels()
Get the xtick labels as a list of Text instances

get_yminorticklabels()
Get the xtick labels as a list of Text instances

get_yscale()

get_yticklabels(*minor=False*)
Get the xtick 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, **kwargs*)
call signature:

```
grid(self, b=None, **kwargs)
```

Set the axes grids on or off; *b* is a boolean

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*

kawrgs are used to set the grid line properties, eg:

```
ax.grid(color='r', linestyle='-', linewidth=2)
```

Valid `Line2D` kwargs are

Property	Description
alpha	float
animated	[True | False]
antialiased or aa	[True | False]
axes	unknown
clip_box	a `matplotlib.transform.Bbox` instance
clip_on	[True | False]
clip_path	a `Path` instance and a
color or c	any matplotlib color
contains	unknown
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' | 'steps-pre' | 'steps-mid' | 'steps-post' | 'None' | ' ' | '']
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
picker	unknown
pickradius	unknown
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

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', cmap=None, norm=None, vmin=None, vmax=None, alpha=1.0, linewidths=None, edgecolors='none', reduce_C_function=<function mean at 0x85c7d14>, **kwargs*)

call signature:

```
hexbin(x, y, C = None, gridsize = 100, bins = None,
       xscale = 'linear', yscale = 'linear',
       cmap=None, norm=None, vmin=None, vmax=None,
       alpha=1.0, linewidths=None, edgecolors='none'
       reduce_C_function = np.mean,
       **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.

Other keyword arguments controlling color mapping and normalization arguments:

***cmap*: [None | Colormap]** a `matplotlib.cm.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** 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:

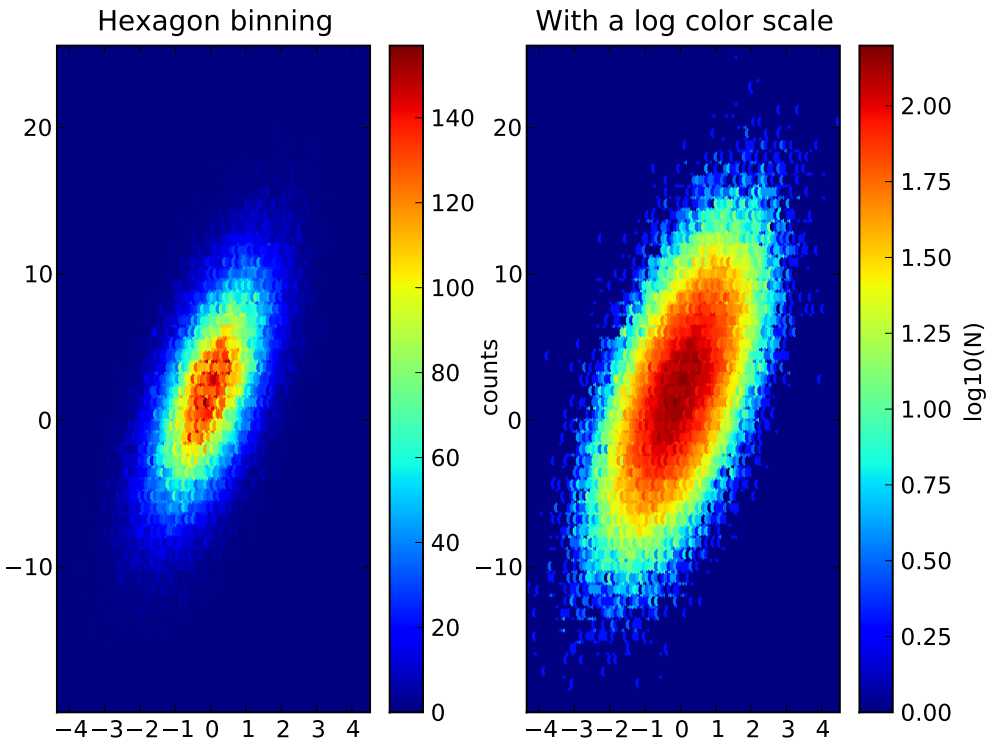
***edgecolors*: [None | mpl color | color sequence]** If 'none', draws the edges in the same color as the fill color. This is the default, as it avoids unsightly unpainted pixels between the hexagons.
If *None*, draws the outlines in the default color.
If a matplotlib color arg or sequence of rgba tuples, draws the outlines in the specified color.

Here are the standard descriptions of all the `Collection` kwargs:

Property	Description
alpha	float
animated	[True | False]
antialiased	Boolean or sequence of booleans
antialiaseds	Boolean or sequence of booleans
array	unknown
axes	an axes instance
clim	a length 2 sequence of floats
clip_box	a `matplotlib.transform.Bbox` instance
clip_on	[True | False]
clip_path	a `Path` instance and a
cmap	a colormap
color	matplotlib color arg or sequence of rgba tuples
colorbar	unknown
contains	unknown
dashes	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
edgecolor	matplotlib color arg or sequence of rgba tuples
edgecolors	matplotlib color arg or sequence of rgba tuples
facecolor	matplotlib color arg or sequence of rgba tuples
facecolors	matplotlib color arg or sequence of rgba tuples
figure	a `matplotlib.figure.Figure` instance
label	any string
linestyle	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
linestyles	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
linewidth	float or sequence of floats
linewidths	float or sequence of floats
lod	[True | False]
lw	float or sequence of floats
norm	unknown
offsets	float or sequence of floats
picker	[None|float|boolean|callable]
pickradius	unknown
transform	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.

Example:



hist(*x, bins=10, range=None, normed=False, cumulative=False, bottom=None, histtype='bar', align='mid', orientation='vertical', rwidth=None, log=False, **kwargs*)
call signature:

```
hist(x, bins=10, range=None, normed=False, cumulative=False,
     bottom=None, histtype='bar', align='mid',
     orientation='vertical', rwidth=None, log=False, **kwargs)
```

Compute 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.

Keyword arguments:

***bins*:** either an integer number of bins or a sequence giving the bins. *x* are the data to be binned. *x* can be an array or a 2D array with multiple data in its columns. Note, if *bins* is an integer input argument=numbins, *bins* + 1 bin edges will be returned, compatible with the semantics of `numpy.histogram()` with the *new* = True argument. 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.

***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))
```

***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 one. 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
- **'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 this 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'.

***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.

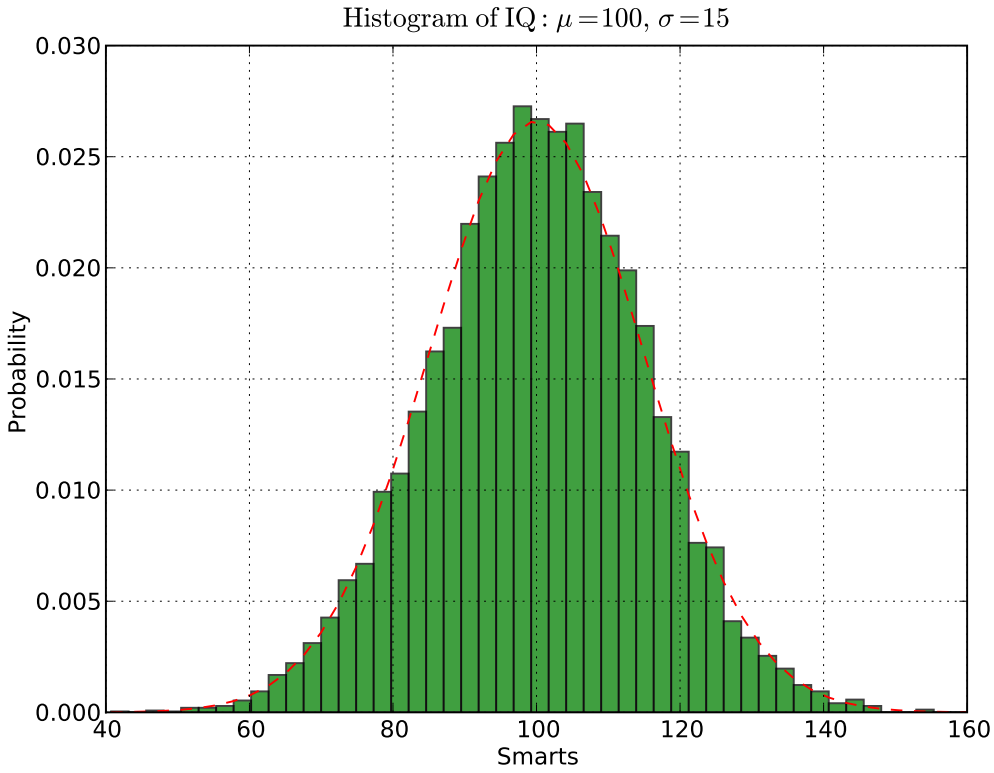
kwargs are used to update the properties of the hist `Rectangle` instances:

Property	Description
aa	[True | False] or None for default
alpha	float
animated	[True | False]
antialiased	[True | False] or None for default
axes	an axes instance
clip_box	a `matplotlib.transform.Bbox` instance
clip_on	[True | False]
clip_path	a `Path` instance and a
contains	unknown
ec	mpl color spec, or None for default, or 'none' for no color
edgecolor	mpl color spec, or None for default, or 'none' for no color
facecolor	mpl color spec, or None for default, or 'none' for no color
fc	mpl color spec, or None for default, or 'none' for no color
figure	a `matplotlib.figure.Figure` instance
fill	[True | False]
hatch	unknown
label	any string
linestyle	['solid' | 'dashed' | 'dashdot' | 'dotted']
linewidth	float or None for default
lod	[True | False]
ls	['solid' | 'dashed' | 'dashdot' | 'dotted']
lw	float or None for default
picker	[None|float|boolean|callable]
transform	unknown
visible	[True | False]
zorder	any number

You can use labels for your histogram, and only the first `Rectangle` gets the label (the others get the magic string '_nolegend_'. This will make the histograms work in the intuitive way for bar charts:

```
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()
```

Example:



hlines(*y, xmin, xmax, colors='k', linestyles='solid', label='', **kwargs*)

call signature:

```
hlines(y, xmin, xmax, colors='k', linestyle='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.

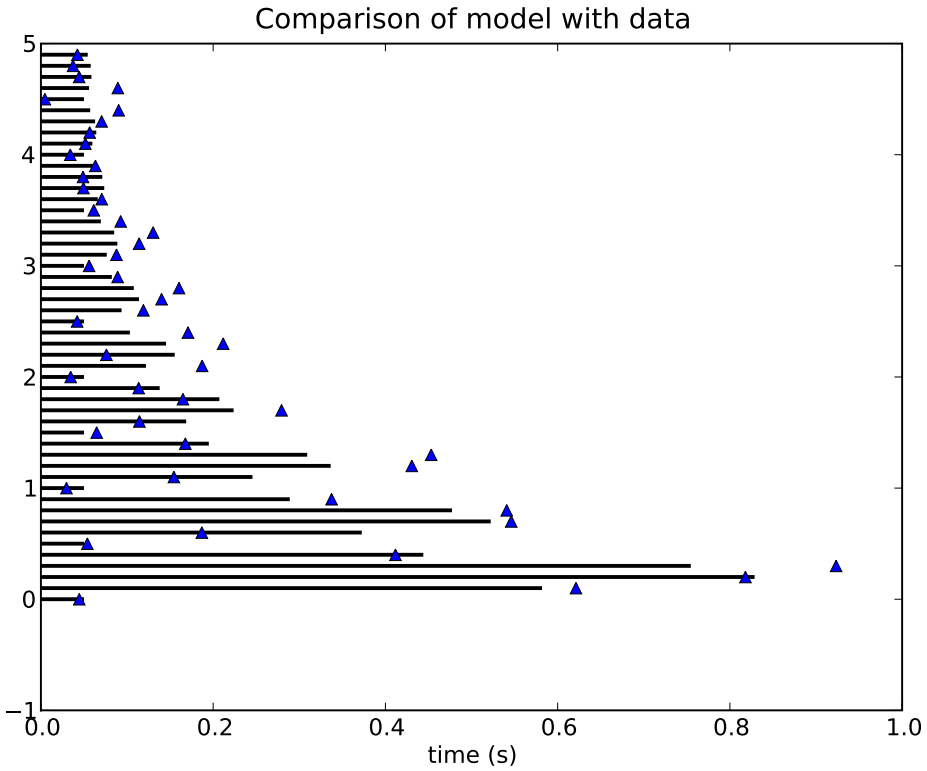
***xmin* and *xmax*:** can be scalars or `len(x)` numpy arrays. If they are scalars, then the respective values are constant, else the widths of the lines are determined by *xmin* and *xmax*.

Optional keyword arguments:

***colors*:** a line collections color argument, either a single color or a `len(y)` list of colors

***linestyle*:** ['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=1.0, vmin=None, vmax=None, origin=None, extent=None, shape=None, filternorm=1, filterrad=4.0, imlim=None, resample=None, **kwargs*)

call signature:

```
imshow(X, cmap=None, norm=None, aspect=None, interpolation=None,
       alpha=1.0, 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.cm.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*, 'nearest', 'bilinear', 'bicubic', 'spline16', 'spline36', 'hanning', 'hamming', 'hermite', 'kaiser', 'quadric', 'catrom', 'gaussian', 'bessel', 'mitchell', 'sinc', 'lanczos', 'blackman'
If *interpolation* is *None*, default to rc `image.interpolation`. See also the *filternorm* and *filterrad* parameters

***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)

***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)]** Eata values of 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

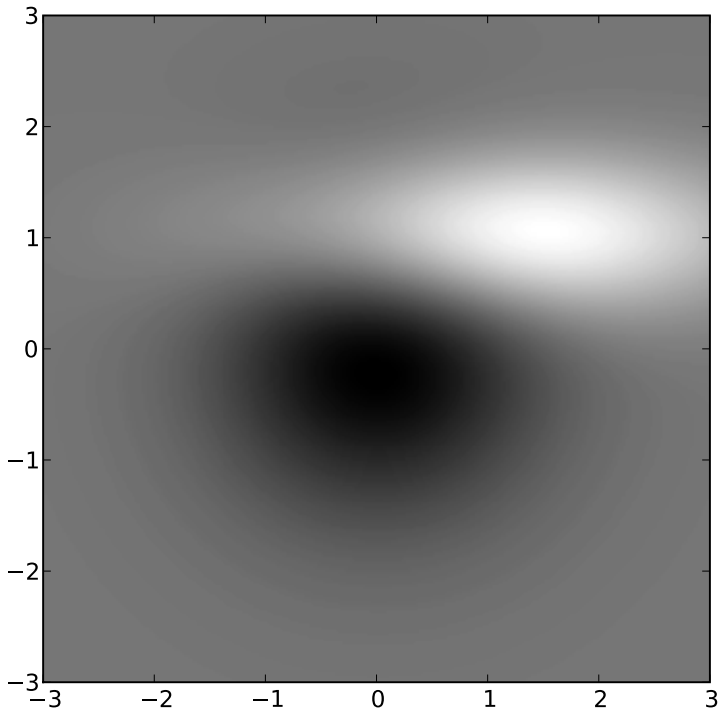
***filternorm*:** A parameter for the antigrain image resize filter. From the antigrain documentation, if *filternorm* = 1, the filter normalizes integer values and corrects the rounding errors. It doesn't do anything with the source floating point values, it corrects only integers according to the rule 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
alpha	float
animated	[True | False]
axes	an axes instance
clip_box	a `matplotlib.transform.Bbox` instance
clip_on	[True | False]
clip_path	a `Path` instance and a
contains	unknown
figure	a `matplotlib.figure.Figure` instance
label	any string
lod	[True | False]
picker	[None|float|boolean|callable]
transform	unknown
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*)

call signature:

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

Place a legend on the current axes 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

If none of these are locations are suitable, loc can be a 2-tuple giving x,y in axes coords, ie:

```
loc = 0, 1 # left top
loc = 0.5, 0.5 # center
```

Keyword arguments:

***isaxes*: [True | False]** Indicates that this is an axes legend

***numpoints*: integer** The number of points in the legend line, default is 4

***prop*: [None | FontProperties]** A `matplotlib.font_manager.FontProperties` instance, or *None* to use rc settings.

***pad*: [None | scalar]** The fractional whitespace inside the legend border, between 0 and 1. If *None*, use rc settings.

***markerscale*: [None | scalar]** The relative size of legend markers vs. original. If *None*, use rc settings.

***shadow*: [None | False | True]** If *True*, draw a shadow behind legend. If *None*, use rc settings.

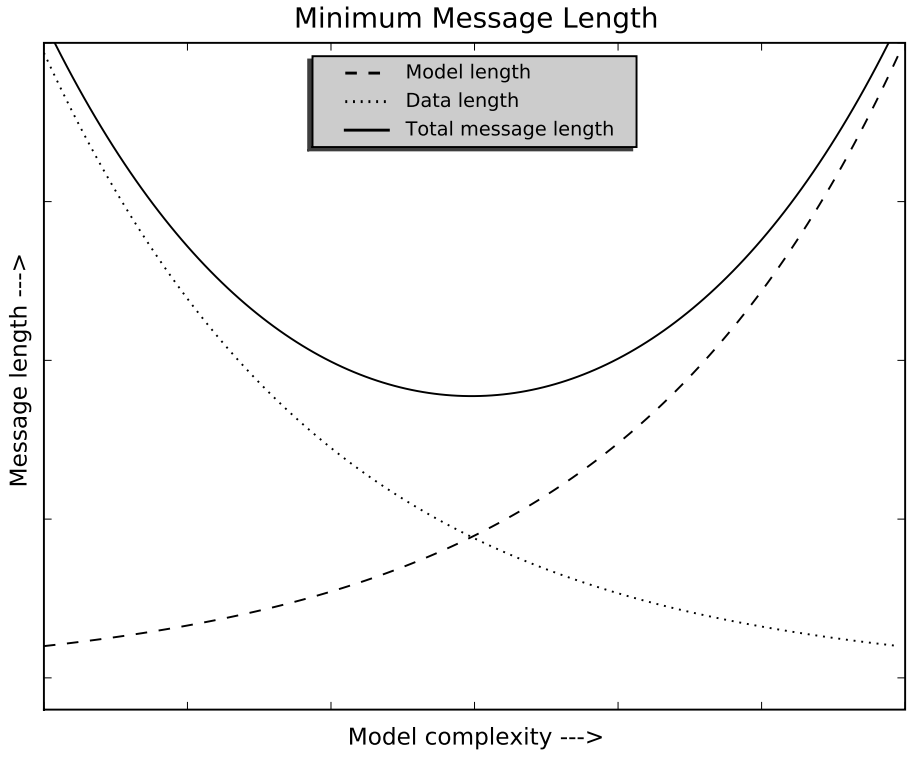
***labelsep*: [None | scalar]** The vertical space between the legend entries. If *None*, use rc settings.

***handlelen*: [None | scalar]** The length of the legend lines. If *None*, use rc settings.

***handletextsep*: [None | scalar]** The space between the legend line and legend text. If *None*, use rc settings.

***axespad*: [None | scalar]** The border between the axes and legend edge. If *None*, use rc settings.

Example:



loglog(**args, **kwargs*)

call signature:

```
loglog(*args, **kwargs)
```

Make a plot with log scaling on the *x* and *y* axis.

loglog() supports all the keyword arguments of plot() and matplotlib.axes.Axes.set_xscale()/matplotlib.axes.Axes.set_yscale().

Notable keyword arguments:

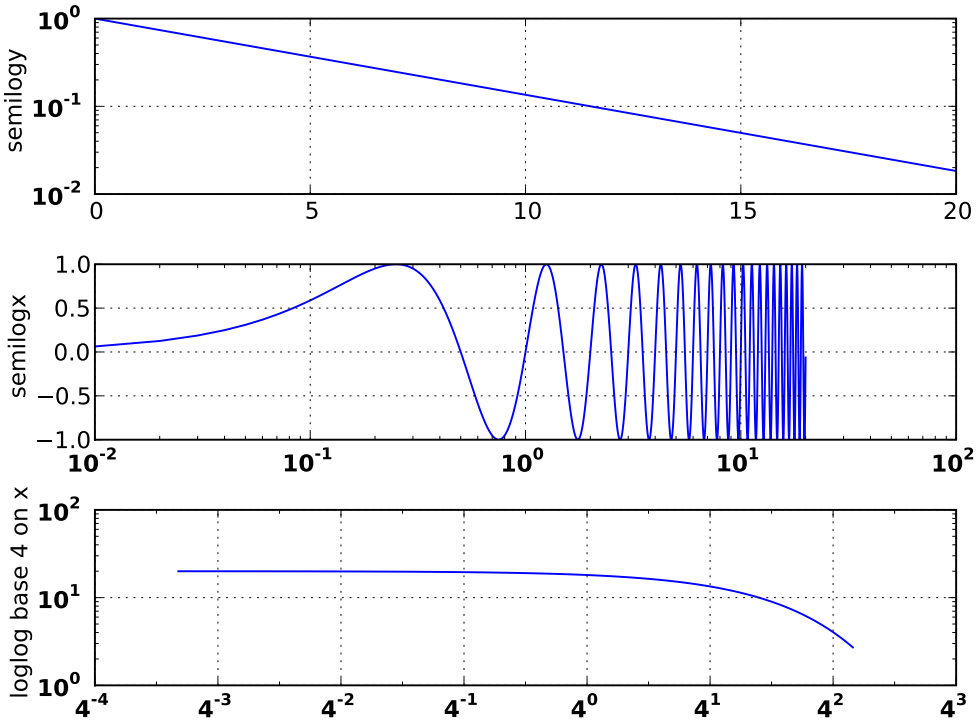
***basex/basey*: scalar > 1** base of the *x*/*y* logarithm

***subsx/subsy*: [None | sequence]** the location of the minor *x*/*y* ticks; *None* defaults to autosubs, which depend on the number of decades in the plot; see matplotlib.axes.Axes.set_xscale()/matplotlib.axes.Axes.set_yscale() for details

The remaining valid kwargs are Line2D properties:

Property	Description
alpha	float
animated	[True | False]
antialiased or aa	[True | False]
axes	unknown
clip_box	a `matplotlib.transform.Bbox` instance
clip_on	[True | False]
clip_path	a `Path` instance and a
color or c	any matplotlib color
contains	unknown
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' | 'steps-pre' | 'steps-mid' | 'steps-post' | 'None' | ' ' | '']
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
picker	unknown
pickradius	unknown
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

Example:



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 *extent*, *origin*, *interpolation*, and *aspect*; use care in overriding the *extent* and *origin* kwargs, because they interact. (Also, if you want to change them, you probably should be using imshow directly in your own version of matshow.)

Returns: an `matplotlib.image.AxesImage` instance.

pcolor(**args, **kwargs*)

call signatures:

```
pcolor(C, **kwargs)
pcolor(X, Y, C, **kwargs)
```

Create a pseudocolor plot of a 2-D array.

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.cm.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 are *None*, the min and max of the color array *C* is used. If you pass a *norm* instance, *vmin* and *vmax* will be ignored.

***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(TM).

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** the alpha blending value

Return value is a `matplotlib.collection.Collection` instance. The grid orientation follows the Matlab(TM) 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
alpha	float
animated	[True | False]
antialiased	Boolean or sequence of booleans
antialiaseds	Boolean or sequence of booleans
array	unknown
axes	an axes instance
clim	a length 2 sequence of floats
clip_box	a `matplotlib.transform.Bbox` instance
clip_on	[True | False]
clip_path	a `Path` instance and a
cmap	a colormap
color	matplotlib color arg or sequence of rgba tuples
colorbar	unknown
contains	unknown
dashes	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
edgecolor	matplotlib color arg or sequence of rgba tuples
edgecolors	matplotlib color arg or sequence of rgba tuples
facecolor	matplotlib color arg or sequence of rgba tuples
facecolors	matplotlib color arg or sequence of rgba tuples
figure	a `matplotlib.figure.Figure` instance
label	any string
linestyle	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
linestyles	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
linewidth	float or sequence of floats
linewidths	float or sequence of floats
lod	[True | False]
lw	float or sequence of floats
norm	unknown
offsets	float or sequence of floats
picker	[None|float|boolean|callable]
pickradius	unknown
transform	unknown
visible	[True | False]
zorder	any number

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 cm Colormap instance from cm. If None, use rc settings.

***norm*: [None | Normalize]** An mcolors.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** 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*)

call signatures:

```
pcolormesh(C)
pcolormesh(X, Y, C)
pcolormesh(C, **kwargs)
```

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.cm.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 are *None*, the min and max of the color array *C* is used. If you pass a *norm* instance, *vmin* and *vmax* will be ignored.

***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(TM).

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** the alpha blending value

Return value is a `matplotlib.collection.QuadMesh` object.

See `pcolor()` for an explanation of the grid orientation and the expansion of 1-D *X* and/or *Y* to 2-D arrays.

kwargs can be used to control the `matplotlib.collections.QuadMesh` properties:

Property	Description
alpha	float
animated	[True | False]
antialiased	Boolean or sequence of booleans
antialiaseds	Boolean or sequence of booleans
array	unknown
axes	an axes instance
clim	a length 2 sequence of floats
clip_box	a `matplotlib.transform.Bbox` instance
clip_on	[True | False]
clip_path	a `Path` instance and a
cmap	a colormap
color	matplotlib color arg or sequence of rgba tuples
colorbar	unknown
contains	unknown
dashes	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
edgecolor	matplotlib color arg or sequence of rgba tuples
edgecolors	matplotlib color arg or sequence of rgba tuples
facecolor	matplotlib color arg or sequence of rgba tuples
facecolors	matplotlib color arg or sequence of rgba tuples
figure	a `matplotlib.figure.Figure` instance
label	any string
linestyle	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
linestyles	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
linewidth	float or sequence of floats
linewidths	float or sequence of floats
lod	[True | False]
lw	float or sequence of floats
norm	unknown
offsets	float or sequence of floats
picker	[None|float|boolean|callable]
pickradius	unknown
transform	unknown
visible	[True | False]
zorder	any number

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=0.59999999999999998, shadow=False, labeldistance=1.1000000000000001*)

call signature:

```
pie(x, explode=None, labels=None,
    colors=('b', 'g', 'r', 'c', 'm', 'y', 'k', 'w'),
    autopct=None, pctdistance=0.6, labeldistance=1.1, shadow=False)
```

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.

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.

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 the 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.

The following line styles are supported:

```
-     # solid line
--    # dashed line
-.    # dash-dot line
:     # dotted line
.     # points
,     # pixels
```

```
o     # circle symbols
^     # triangle up symbols
v     # triangle down symbols
<     # triangle left symbols
>     # triangle right symbols
s     # square symbols
+     # plus symbols
x     # cross symbols
D     # diamond symbols
d     # thin diamond symbols
1     # tripod down symbols
2     # tripod up symbols
3     # tripod left symbols
4     # tripod right symbols
h     # hexagon symbols
H     # rotated hexagon symbols
p     # pentagon symbols
|     # vertical line symbols
_     # horizontal line symbols
steps # use gnuplot style 'steps' # kwarg only
```

The following color abbreviations are supported:

```
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.

The kwargs are `Line2D` properties:

Property	Description
alpha	float
animated	[True | False]
antialiased or aa	[True | False]
axes	unknown
clip_box	a `matplotlib.transform.Bbox` instance
clip_on	[True | False]
clip_path	a `Path` instance and a
color or c	any matplotlib color
contains	unknown
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' | 'steps-pre' | 'steps-mid' | 'steps-post' | 'None' | ' ' | '']
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
picker	unknown
pickradius	unknown
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

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*)

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.

See `dates` for helper functions `date2num()`, `num2date()` and `drange()` for help on creating the required floating point dates.

Keyword arguments:

***fmt*: string** The plot format string.

***tz*: [None | timezone string]** 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.ticker.AutoDateLocator` (if the tick locator is not already set to a `matplotlib.ticker.DateLocator` instance) and the default tick formatter to `matplotlib.ticker.AutoDateFormatter` (if the tick formatter is not already set to a `matplotlib.ticker.DateFormatter` instance).

Valid kwargs are `Line2D` properties:

Property	Description
alpha	float
animated	[True | False]
antialiased or aa	[True | False]
axes	unknown
clip_box	a `matplotlib.transform.Bbox` instance
clip_on	[True | False]
clip_path	a `Path` instance and a
color or c	any matplotlib color
contains	unknown
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' | 'steps-pre' | 'steps-mid' | 'steps-post' | 'None' | ' ' | '']
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
picker	unknown
pickradius	unknown
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

psd(*x, NFFT=256, Fs=2, Fc=0, detrend=<function detrend_none at 0x8d3d7d4>, window=<function window_hanning at 0x8d3d4c4>, noverlap=0, **kwargs*)
call signature:

```
psd(x, NFFT=256, Fs=2, Fc=0, detrend=mlab.detrend_none,
    window=mlab.window_hanning, noverlap=0, **kwargs)
```

The power spectral density by Welches average periodogram method. The vector *x* is divided into *NFFT* length segments. Each segment is detrended by function *detrend* and windowed by function *window*. *noperlap* 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 length of the fft segment, must be a power of 2

***Fs*: integer** The sampling frequency.

***Fc*: integer** The center frequency of *x* (defaults to 0), which offsets the yextents of the image to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

***detrend*:** 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*:** The function used to window the segments. *window* is a function, unlike in matlab where it is a vector. `pylab` defines `window_none()`, and `window_hanning()`, but you can use a custom function as well.

***noverlap*: integer** Gives the length of the overlap between segments.

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
alpha	float
animated	[True | False]
antialiased or aa	[True | False]
axes	unknown
clip_box	a `matplotlib.transform.Bbox` instance
clip_on	[True | False]
clip_path	a `Path` instance and a
color or c	any matplotlib color
contains	unknown
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' | 'steps-pre' | 'steps-mid' | 'steps-post' | 'None' | ' ' | '']
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
picker	unknown
pickradius	unknown
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

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*, ** are not supported at present.

Keyword arguments:

***units*: ['width' | 'height' | 'dots' | 'inches' | 'x' | 'y']** 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' or 'y': *X* or *Y* data units

In all cases 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.

The arrows scale differently depending on the units, however. 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.

***scale*: [None | float]** data units per arrow 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.

***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
alpha	float
animated	[True | False]
antialiased	Boolean or sequence of booleans
antialiaseds	Boolean or sequence of booleans
array	unknown
axes	an axes instance
clim	a length 2 sequence of floats
clip_box	a `matplotlib.transform.Bbox` instance
clip_on	[True | False]
clip_path	a `Path` instance and a
cmap	a colormap
color	matplotlib color arg or sequence of rgba tuples
colorbar	unknown
contains	unknown
dashes	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
edgecolor	matplotlib color arg or sequence of rgba tuples
edgecolors	matplotlib color arg or sequence of rgba tuples
facecolor	matplotlib color arg or sequence of rgba tuples
facecolors	matplotlib color arg or sequence of rgba tuples
figure	a `matplotlib.figure.Figure` instance
label	any string
linestyle	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
linestyles	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
linewidth	float or sequence of floats
linewidths	float or sequence of floats
lod	[True | False]
lw	float or sequence of floats
norm	unknown
offsets	float or sequence of floats
picker	[None|float|boolean|callable]
pickradius	unknown
transform	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

scatter(*x, y, s=20, c='b', marker='o', cmap=None, norm=None, vmin=None, vmax=None, alpha=1.0, linewidths=None, faceted=True, verts=None, **kwargs*)
call signatures:

```
scatter(x, y, s=20, c='b', marker='o', cmap=None, norm=None,
        vmin=None, vmax=None, alpha=1.0, linewidths=None,
        verts=None, **kwargs)
```

Make a scatter plot of *x* versus *y*, where *x*, *y* are 1-D sequences 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:

Value	Description
‘s’	square
‘o’	circle
‘^’	triangle up
‘>’	triangle right
‘v’	triangle down
‘<’	triangle left
‘d’	diamond
‘p’	pentagram
‘h’	hexagon
‘8’	octagon
‘+’	plus
‘x’	cross

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

Finally, *marker* can be (*verts*, 0): *verts* is a sequence of (*x*, *y*) vertices for a custom scatter symbol. Alternatively, use the kwarg combination *marker* = *None*, *verts* = *verts*.

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. 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** 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*:** ‘none’ to plot faces with no outlines

***facecolors*:** ‘none’ to plot unfilled outlines

Here are the standard descriptions of all the Collection kwargs:

Property	Description
alpha	float
animated	[True | False]
antialiased	Boolean or sequence of booleans
antialiaseds	Boolean or sequence of booleans
array	unknown
axes	an axes instance
clim	a length 2 sequence of floats
clip_box	a matplotlib.transform.Bbox instance
clip_on	[True | False]
clip_path	a Path instance and a
cmap	a colormap
color	matplotlib color arg or sequence of rgba tuples
colorbar	unknown
contains	unknown
dashes	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
edgecolor	matplotlib color arg or sequence of rgba tuples
edgecolors	matplotlib color arg or sequence of rgba tuples
facecolor	matplotlib color arg or sequence of rgba tuples
facecolors	matplotlib color arg or sequence of rgba tuples
figure	a matplotlib.figure.Figure instance
label	any string
linestyle	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
linestyles	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
linewidth	float or sequence of floats
linewidths	float or sequence of floats
lod	[True | False]
lw	float or sequence of floats
norm	unknown
offsets	float or sequence of floats
picker	[None|float|boolean|callable]
pickradius	unknown
transform	unknown
visible	[True | False]
zorder	any number

A Collection instance is returned.

semilogx(**args, **kwargs*)

call signature:

```
semilogx(*args, **kwargs)
```

Make a plot with log scaling on the *x* axis.

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.

The remaining valid kwargs are `Line2D` properties:

Property	Description
alpha	float
animated	[True | False]
antialiased or aa	[True | False]
axes	unknown
clip_box	a `matplotlib.transform.Bbox` instance
clip_on	[True | False]
clip_path	a `Path` instance and a
color or c	any matplotlib color
contains	unknown
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' | 'steps-pre' | 'steps-mid' | 'steps-post' | 'None' | ' ' | '']
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
picker	unknown
pickradius	unknown
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

See `loglog()` for example code and figure

semilogy(**args, **kwargs*)

call signature:

```
semilogy(*args, **kwargs)
```

Make a plot with log scaling on the *y* axis.

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.

The remaining valid kwargs are Line2D properties:

Property	Description
alpha	float
animated	[True | False]
antialiased or aa	[True | False]
axes	unknown
clip_box	a matplotlib.transform.Bbox instance
clip_on	[True | False]
clip_path	a Path instance and a
color or c	any matplotlib color
contains	unknown
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' | 'steps-pre' | 'steps-mid' | 'steps-post' | 'None' | ' ' | '']
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
picker	unknown
pickradius	unknown
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

See loglog() for example code and figure

set_adjustable(*adjustable*)

ACCEPTS: ['box' | 'datalim']

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

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_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:*~matplotlib.axes.Axes* figure

accepts a class:*~matplotlib.figure.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_title(*label, fontdict=None, **kwargs*)

call signature:

```
set_title(label, fontdict=None, **kwargs):
```

Set the title for the axes. See the `text()` for information of how override and the optional args work

kwargs are Text properties:

Property	Description
alpha	float
animated	[True | False]
axes	an axes instance
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
color	any matplotlib color
contains	unknown
family	['serif' | 'sans-serif' | 'cursive' | 'fantasy' | 'monospace']
figure	a `matplotlib.figure.Figure` instance
fontproperties	a matplotlib.font_manager.FontProperties instance
horizontalalignment or ha	['center' | 'right' | 'left']
label	any string
linespacing	float
lod	[True | False]
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	unknown
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

ACCEPTS: str

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.

set_xlabel(*xlabel, fontdict=None, **kwargs*)

call signature:

```
set_xlabel(xlabel, fontdict=None, **kwargs)
```

Set the label for the xaxis. See the `text()` docstring for information of how override and the optional args work.

Valid kwargs are Text properties:

Property	Description
alpha	float
animated	[True | False]
axes	an axes instance
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
color	any matplotlib color
contains	unknown
family	['serif' | 'sans-serif' | 'cursive' | 'fantasy' | 'monospace']
figure	a `matplotlib.figure.Figure` instance
fontproperties	a matplotlib.font_manager.FontProperties instance
horizontalalignment or ha	['center' | 'right' | 'left']
label	any string
linespacing	float
lod	[True | False]
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	unknown
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

ACCEPTS: str

set_xlim(*xmin=None, xmax=None, emit=True, **kwargs*)

call signature:

```
set_xlim(self, *args, **kwargs)
```

Set the limits for the xaxis

Returns the current xlimits as a length 2 tuple: [*xmin*, *xmax*]

Examples:

```
set_xlim((valmin, valmax))
set_xlim(valmin, valmax)
set_xlim(xmin=1) # xmax unchanged
set_xlim(xmax=1) # xmin unchanged
```

Keyword arguments:

***ymin*: scalar** the min of the ylim

***ymax*: scalar** the max of the ylim

***emit*: [True | False]** notify observers of lim change

ACCEPTS: len(2) sequence of floats

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

***subsx/subsy*:** Where to place the subticks between each major tick. Should be a sequence of integers. For example, in a log10 scale: `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`
will place 10 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: `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`
will place 10 logarithmically spaced minor ticks between each major tick.

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
alpha	float
animated	[True | False]
axes	an axes instance
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
color	any matplotlib color
contains	unknown
family	['serif' | 'sans-serif' | 'cursive' | 'fantasy' | 'monospace']
figure	a `matplotlib.figure.Figure` instance
fontproperties	a matplotlib.font_manager.FontProperties instance
horizontalalignment or ha	['center' | 'right' | 'left']
label	any string
linespacing	float
lod	[True | False]
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	unknown
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

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.

set_ylabel(*ylabel, fontdict=None, **kwargs*)

call signature:

```
set_ylabel(ylabel, fontdict=None, **kwargs)
```

Set the label for the yaxis

See the `text()` doctstring for information of how override and the optional args work

Valid kwargs are Text properties:

Property	Description
alpha	float
animated	[True | False]
axes	an axes instance
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
color	any matplotlib color
contains	unknown
family	['serif' | 'sans-serif' | 'cursive' | 'fantasy' | 'monospace']
figure	a `matplotlib.figure.Figure` instance
fontproperties	a matplotlib.font_manager.FontProperties instance
horizontalalignment or ha	['center' | 'right' | 'left']
label	any string
linespacing	float
lod	[True | False]
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	unknown
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

ACCEPTS: str

set_ylim(*ymin=None, ymax=None, emit=True, **kwargs*)

call signature:

```
set_ylim(self, *args, **kwargs):
```

Set the limits for the yaxis; v = [ymin, ymax]:

```
set_ylim((valmin, valmax))
set_ylim(valmin, valmax)
set_ylim(ymin=1) # ymax unchanged
set_ylim(ymax=1) # ymin unchanged
```

Keyword arguments:

***ymin*: scalar** the min of the ylim

***ymax*: scalar** the max of the ylim

***emit*: [True | False]** notify observers of lim change

Returns the current ylimits as a length 2 tuple

ACCEPTS: len(2) sequence of floats

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

***subsx/subsy*:** Where to place the subticks between each major tick. Should be a sequence of integers. For example, in a log10 scale: `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`
will place 10 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: `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`
will place 10 logarithmically spaced minor ticks between each major tick.

set_yticklabels(*labels, fontdict=None, minor=False, **kwargs*)

call signature:

```
set_yticklabels(labels, fontdict=None, minor=False, **kwargs)
```

Set the ytick labels with list of strings *labels*. Return a list of `Text` instances.

kwargs set `Text` properties for the labels. Valid properties are

Property	Description
alpha	float
animated	[True | False]
axes	an axes instance
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
color	any matplotlib color
contains	unknown
family	['serif' | 'sans-serif' | 'cursive' | 'fantasy' | 'monospace']
figure	a `matplotlib.figure.Figure` instance
fontproperties	a matplotlib.font_manager.FontProperties instance
horizontalalignment or ha	['center' | 'right' | 'left']
label	any string
linespacing	float
lod	[True | False]
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	unknown
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

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 0x8d3d7d4>, window=<function window_hanning at 0x8d3d4c4>, noverlap=128, cmap=None, xextent=None*)

call signature:

```
specgram(x, NFFT=256, Fs=2, Fc=0, detrend=mlab.detrend_none,
         window = mlab.window_hanning, noverlap=128,
         cmap=None, xextent=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*.

Keyword arguments:

***cmap*:** A `matplotlib.cm.Colormap` instance; if *None* use default determined by rc

***xextent*:** The image extent in the xaxes xextent=xmin, xmax default 0, max(bins), 0, max(freqs) where bins is the return value from mlab.specgram

See `psd()` for information on the other keyword arguments.

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 a len(times) x len(freqs) array of power
- *im* is a `matplotlib.image.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.

spy(*Z, precision=None, marker=None, markersize=None, aspect='equal', **kwargs*)

call signature:

```
spy(Z, precision=None, marker=None, markersize=None,
    aspect='equal', **kwargs)
```

`spy(Z)` plots the sparsity pattern of the 2-D array *Z*.

If *precision* is *None*, any non-zero value will be plotted; else, values of $|Z| >$ *precision* will be plotted.

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 documentation for `imshow()` for details.

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*

See documentation for `plot()` for details.

Useful values for *marker* include:

- 's' square (default)
- 'o' circle
- '.' point
- ',' pixel

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-'*)

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 this document for details and `examples/pylab_examples/stem_plot.py` for a demo.

step(*x, y, *args, **kwargs*)

call signature:

```
step(x, y, *args, **kwargs)
```

Make a step plot. 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]
If 'post', that interval has level y[i+1]
If 'mid', the jumps in *y* occur half-way between the *x*-values.

table(***kwargs*)

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):
```

Add a table to the current axes. 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
alpha	float
animated	[True | False]
axes	an axes instance
clip_box	a `matplotlib.transform.Bbox` instance
clip_on	[True | False]
clip_path	a `Path` instance and a
contains	unknown
figure	a `matplotlib.figure.Figure` instance
fontsize	a float in points
label	any string
lod	[True | False]
picker	[None|float|boolean|callable]
transform	unknown
visible	[True | False]
zorder	any number

text(*x, y, s, fontdict=None, withdash=False, **kwargs*)

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 `matplotlib.text.Text` properties:

Property	Description
alpha	float
animated	[True | False]
axes	an axes instance
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
color	any matplotlib color
contains	unknown
family	['serif' | 'sans-serif' | 'cursive' | 'fantasy' | 'monospace']
figure	a `matplotlib.figure.Figure` instance
fontproperties	a matplotlib.font_manager.FontProperties instance
horizontalalignment or ha	['center' | 'right' | 'left']
label	any string
linespacing	float
lod	[True | False]
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	unknown
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

ticklabel_format(***kwargs*)

Convenience method for manipulating the ScalarFormatter used by default for linear axes.

Optional keyword arguments:

Keyword	Description
style	['sci' (or 'scientific') | 'plain'] plain turns off scientific notation
axis	['x' | 'y' | 'both']

Only the major ticks are affected. If the method is called when the `ScalarFormatter` is not the `Formatter` being used, an `AttributeError` will be raised with no additional error message.

Additional capabilities and/or friendlier error checking may be added.

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

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

update_datalim(*xys*)
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*)
call signature:

```
vlines(x, ymin, ymax, color='k')
```

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 is a line collections color args, either a single color or a len(*x*) list of colors

linestyle is one of ['solid' | 'dashed' | 'dashdot' | 'dotted']

Returns the matplotlib.collections.LineCollection that was added.

kwargs are LineCollection properties:

Property	Description
alpha	float
animated	[True | False]
antialiased	Boolean or sequence of booleans
antialiaseds	Boolean or sequence of booleans
array	unknown
axes	an axes instance
clim	a length 2 sequence of floats
clip_box	a `matplotlib.transform.Bbox` instance
clip_on	[True | False]
clip_path	a `Path` instance and a
cmap	a colormap
color	matplotlib color arg or sequence of rgba tuples
colorbar	unknown
contains	unknown
dashes	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
edgecolor	matplotlib color arg or sequence of rgba tuples
edgecolors	matplotlib color arg or sequence of rgba tuples
facecolor	matplotlib color arg or sequence of rgba tuples
facecolors	matplotlib color arg or sequence of rgba tuples
figure	a `matplotlib.figure.Figure` instance
label	any string
linestyle	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
linestyles	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
linewidth	float or sequence of floats
linewidths	float or sequence of floats
lod	[True | False]
lw	float or sequence of floats
norm	unknown
offsets	float or sequence of floats
picker	[None|float|boolean|callable]
pickradius	unknown
segments	unknown
transform	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 the time zone to use in labeling dates. Defaults to rc value.

xaxis_inverted()

Returns True if the x-axis is inverted.

xcorr(*x, y, normed=False, detrend=<function detrend_none at 0x8d3d7d4>, usevlines=False, maxlags=None, **kwargs*)

call signature:

```
xcorr(x, y, normed=False, detrend=mlab.detrend_none,
      usevlines=False, **kwargs):
```

Plot the cross correlation between *x* and *y*. If *normed* = *True*, normalize the data but 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*:

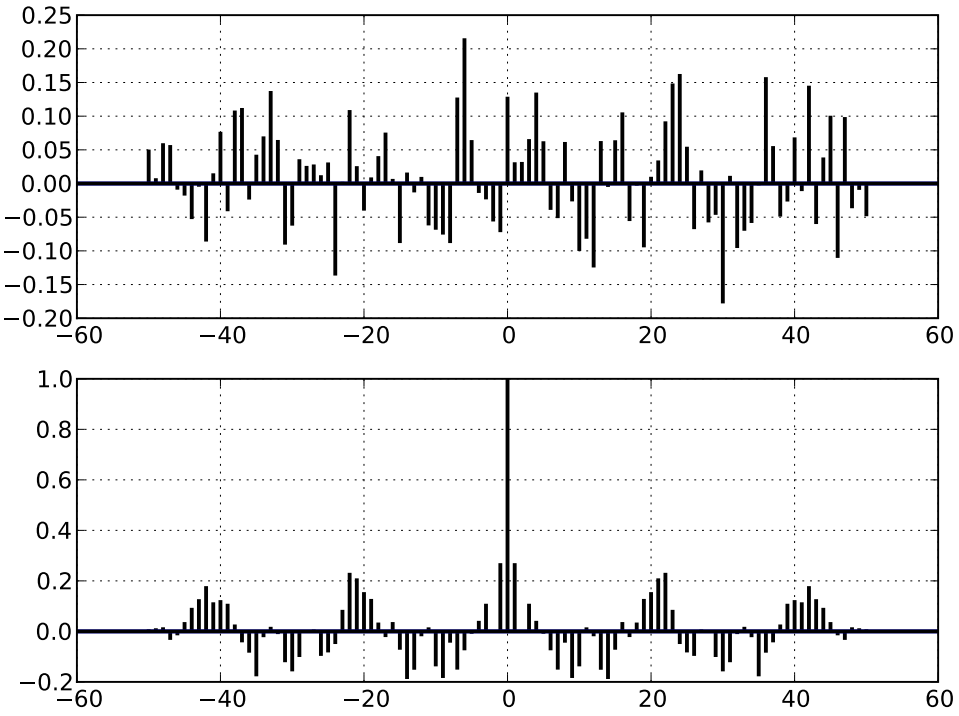
> `vlines()` rather than `plot()` is used to draw vertical lines from the origin to the xcorr. Otherwise the plotstyle is determined by the kwargs, which are `Line2D` properties.
>
> The return value is a tuple (*lags*, *c*, *linecol*, *b*) where *linecol* is the `matplotlib.collections.LineCollection` 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()` above, and `acorr()` below.

Example:



yaxis_date(*tz=None*)
Sets up y-axis ticks and labels that treat the y data as dates.

tz is the time zone to use in labeling dates. Defaults to rc value.

yaxis_inverted()
Returns True if the y-axis is inverted.

class Subplot(*fig, *args, **kwargs*)
Bases: matplotlib.axes.SubplotBase, matplotlib.axes.Axes

fig is a figure instance

args is 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 right corner and increases to the right.

If numRows<=numCols<=plotNum<10, args can be the decimal integer numRows*100 + numCols*10 + plotNum.

class 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 figure instance

args is 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 right 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

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

update_params()
update the subplot position from fig.subplotpars

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

subplot_class_factory(*axes_class=None*)

CHAPTER

TWENTYFOUR

Matplotlib axis

24.1 matplotlib.axis

Classes for the ticks and x and y axis

class **Axis**(*axes, pickradius=15*)

Bases: matplotlib.artist.Artist

Public attributes

- `transData` - transform data coords to display coords
- `transAxis` - transform axis coords to display coords

Init the axis with the parent Axes instance

cla()

clear the current axis

convert_units(*x*)

draw(*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_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_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_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' 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)

set_clip_path(*clippath, transform=None*)

set_data_interval()
Set the axis data limits

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_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_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.

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* is *data* is registered for unit conversion

zoom(*direction*)
Zoom in/out on axis; if *direction* is >0 zoom in, else zoom out

class Tick(*axes, loc, label, size=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 relative, axes coords

contains(*mouseevent*)
Test whether the mouse event occured 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(*renderer*)

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: a Path instance and a Transform instance, a Patch instance, or *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

set_view_interval(*vmin, vmax, ignore=False*)

class Ticker()

class XAxis(*axes, pickradius=15*)
Bases: matplotlib.axis.Axis

Init the axis with the parent Axes instance

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*)
return the Interval instance for this axis data limits

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.
ACCEPTS: ['top' | 'bottom' | 'both' | 'default' | 'none']

set_view_interval(*vmin, vmax, ignore=False*)

tick_bottom()
use ticks only on bottom

tick_top()
use ticks only on top

class XTick(*axes, loc, label, size=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 relative, axes coords

get_data_interval()
return the Interval instance for this axis data limits

get_minpos()

get_view_interval()
return the Interval instance for this axis view limits

set_view_interval(*vmin, vmax, ignore=False*)

update_position(*loc*)
Set the location of tick in data coords with scalar *loc*

class YAxis(*axes, pickradius=15*)
Bases: matplotlib.axis.Axis

Init the axis with the parent Axes instance

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*)
return the Interval instance for this axis data limits

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 or default) 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 on the left.
ACCEPTS: ['left' | 'right' | 'both' | 'default' | 'none']

set_view_interval(*vmin, vmax, ignore=False*)

tick_left()
use ticks only on left

tick_right()
use ticks only on right

class YTick(*axes, loc, label, size=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 relative, axes coords

get_data_interval()
return the Interval instance for this axis data limits

get_minpos()

get_view_interval()
return the Interval instance for this axis view limits

set_view_interval(*vmin, vmax*)

update_position(*loc*)
Set the location of tick in data coords with scalar loc

CHAPTER

TWENTYFIVE

Matplotlib cbook

25.1 matplotlib.cbook

error while formatting signature for matplotlib.cbook.Xlator: arg is not a Python function

A collection of utility functions and classes. Many (but not all) from the Python Cookbook – hence the name cbook

class 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 CallbackRegistry(*signals*)

Handle registering and disconnecting for a set of signals and callbacks:

```
signals = 'eat', 'drink', 'be merry'

def oneat(x):
    print 'eat', x

def ondrink(x):
    print 'drink', x

callbacks = CallbackRegistry(signals)

ideat = callbacks.connect('eat', oneat)
iddrink = callbacks.connect('drink', ondrink)

#tmp = callbacks.connect('drunk', ondrink) # this will raise a ValueError

callbacks.process('drink', 123)    # will call oneat
callbacks.process('eat', 456)      # will call ondrink
callbacks.process('be merry', 456) # nothing will be called
callbacks.disconnect(ideat)        # disconnect oneat
```

```
callbacks.process('eat', 456)      # nothing will be called
```

signals is a sequence of valid signals

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 GetRealpathAndStat()

class 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.

For example:

```
>>> g = grouper.Grouper()
>>> g.join('a', 'b')
>>> g.join('b', 'c')
>>> g.join('d', 'e')
>>> list(g)
[['a', 'b', 'c'], ['d', 'e']]
>>> g.joined('a', 'b')
True
>>> g.joined('a', 'c')
True
>>> g.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 Idle(*func*)
Bases: `matplotlib.cbook.Scheduler`

Schedule callbacks when scheduler is idle

run()

class MemoryMonitor(*nmax=20000*)

clear()

plot(*i0=0, isub=1, fig=None*)

report(*segments=4*)

xy(*i0=0, isub=1*)

class Null(**args, **kwargs*)

Null objects always and reliably "do nothing."

class 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 Scheduler()

Bases: `threading.Thread`

Base class for timeout and idle scheduling

stop()

class 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 Stack(*default=None*)

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 Timeout(*wait, func*)
Bases: matplotlib.cbook.Scheduler

Schedule recurring events with a wait time in seconds

run()

class 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.

allequal(*seq*)
Return *True* if all elements of *seq* compare equal. If *seq* is 0 or 1 length, return *True*

allpairs(*x*)
return all possible pairs in sequence *x*

Condensed by Alex Martelli from this thread on c.l.python

alltrue(*seq*)
Return *True* if all elements of *seq* evaluate to *True*. If *seq* is empty, return *False*.

class converter(*missing='Null', missingval=None*)

Base class for handling string -> python type with support for missing values

is_missing(*s*)

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.

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().

dict_delall(*d, keys*)

delete all of the *keys* from the `dict` *d*

distances_along_curve(*X*)

Computes the distance between a set of successive points in N dimensions.

where X is an MxN array or matrix. The distances between successive rows is computed. Distance is the standard Euclidean distance.

exception_to_str(*s=None*)

finddir(*o, match, case=False*)

return all attributes of *o* which match string in match. if case is True require an exact case match.

flatten(*seq, scalarp=<function is_scalar at 0x897e064>*)

this generator flattens nested containers such as

```
>>> l=( ('John', 'Hunter'), (1,23), [[[[42,(5,23)]]]])
```

so that

```
>>> for i in flatten(l): print i,
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

get_recursive_filelist(*args*)
Recurs all the files and dirs in *args* ignoring symbolic links and return the files as a list of strings

get_split_ind(*seq, N*)
seq is a list of words. Return the index into seq such that:

```
len(' '.join(seq[:ind])<=N
```

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.

is_numlike(*obj*)
return true if *obj* looks like a number

is_scalar(*obj*)
return true if *obj* is not string like and is not iterable

is_sequence_of_strings(*obj*)
Returns true if *obj* is iterable and contains strings

is_string_like(*obj*)
return true if *obj* looks like a string

is_writable_file_like(*obj*)
return true if *obj* looks like a file object with a *write* method

issubclass_safe(*x, klass*)
return issubclass(x, klass) and return False on a TypeError

isvector(*X*)
Like the Matlab (TM) function with the same name, returns true if the supplied numpy array or matrix 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

iterable(*obj*)
return true if *obj* is iterable

less_simple_linear_interpolation(*x, y, xi, extrap=False*)
This function provides simple (but somewhat less so than 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.

listFiles(*root, patterns='*', recurse=1, return_folders=0*)
Recursively list files

from Parmar and Martelli in the Python Cookbook

class 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

mkdirs(*newdir, mode=511*)

onetrue(*seq*)
Return *True* if one element of *seq* is *True*. It *seq* is empty, return *False*.

path_length(*X*)
Computes the distance travelled along a polygonal curve in N dimensions.

where X is an MxN 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).

pieces(*seq, num=2*)
Break up the *seq* into *num* tuples

popall(*seq*)
empty a list

popd(*d, *args*)
Should behave like python2.3 `dict.pop()` method; *d* is a `dict`:

```
# returns value for key and deletes item; raises a KeyError if key
# is not in dict
val = popd(d, key)

# returns value for key if key exists, else default.  Delete key,
# val item if it exists.  Will not raise a KeyError
val = popd(d, key, default)
```

print_cycles(*objects, outstream=<open file '<stdout>', mode 'w' at 0x401c0068>,**objects show_progress=False*)
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.

recursive_remove(*path*)

report_memory(*i=0*)
return the memory consumed by process

reverse_dict(*d*)
reverse the dictionary – may lose data if values are not unique!

safezip(*args*)
make sure *args* are equal len before zipping

class 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 give type

simple_linear_interpolation(*a, steps*)

soundex(*name, len=4*)

soundex module conforming to Odell-Russell algorithm

strip_math(*s*)

remove latex formatting from mathtext

to_filehandle(*fname, flag='r', 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 todate(*fmt='%Y-%m-%d', missing='Null', missingval=None*)

Bases: matplotlib.cbook.converter

convert to a date or None

use a time.strptime() format string for conversion

class 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 tofloat(*missing='Null', missingval=None*)

Bases: matplotlib.cbook.converter

convert to a float or None

class toint(*missing='Null', missingval=None*)

Bases: matplotlib.cbook.converter

convert to an int or None

class tostr(*missing='Null', missingval=''*)

Bases: matplotlib.cbook.converter

convert to string or None

unicode_safe(*s*)

unique(*x*)

Return a list of unique elements of *x*

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.

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.

wrap(*prefix, text, cols*)

wrap *text* with *prefix* at length *cols*

CHAPTER

TWENTYSIX

Matplotlib cm

26.1 matplotlib.cm

This module contains the instantiations of color mapping classes

class 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

set_colorbar(*im, ax*)
set the colorbar image and axes associated with mappable

set_norm(*norm*)
set the normalization instance

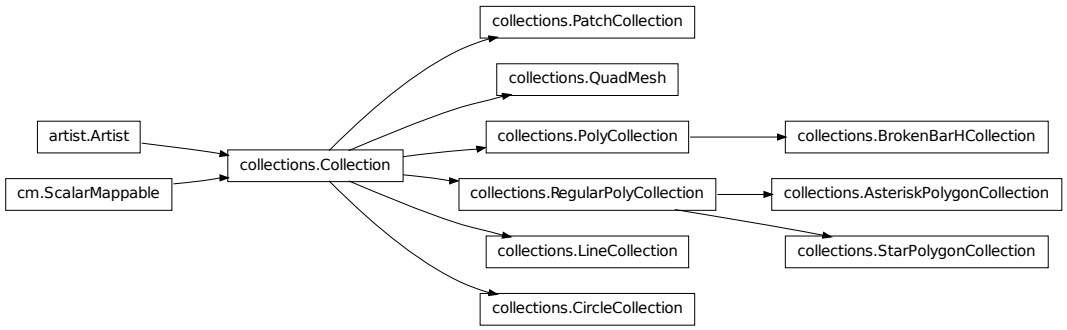
to_rgba(*x, alpha=1.0, bytes=False*)
Return a normalized rgba array corresponding to *x*. If *x* is already an rgb array, insert *alpha*; if it is already rgba, return it unchanged. If *bytes* is True, return rgba as 4 uint8s instead of 4 floats.

get_cmap(*name=None, lut=None*)
Get a colormap instance, defaulting to rc values if *name* is None

CHAPTER

TWENTYSEVEN

Matplotlib collections



27.1 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 bunch of solid line segemnts)

class **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 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.

class CircleCollection(*sizes*)

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(*renderer*)

class Collection(*edgecolors=None, facecolors=None, linewidths=None, linestyles='solid', antialiaseds=None, offsets=None, transOffset=None, norm=None, cmap=None, pickradius=5.0, **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()
- *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.

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(*renderer*)

get_dashes()

get_datalim(*transData*)

get_edgecolor()

get_edgecolors()

get_facecolor()

get_facecolors()

get_linestyle()

get_linestyles()

get_linewidth()

get_linewidths()

get_offsets()
Return the offsets for the collection.

get_paths()

get_pickradius()

get_transforms()

set_alpha(*alpha*)
Set the alpha tranparencies of the collection. *alpha* must be a float.
ACCEPTS: float

set_antialiased(*aa*)
Set the antialiasing state for rendering.
ACCEPTS: Boolean or sequence of booleans

set_antialiaseds(*aa*)
Set the antialiasing state for rendering.
ACCEPTS: Boolean or sequence of booleans

set_color(*c*)
Set both the edgecolor and the facecolor. See `set_facecolor()` and `set_edgecolor()`.
ACCEPTS: matplotlib color arg or sequence of rgba tuples

set_dashes(*ls*)
Set the linestyles(s) for the collection. ACCEPTS: ['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]

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 or 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.
ACCEPTS: matplotlib color arg or sequence of rgba tuples

set_edgecolors(*c*)
Set the edgecolor(s) of the 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.
If *c* is 'face', the edge color will always be the same as the face color.
ACCEPTS: matplotlib color arg or sequence of rgba tuples

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 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_facecolors(*c*)
Set the facecolor(s) of the 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_linestyle(*ls*)
Set the linestyles(s) for the collection. ACCEPTS: ['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]

set_linestyles(*ls*)
Set the linestyles(s) for the collection. ACCEPTS: ['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]

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*)
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_lw(*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_offsets(*offsets*)
Set the offsets for the collection. *offsets* can be a scalar or a sequence.
ACCEPTS: float or sequence of floats

set_pickradius(*pickradius*)

update_scalarmappable()
If the scalar mappable array is not none, update colors from scalar data

class 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` 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 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()

get_paths()

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_segments(*segments*)

set_verts(*segments*)

class 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.

get_paths()

class 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(*renderer*)

get_paths()

set_verts(*verts, closed=True*)
This allows one to delay initialization of the vertices.

class QuadMesh(*meshWidth, meshHeight, coordinates, showedges, antialiased=True*)
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.

convert_mesh_to_paths
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.

draw(*renderer*)

get_datalim(*transData*)

get_paths(*dataTrans=None*)

class 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(*renderer*)

get_paths()

class 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,
    )
```

CHAPTER

TWENTYEIGHT

Matplotlib colorbar

28.1 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 `matplotlib.Figure.colorbar()` method uses `make_axes()` and `Colorbar`; the `matplotlib.pyplot.colorbar()` function is a thin wrapper over `matplotlib.Figure.colorbar()`.

class Colorbar(*ax, mappable, **kw*)

Bases: `matplotlib.colorbar.ColorbarBase`

add_lines(*CS*)

Add the lines from a non-filled `ContourSet` to the colorbar.

update_bruteforce(*mappable*)

Manually change any contour line colors. This is called when the image or contour plot to which this colorbar belongs is changed.

class ColorbarBase(*ax, cmap=None, norm=None, alpha=1.0, values=None, boundaries=None, orientation='vertical', extend='neither', spacing='uniform', ticks=None, format=None, drawedges=False, filled=True*)

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()` method and pylab function.

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.

add_lines(*levels, colors, linewidths*)
Draw lines on the colorbar.

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*)

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
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

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

TWENTYNINE

Matplotlib colors

29.1 matplotlib.colors

A class for converting color arguments to RGB or RGBA

This class instantiates a single instance colorConverter that is used to convert matlab color strings to RGB. RGB is a tuple of float RGB values in the range 0-1.

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 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 ColorConverter()

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()`. If *arg* is an *RGBA* sequence and *alpha* is not *None*, *alpha* will replace the original *A*.

to_rgba_array(*c, alpha=None*)

Returns an Numpy array of *RGBA* tuples.

Accepts a single mpl color spec or a sequence of specs. If the sequence is a list or array, the items are changed in place, but an array copy is still returned.

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 Colormap(*name, N=256*)

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=1.0*)

Set color to be used for masked values.

set_over(*color='k', alpha=1.0*)

Set color to be used for high out-of-range values. Requires norm.clip = False

set_under(*color='k', alpha=1.0*)
Set color to be used for low out-of-range values. Requires norm.clip = False

class LinearSegmentedColormap(*name, segmentdata, N=256*)
Bases: matplotlib.colors.Colormap

Colormap objects based on lookup tables using linear segments.

The lookup transfer function is a simple linear function between defined intensities. There is no limit to the number of segments that may be defined. Though as the segment intervals start containing fewer and fewer array locations, there will be inevitable quantization errors

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. See makeMappingArray for details

class 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 floating point array (*N* rgb 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 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*.

inverse(*value*)

class 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 Normalize(*vmin=None, vmax=None, clip=False*)

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*)

scaled()

return true if vmin and vmax set

hex2color(*s*)

Take a hex string *s* and return the corresponding rgb 3-tuple Example: #efefef -> (0.93725, 0.93725, 0.93725)

is_color_like(*c*)

makeMappingArray(*N, data*)

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.

The function returns an array "result" where `result[x*(N-1)]` gives the closest value for values of x between 0 and 1.

class no_norm(*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 normalize(*vmin=None, vmax=None, clip=False*)

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*)

scaled()

return true if vmin and vmax set

rgb2hex(*rgb*)

Given a len 3 rgb tuple of 0-1 floats, return the hex string

CHAPTER

THIRTY

Matplotlib pyplot

30.1 matplotlib.pyplot

acorr(**args, **kwargs*)

call signature:

```
acorr(x, normed=False, detrend=mlab.detrend_none, usevlines=False,
    maxlags=None, **kwargs)
```

Plot the autocorrelation of *x*. If *normed* = *True*, normalize the data but 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. The return value is a tuple (*lags*, *c*, *linecol*, *b*) where

- *linecol* is the `LineCollection`
- *b* is the *x*-axis.

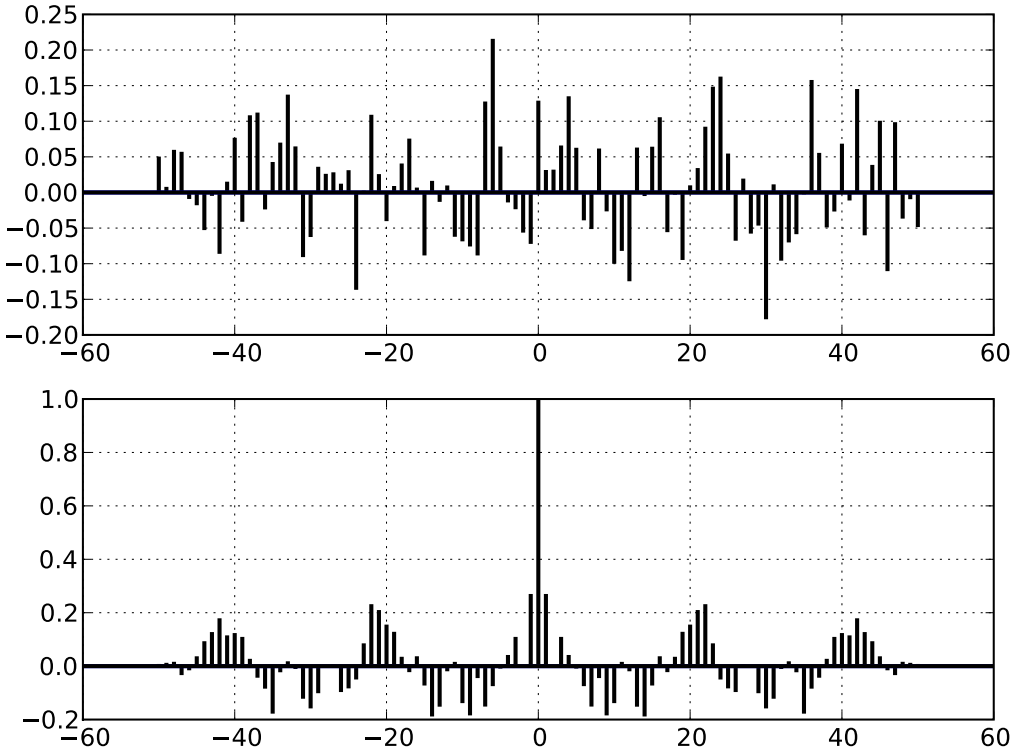
maxlags is a positive integer detailing the number of lags to show. The default value of *None* will return all `(2*len(x)-1)` lags.

See the respective `plot()` or `vlines()` functions for documentation on valid kwargs.

Example:

`xcorr()` above, and `acorr()` below.

Example:



Additional kwargs: hold = [True|False] overrides default hold state

annotate(**args, **kwargs*)

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. Valid keys 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`

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'
```

Additional kwargs are Text properties:

Property	Description
alpha	float
animated	[True | False]
axes	an axes instance
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
color	any matplotlib color
contains	unknown
family	['serif' | 'sans-serif' | 'cursive' | 'fantasy' | 'monospace']
figure	a `matplotlib.figure.Figure` instance
fontproperties	a matplotlib.font_manager.FontProperties instance
horizontalalignment or ha	['center' | 'right' | 'left']
label	any string
linespacing	float
lod	[True | False]
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	unknown
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

arrow(**args, **kwargs*)

call signature:

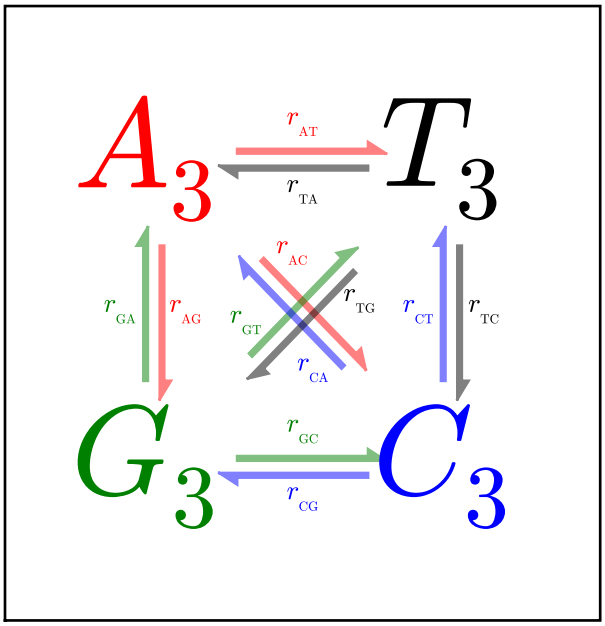
```
arrow(x, y, dx, dy, **kwargs)
```

Draws arrow on specified axis from (*x*, *y*) to (*x* + *dx*, *y* + *dy*).

Optional kwargs control the arrow properties:

Property	Description
aa	[True | False] or None for default
alpha	float
animated	[True | False]
antialiased	[True | False] or None for default
axes	an axes instance
clip_box	a `matplotlib.transform.Bbox` instance
clip_on	[True | False]
clip_path	a `Path` instance and a
contains	unknown
ec	mpl color spec, or None for default, or 'none' for no color
edgecolor	mpl color spec, or None for default, or 'none' for no color
facecolor	mpl color spec, or None for default, or 'none' for no color
fc	mpl color spec, or None for default, or 'none' for no color
figure	a `matplotlib.figure.Figure` instance
fill	[True | False]
hatch	unknown
label	any string
linestyle	['solid' | 'dashed' | 'dashdot' | 'dotted']
linewidth	float or None for default
lod	[True | False]
ls	['solid' | 'dashed' | 'dashdot' | 'dotted']
lw	float or None for default
picker	[None|float|boolean|callable]
transform	unknown
visible	[True | False]
zorder	any number

Example:



Additional kwargs: hold = [True|False] overrides default hold state

autumn()
Set the default colormap to *autumn* and apply to current image if any. See `colormaps()` for more information.

axes(**args, **kwargs*)
Add an axes 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*.

axhline(**args, **kwargs*)
call signature:

```
axhline(y=0, xmin=0, xmax=1, **kwargs)
```

Axis Horizontal Line

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:

Property	Description
alpha	float
animated	[True | False]
antialiased or aa	[True | False]
axes	unknown
clip_box	a matplotlib.transform.Bbox instance
clip_on	[True | False]
clip_path	a Path instance and a
color or c	any matplotlib color
contains	unknown
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' | 'steps-pre' | 'steps-mid' | 'steps-post' | 'None' | ' ' | '']
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
picker	unknown
pickradius	unknown
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

See axhspan() for example plot and source code

Additional kwargs: hold = [True|False] overrides default hold state

axhspan(**args, **kwargs*)

call signature:

```
axhspan(ymin, ymax, xmin=0, xmax=1, **kwargs)
```

Axis Horizontal Span.

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 span 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:

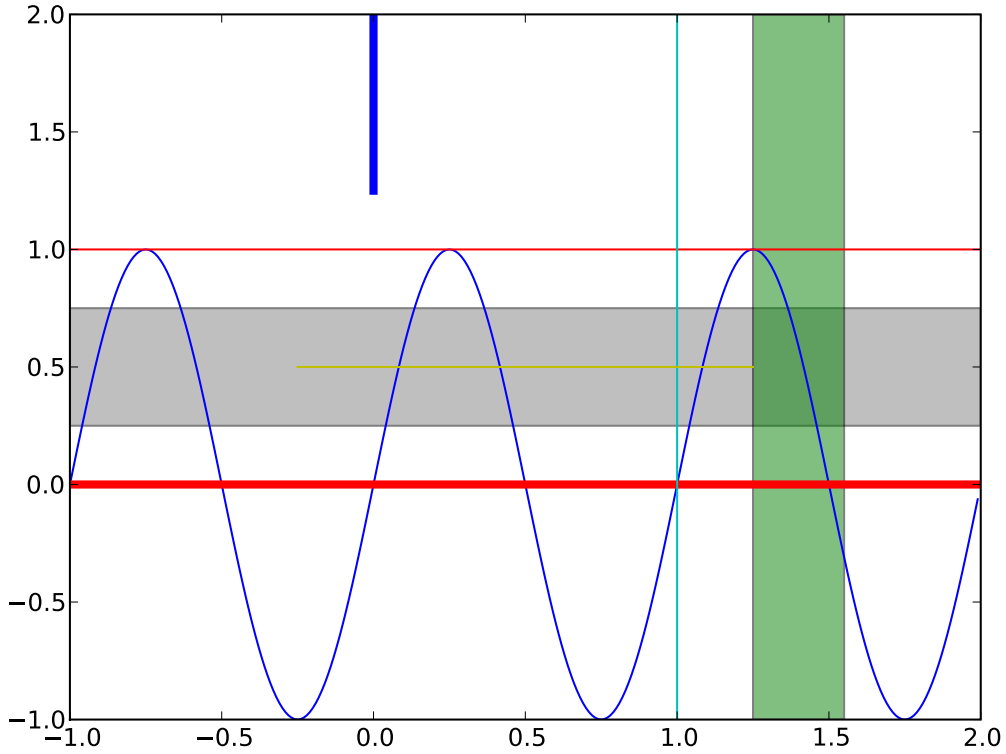
- draw a gray rectangle from *y* = 0.25-0.75 that spans the horizontal extent of the axes

```
>>> axhspan(0.25, 0.75, facecolor='0.5', alpha=0.5)
```

Valid kwargs are `Polygon` properties:

Property	Description
aa	[True | False] or None for default
alpha	float
animated	[True | False]
antialiased	[True | False] or None for default
axes	an axes instance
clip_box	a `matplotlib.transform.Bbox` instance
clip_on	[True | False]
clip_path	a `Path` instance and a
contains	unknown
ec	mpl color spec, or None for default, or 'none' for no color
edgecolor	mpl color spec, or None for default, or 'none' for no color
facecolor	mpl color spec, or None for default, or 'none' for no color
fc	mpl color spec, or None for default, or 'none' for no color
figure	a `matplotlib.figure.Figure` instance
fill	[True | False]
hatch	unknown
label	any string
linestyle	['solid' | 'dashed' | 'dashdot' | 'dotted']
linewidth	float or None for default
lod	[True | False]
ls	['solid' | 'dashed' | 'dashdot' | 'dotted']
lw	float or None for default
picker	[None|float|boolean|callable]
transform	unknown
visible	[True | False]
zorder	any number

Example:



Additional kwargs: hold = [True|False] overrides default hold state

axis(**v, **kwargs*)

Set/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. See `xlim()` and `ylim()` for more information

The xmin, xmax, ymin, ymax tuple is returned

axvline(**args, **kwargs*)

call signature:

```
axvline(x=0, ymin=0, ymax=1, **kwargs)
```

Axis Vertical Line

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 xlim settings, even if you change them, eg. with the `set_xlim()` 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:

Property	Description
alpha	float
animated	[True | False]
antialiased or aa	[True | False]
axes	unknown
clip_box	a `matplotlib.transform.Bbox` instance
clip_on	[True | False]
clip_path	a `Path` instance and a
color or c	any matplotlib color
contains	unknown
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' | 'steps-pre' | 'steps-mid' | 'steps-post' | 'None' | ' ' | '']
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
picker	unknown
pickradius	unknown
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

See `axhspan()` for example plot and source code

Additional kwargs: hold = [True|False] overrides default hold state

axvspan(**args, **kwargs*)

call signature:

```
axvspan(xmin, xmax, ymin=0, ymax=1, **kwargs)
```

Axis Vertical Span.

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 span 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
aa	[True | False] or None for default
alpha	float
animated	[True | False]
antialiased	[True | False] or None for default
axes	an axes instance
clip_box	a `matplotlib.transform.Bbox` instance
clip_on	[True | False]
clip_path	a `Path` instance and a
contains	unknown
ec	mpl color spec, or None for default, or 'none' for no color
edgecolor	mpl color spec, or None for default, or 'none' for no color
facecolor	mpl color spec, or None for default, or 'none' for no color
fc	mpl color spec, or None for default, or 'none' for no color
figure	a `matplotlib.figure.Figure` instance
fill	[True | False]
hatch	unknown
label	any string
linestyle	['solid' | 'dashed' | 'dashdot' | 'dotted']
linewidth	float or None for default
lod	[True | False]
ls	['solid' | 'dashed' | 'dashdot' | 'dotted']
lw	float or None for default
picker	[None|float|boolean|callable]
transform	unknown
visible	[True | False]
zorder	any number

See `axhspan()` for example plot and source code

Additional kwargs: hold = [True|False] overrides default hold state

bar(**args, **kwargs*)

call signature:

```
bar(left, height, width=0.8, bottom=0,
    color=None, edgecolor=None, linewidth=None,
    yerr=None, xerr=None, ecolor=None, capsize=3,
    align='edge', orientation='vertical', log=False)
```

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
align	'edge' (default) | 'center'
orientation	'vertical' | 'horizontal'
log	[False|True] False (default) leaves the orientation axis as-is; True sets it to log scale

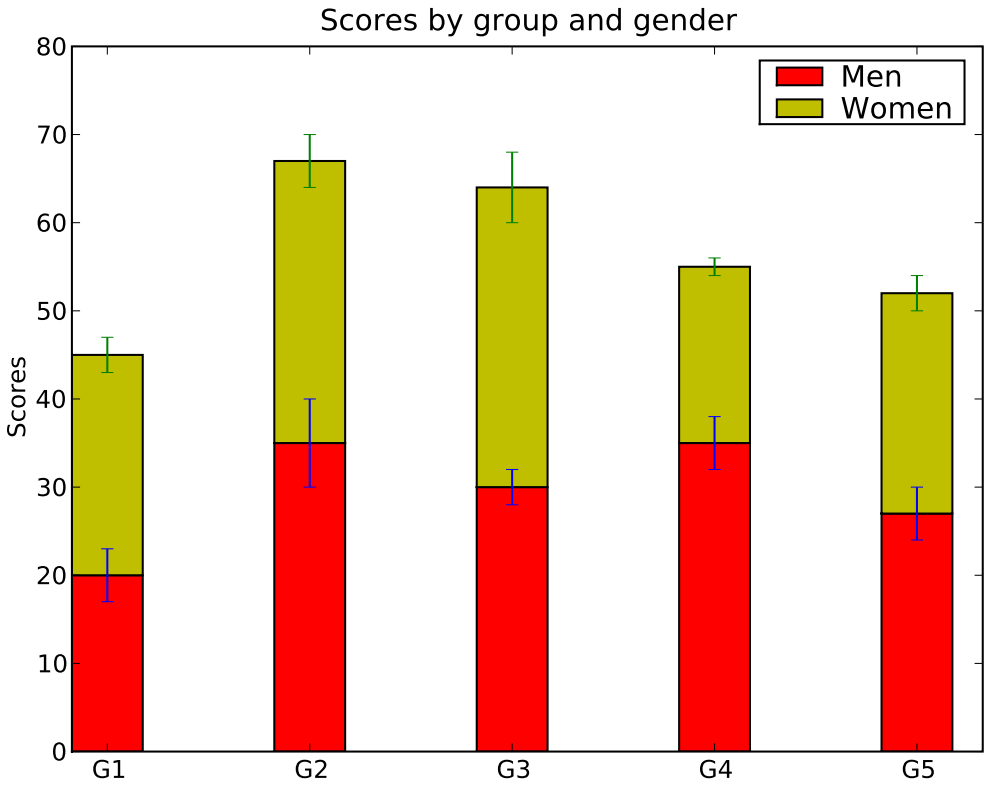
For vertical bars, *align* = 'edge' aligns bars by their left edges in left, while *align* = 'center' interprets these values as the *x* coordinates of the bar centers. For horizontal bars, *align* = 'edge' aligns bars by their bottom edges in bottom, while *align* = 'center' interprets these values 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.

Other optional kwargs:

Property	Description
aa	[True | False] or None for default
alpha	float
animated	[True | False]
antialiased	[True | False] or None for default
axes	an axes instance
clip_box	a `matplotlib.transform.Bbox` instance
clip_on	[True | False]
clip_path	a `Path` instance and a
contains	unknown
ec	mpl color spec, or None for default, or 'none' for no color
edgecolor	mpl color spec, or None for default, or 'none' for no color
facecolor	mpl color spec, or None for default, or 'none' for no color
fc	mpl color spec, or None for default, or 'none' for no color
figure	a `matplotlib.figure.Figure` instance
fill	[True | False]
hatch	unknown
label	any string
linestyle	['solid' | 'dashed' | 'dashdot' | 'dotted']
linewidth	float or None for default
lod	[True | False]
ls	['solid' | 'dashed' | 'dashdot' | 'dotted']
lw	float or None for default
picker	[None|float|boolean|callable]
transform	unknown
visible	[True | False]
zorder	any number

Example: A stacked bar chart.



Additional kwargs: hold = [True|False] overrides default hold state

barbs(**args, **kwargs*)

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:
Unexpected indentation.

'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.
Unexpected indentation.

'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
alpha	float
animated	[True | False]
antialiased	Boolean or sequence of booleans
antialiaseds	Boolean or sequence of booleans
array	unknown
axes	an axes instance
clim	a length 2 sequence of floats
clip_box	a `matplotlib.transform.Bbox` instance
clip_on	[True | False]
clip_path	a `Path` instance and a
cmap	a colormap
color	matplotlib color arg or sequence of rgba tuples
colorbar	unknown
contains	unknown
dashes	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
edgecolor	matplotlib color arg or sequence of rgba tuples
edgecolors	matplotlib color arg or sequence of rgba tuples
facecolor	matplotlib color arg or sequence of rgba tuples
facecolors	matplotlib color arg or sequence of rgba tuples
figure	a `matplotlib.figure.Figure` instance
label	any string
linestyle	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
linestyles	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
linewidth	float or sequence of floats
linewidths	float or sequence of floats
lod	[True | False]
lw	float or sequence of floats
norm	unknown
offsets	float or sequence of floats
picker	[None|float|boolean|callable]
pickradius	unknown
transform	unknown
visible	[True | False]
zorder	any number

Additional kwargs: hold = [True|False] overrides default hold state

barh(**args, **kwargs*)

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
aa	[True | False] or None for default
alpha	float
animated	[True | False]
antialiased	[True | False] or None for default
axes	an axes instance
clip_box	a `matplotlib.transform.Bbox` instance
clip_on	[True | False]
clip_path	a `Path` instance and a
contains	unknown
ec	mpl color spec, or None for default, or 'none' for no color
edgecolor	mpl color spec, or None for default, or 'none' for no color
facecolor	mpl color spec, or None for default, or 'none' for no color
fc	mpl color spec, or None for default, or 'none' for no color
figure	a `matplotlib.figure.Figure` instance
fill	[True | False]
hatch	unknown
label	any string
linestyle	['solid' | 'dashed' | 'dashdot' | 'dotted']
linewidth	float or None for default
lod	[True | False]
ls	['solid' | 'dashed' | 'dashdot' | 'dotted']
lw	float or None for default
picker	[None|float|boolean|callable]
transform	unknown
visible	[True | False]
zorder	any number

Additional kwargs: hold = [True|False] overrides default hold state

bone()

Set the default colormap to bone and apply to current image if any. See `colormaps()` for more information.

box(*on=None*)

Turn the axes box on or off according to *on*.

If *on* is *None*, toggle state.

boxplot(**args, **kwargs*)

call signature:

```
boxplot(x, notch=0, sym='+', vert=1, whis=1.5,
        positions=None, widths=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.

- *notch* = 0 (default) produces a rectangular box plot.
- *notch* = 1 will produce a notched box plot

sym (default 'b+') is the default symbol for flier points. Enter an empty string ('') if you don't want to show fliers.

- *vert* = 1 (default) makes the boxes vertical.
- *vert* = 0 makes horizontal boxes. This seems goofy, but that's how Matlab did it.

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.

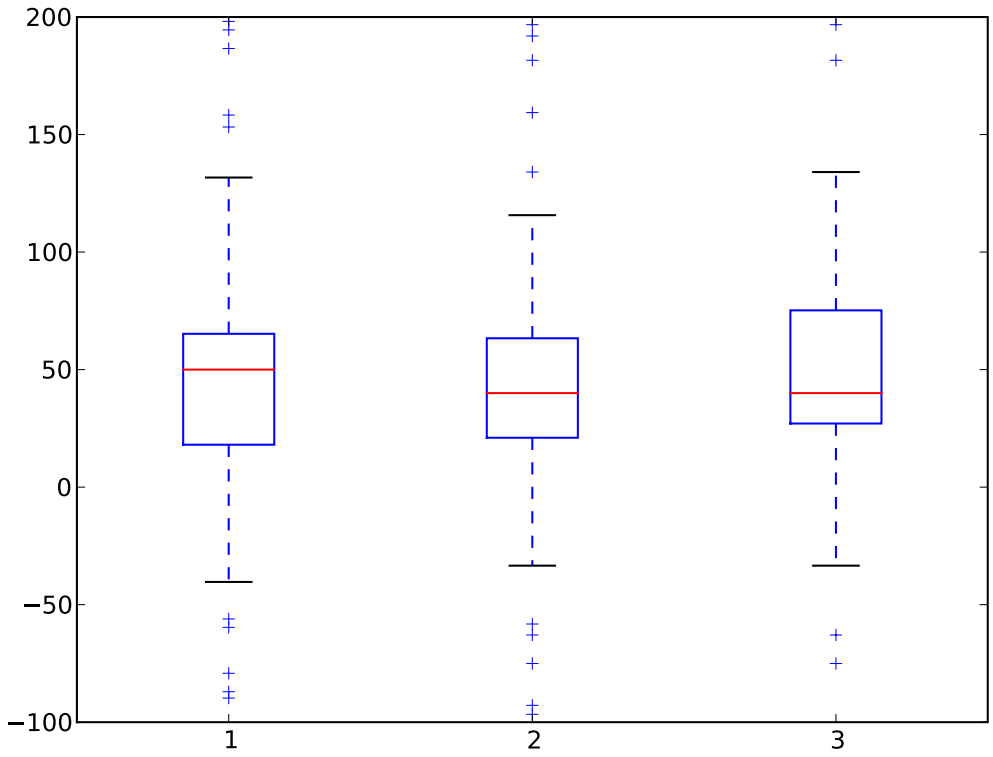
positions (default 1,2,...,n) sets the horizontal positions of the boxes. The ticks and limits are automatically set to match the positions.

widths is 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.

x is an array or a sequence of vectors.

Returns a list of the `matplotlib.lines.Line2D` instances added.

Example:



Additional kwargs: hold = [True|False] overrides default hold state

broken_barh(**args, **kwargs*)

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
alpha	float
animated	[True | False]
antialiased	Boolean or sequence of booleans
antialiaseds	Boolean or sequence of booleans
array	unknown
axes	an axes instance
clim	a length 2 sequence of floats
clip_box	a `matplotlib.transform.Bbox` instance
clip_on	[True | False]
clip_path	a `Path` instance and a
cmap	a colormap
color	matplotlib color arg or sequence of rgba tuples
colorbar	unknown
contains	unknown
dashes	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
edgecolor	matplotlib color arg or sequence of rgba tuples
edgecolors	matplotlib color arg or sequence of rgba tuples
facecolor	matplotlib color arg or sequence of rgba tuples
facecolors	matplotlib color arg or sequence of rgba tuples
figure	a `matplotlib.figure.Figure` instance
label	any string
linestyle	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
linestyles	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
linewidth	float or sequence of floats
linewidths	float or sequence of floats
lod	[True | False]
lw	float or sequence of floats
norm	unknown
offsets	float or sequence of floats
picker	[None|float|boolean|callable]
pickradius	unknown
transform	unknown
visible	[True | False]
zorder	any number

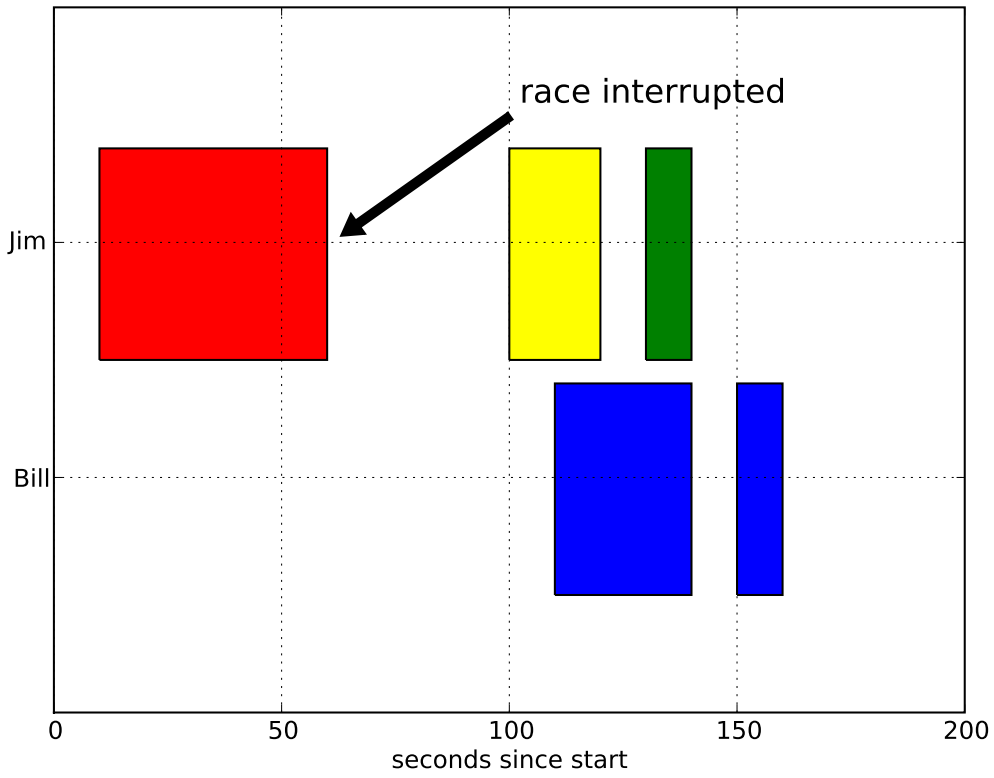
these can either be a single argument, ie:

```
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

cla(**args, **kwargs*)
Clear the current axes

clabel(**args, **kwargs*)
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*:** See http://matplotlib.sf.net/fonts.html

Additional kwargs: hold = [True|False] overrides default hold state

clf()
Clear the current figure

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)
```

close(**args*)
Close a figure window

`close()` by itself closes the current figure

`close(num)` closes figure number *num*

`close(h)` where *h* is a `Figure` instance, closes that figure

`close('all')` closes all the figure windows

cohere(**args, **kwargs*)
call signature:

```
cohere(x, y, NFFT=256, Fs=2, Fc=0, detrend = mlab.detrend_none,
       window = mlab.window_hanning, noverlap=0, **kwargs)
```

cohere the coherence between *x* and *y*. Coherence is the normalized cross spectral density:

$$C_{xy} = \frac{|P_{xy}|^2}{P_{xx} * P_{yy}} \tag{30.1}$$

The return value is a tuple (Cxy, f), where f are the frequencies of the coherence vector.

See the `psd()` for a description of the optional parameters.

kwargs are applied to the lines.

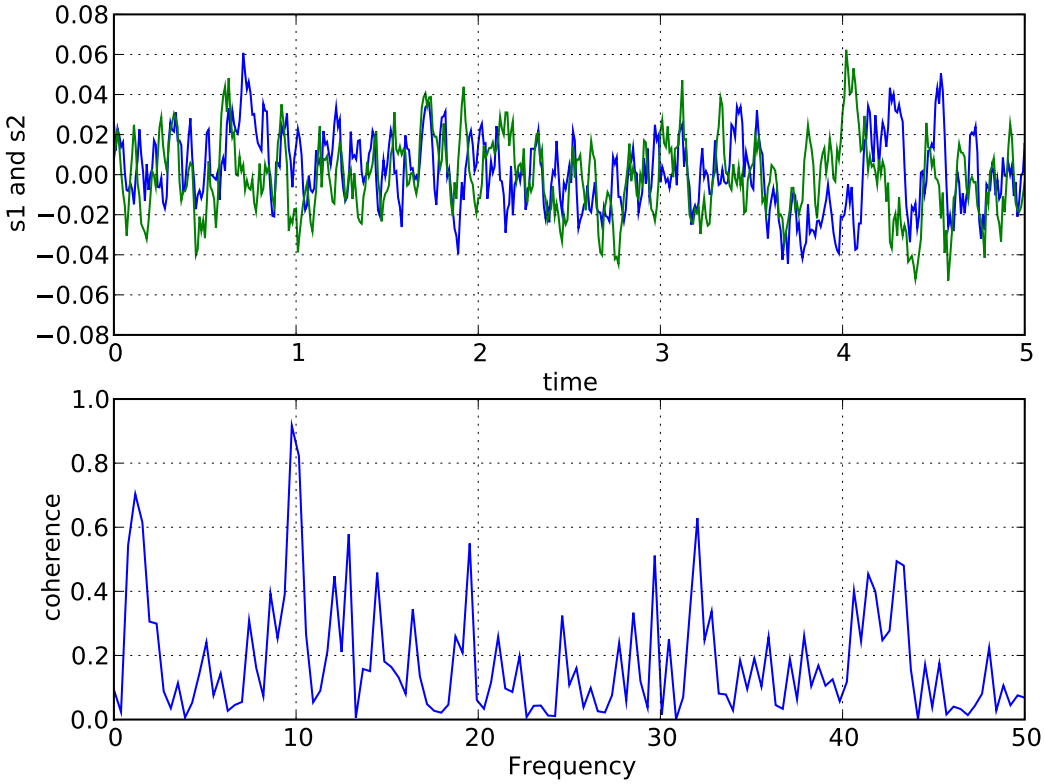
References:

- Bendat & Piersol – Random Data: Analysis and Measurement Procedures, John Wiley & Sons (1986)

kwargs control the `Line2D` properties of the coherence plot:

Property	Description
alpha	float
animated	[True | False]
antialiased or aa	[True | False]
axes	unknown
clip_box	a `matplotlib.transform.Bbox` instance
clip_on	[True | False]
clip_path	a `Path` instance and a
color or c	any matplotlib color
contains	unknown
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' | 'steps-pre' | 'steps-mid' | 'steps-post' | 'None' | ' ' | '']
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
picker	unknown
pickradius	unknown
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

Example:



Additional kwargs: hold = [True|False] overrides default hold state

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 `matplotlib.Figure.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 `matplotlib.Figure.colorbar()` method but optional for the `matplotlib.pyplot.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

Additional keyword arguments are of two kinds:

axes properties:

Property	Description
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

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.
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.

`colormaps()`

matplotlib provides the following colormaps.

- autumn
- bone
- cool
- copper
- flag
- gray

- hot
- hsv
- jet
- pink
- prism
- spring
- summer
- winter
- spectral

You can set the colormap for an image, pcolor, scatter, etc, either as a keyword argument:

```
imshow(X, cmap=cm.hot)
```

or post-hoc using the corresponding pylab interface function:

```
imshow(X)
hot()
jet()
```

In interactive mode, this will update the colormap allowing you to see which one works best for your data.

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')
```

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'

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)
```

contour(**args, **kwargs*)

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 (TM) version in that it does not draw the polygon edges, because the contouring engine yields simply connected regions with branch cuts. To draw the 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, *Y*, and *Z* must be arrays with the same dimensions.

Z may be a masked array, but filled contouring may not handle internal masked regions correctly.

`C = contour(...)` returns a `ContourSet` 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.

***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.cm.Colormap.set_under()` and `matplotlib.cm.Colormap.set_over()` methods.

contour-only keyword arguments:

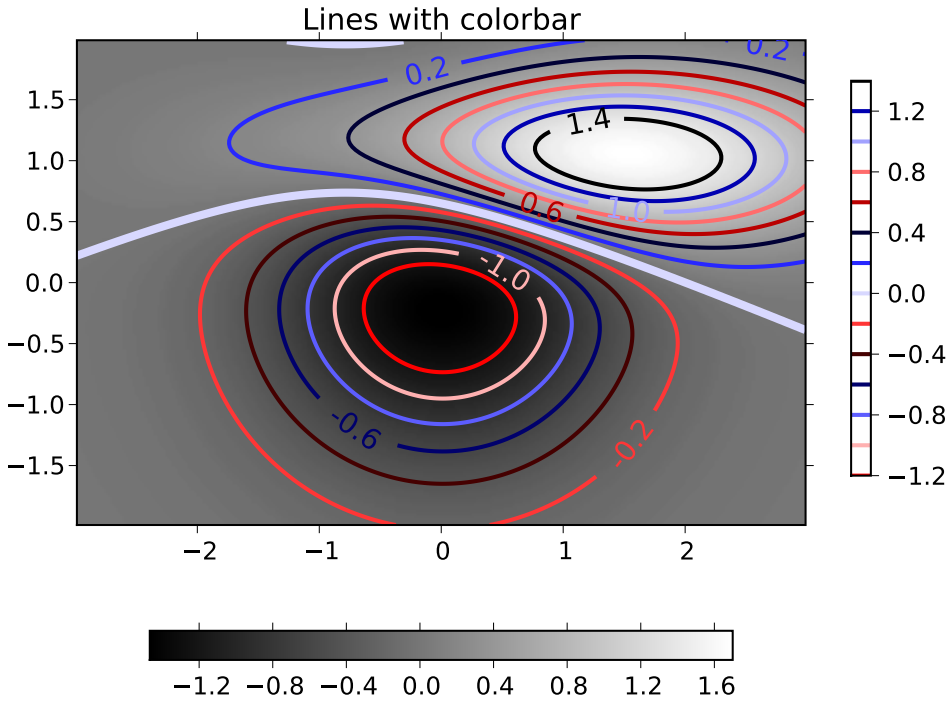
***linewidths*: [None | number | tuple of numbers]** If *linewidths* is *None*, the default width in `lines.linewidth` in `matplotlibrc` is used
If a number, all levels will be plotted with this linewidth.
If a tuple, different levels will be plotted with different linewidths in the order specified

contourf-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*.

Example:



Additional kwargs: hold = [True|False] overrides default hold state

contourf(**args, **kwargs*)

`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 (TM) version in that it does not draw the polygon edges, because the contouring engine yields simply connected regions with branch cuts. To draw the 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, *Y*, and *Z* must be arrays with the same dimensions.

Z may be a masked array, but filled contouring may not handle internal masked regions correctly.

`C = contour(...)` returns a `ContourSet` 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.

***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.cm.Colormap.set_under()` and `matplotlib.cm.Colormap.set_over()` methods.

contour-only keyword arguments:

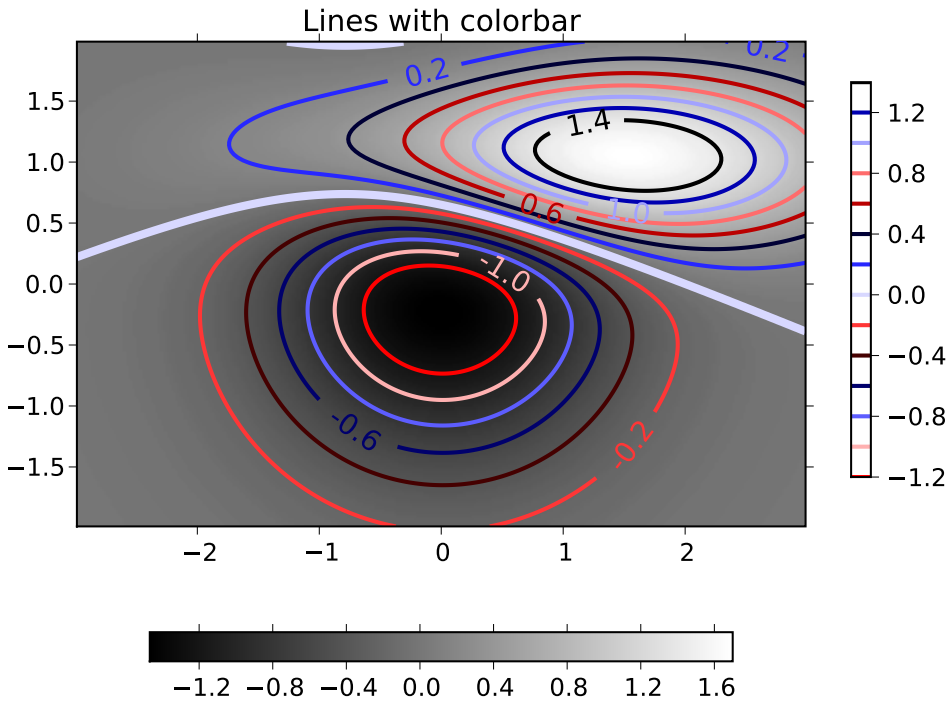
***linewidths*: [None | number | tuple of numbers]** If *linewidths* is *None*, the default width in `lines.linewidth` in `matplotlibrc` is used
If a number, all levels will be plotted with this linewidth.
If a tuple, different levels will be plotted with different linewidths in the order specified

contourf-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*.

Example:



Additional kwargs: hold = [True|False] overrides default hold state

`cool()`

Set the default colormap to cool and apply to current image if any. See `colormaps()` for more information.

copper()

Set the default colormap to copper and apply to current image if any. See colormaps() for more information.

csd(**args, **kwargs*)

call signature:

```
csd(x, y, NFFT=256, Fs=2, Fc=0, detrend=mlab.detrend_none,
    window=window_hanning, noverlap=0, **kwargs)
```

The cross spectral density P_{xy} by Welches 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.

See psd() for a description of the optional parameters.

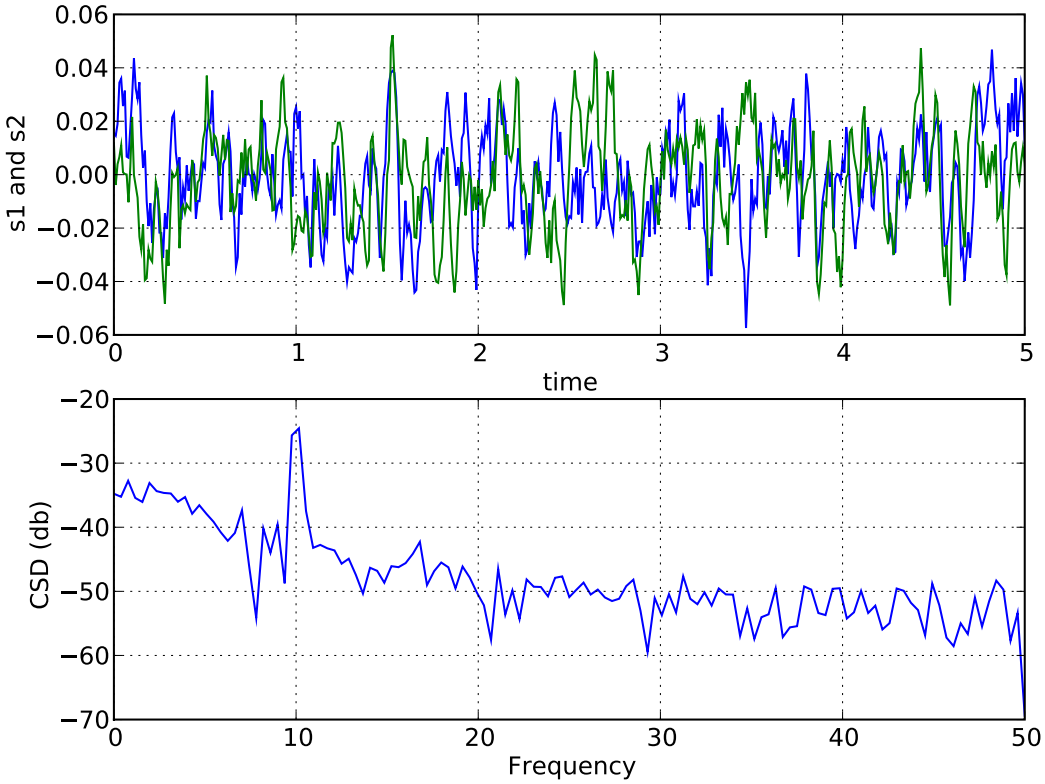
Returns the tuple (*Pxy*, *freqs*). *P* is the cross spectrum (complex valued), and $10\log_{10}|P_{xy}|$ is plotted.

References: Bendat & Piersol – Random Data: Analysis and Measurement Procedures, John Wiley & Sons (1986)

kwargs control the Line2D properties:

Property	Description
alpha	float
animated	[True | False]
antialiased or aa	[True | False]
axes	unknown
clip_box	a `matplotlib.transform.Bbox` instance
clip_on	[True | False]
clip_path	a `Path` instance and a
color or c	any matplotlib color
contains	unknown
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' | 'steps-pre' | 'steps-mid' | 'steps-post' | 'None' | ' ' | '']
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
picker	unknown
pickradius	unknown
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

Example:



Additional kwargs: hold = [True|False] overrides default hold state

delaxes(**args*)

`delaxes(ax)`: remove *ax* from the current figure. If *ax* doesn't exist, an error will be raised.

`delaxes()`: delete the current axes

disconnect(*cid*)

disconnect callback id cid

Example usage:

```
cid = canvas.mpl_connect('button_press_event', on_press)
#...later
canvas.mpl_disconnect(cid)
```

draw()

redraw the current figure

errorbar(**args, **kwargs*)

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)
```

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, Nx2 array-like]** If a scalar number, len(N) array-like object, or an Nx1 array-like object, errorbars are drawn +/- value.
If a rank-1, Nx2 Numpy array, errorbars are drawn at -column1 and +column2

***fmt*: '-'** The plot format symbol for *y*. If *fmt* is *None*, just plot the errorbars with no line symbols. This can be useful for creating a bar plot with errorbars.

***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 size of the error bar caps in points

***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*.

All other keyword arguments are passed on to the plot command for the markers, so you can add additional key=value pairs to control the errorbar 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)
```

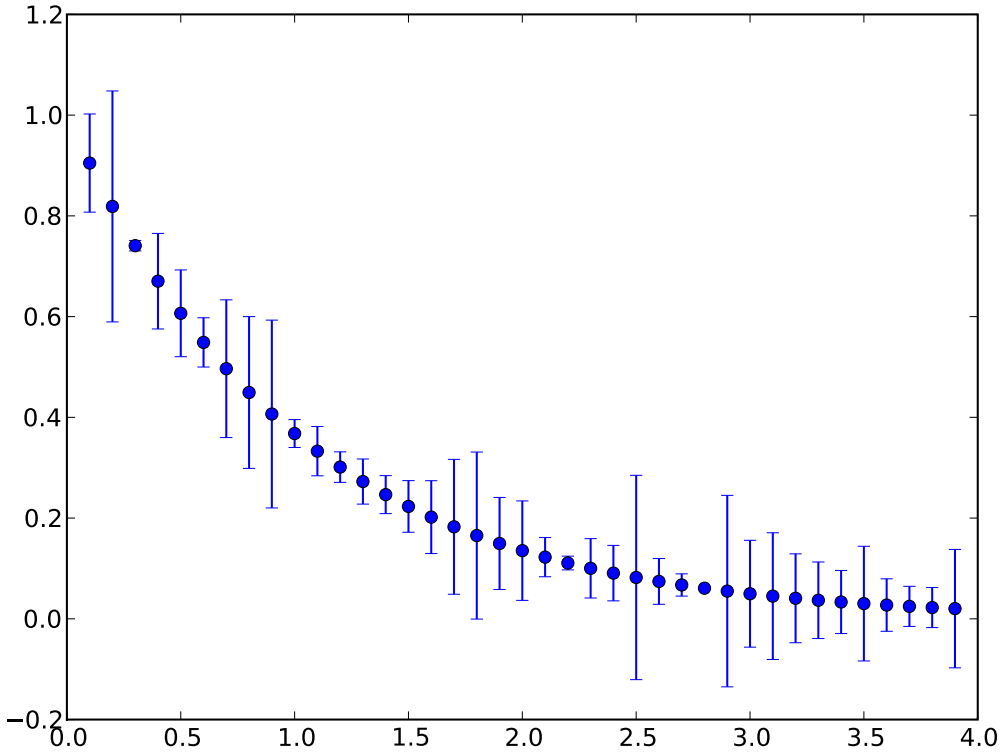
where *mfc*, *mec*, *ms* and *mew* are aliases for the longer property names, *markerfacecolor*, *markeredgecolor*, *markersize* and *markeredgewith*.

valid kwargs for the marker properties are

Property	Description
alpha	float
animated	[True | False]
antialiased or aa	[True | False]
axes	unknown
clip_box	a `matplotlib.transform.Bbox` instance
clip_on	[True | False]
clip_path	a `Path` instance and a
color or c	any matplotlib color
contains	unknown
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' | 'steps-pre' | 'steps-mid' | 'steps-post' | 'None' | ' ' | '']
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
picker	unknown
pickradius	unknown
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

Return value is a length 3 tuple. The first element is the `Line2D` instance for the *y* symbol lines. The second element is a list of error bar cap lines, the third element is a list of `LineCollection` instances for the horizontal and vertical error ranges.

Example:



Additional kwargs: hold = [True|False] overrides default hold state

figimage(**args, **kwargs*)

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:

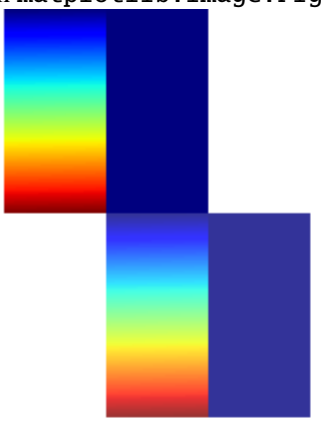
- If *X* is MxN, assume luminance (grayscale)
- If *X* is MxNx3, assume RGB
- If *X* is MxNx4, assume RGBA

Optional keyword arguments:

Key-word	Description
xo or yo	An integer, the *x* and *y* image offset in pixels
cmap	a `matplotlib.cm.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 1.0
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.



Addition kwargs: hold = [True|False] overrides default hold state

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

Example:

```
figlegend( (line1, line2, line3),
           ('label1', 'label2', 'label3'),
           'upper right' )
```

See legend() for information about the location codes

A matplotlib.legend.Legend instance is returned.

figtext(**args, **kwargs*)

Call signature:

```
figtext(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
alpha	float
animated	[True | False]
axes	an axes instance
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
color	any matplotlib color
contains	unknown
family	['serif' | 'sans-serif' | 'cursive' | 'fantasy' | 'monospace']
figure	a `matplotlib.figure.Figure` instance
fontproperties	a matplotlib.font_manager.FontProperties instance
horizontalalignment or ha	['center' | 'right' | 'left']
label	any string
linespacing	float
lod	[True | False]
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	unknown
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

figure(*num=None, figsize=None, dpi=None, facecolor=None, edgecolor=None, frameon=True, FigureClass=<class 'matplotlib.figure.Figure'>, **kwargs*)

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 the handle to it. If `figure(num)` does not exist it will be created. Numbering starts at 1, matlab style:

```
figure(1)
```

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.

fill(**args, **kwargs*)

call signature:

```
fill(*args, **kwargs)
```

Plot filled polygons. *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 `poly_between()`, eg.:

```
xs, ys = poly_between(x, 0, y)
axes.fill(xs, ys, facecolor='red', alpha=0.5)
```

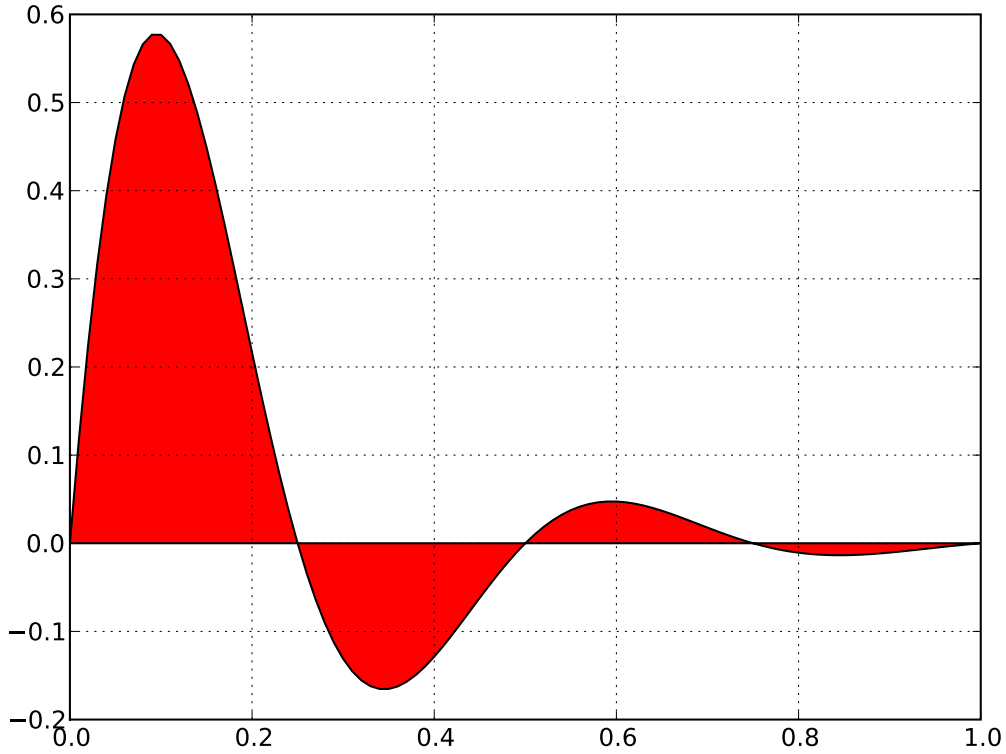
See `examples/pylab_examples/fill_between.py` for more examples.

The *closed* kwarg will close the polygon when *True* (default).

kwargs control the Polygon properties:

Property	Description
aa	[True | False] or None for default
alpha	float
animated	[True | False]
antialiased	[True | False] or None for default
axes	an axes instance
clip_box	a `matplotlib.transform.Bbox` instance
clip_on	[True | False]
clip_path	a `Path` instance and a
contains	unknown
ec	mpl color spec, or None for default, or 'none' for no color
edgecolor	mpl color spec, or None for default, or 'none' for no color
facecolor	mpl color spec, or None for default, or 'none' for no color
fc	mpl color spec, or None for default, or 'none' for no color
figure	a `matplotlib.figure.Figure` instance
fill	[True | False]
hatch	unknown
label	any string
linestyle	['solid' | 'dashed' | 'dashdot' | 'dotted']
linewidth	float or None for default
lod	[True | False]
ls	['solid' | 'dashed' | 'dashdot' | 'dotted']
lw	float or None for default
picker	[None|float|boolean|callable]
transform	unknown
visible	[True | False]
zorder	any number

Example:



Additional kwargs: hold = [True|False] overrides default hold state

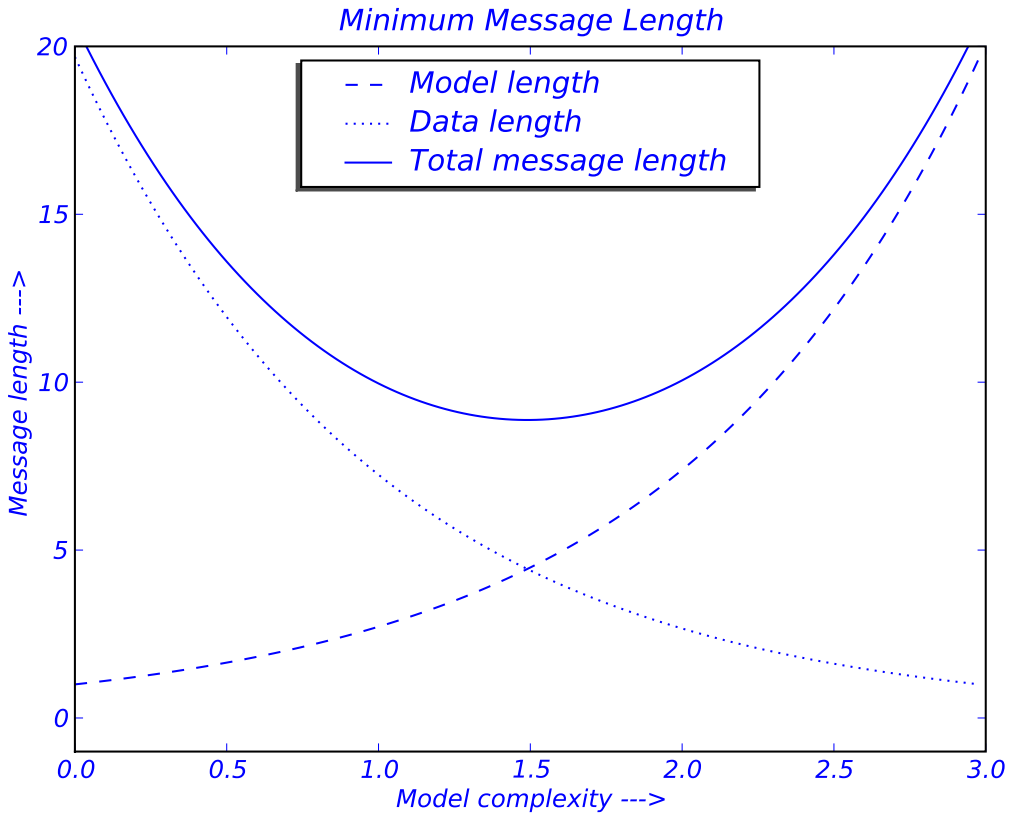
findobj(*o=None, match=None*) **pyplot signatur**

findobj(o=gcf(), match=None)

recursively find all :class:matplotlib.artist.Artist instances contained in self

match can be

- None: return all objects contained in artist (including artist)
- function with signature `boolean = match(artist)` used to filter matches
- class instance: eg Line2D. Only return artists of class type



flag()
: Set the default colormap to flag and apply to current image if any. See `colormaps()` for more information.

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])
```

gcf()
: Return a handle to the current figure.

gci()
: Get 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.

get_current_fig_manager()

get_plot_commands()

ginput(**args, **kwargs*)
call signature:

```
ginput(self, n=1, timeout=30, show_clicks=True)
```

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 negative, does not timeout.

If *n* is negative, accumulate clicks until a middle click terminates the input.

Right clicking cancels last input.

gray()
Set the default colormap to gray and apply to current image if any. See `colormaps()` for more information.

grid(**args, **kwargs*)
call signature:

```
grid(self, b=None, **kwargs)
```

Set the axes grids on or off; *b* is a boolean

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*

kawrgs are used to set the grid line properties, eg:

```
ax.grid(color='r', linestyle='-', linewidth=2)
```

Valid `Line2D` kwargs are

Property	Description
alpha	float
animated	[True | False]
antialiased or aa	[True | False]
axes	unknown
clip_box	a `matplotlib.transform.Bbox` instance
clip_on	[True | False]
clip_path	a `Path` instance and a
color or c	any matplotlib color
contains	unknown
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' | 'steps-pre' | 'steps-mid' | 'steps-post' | 'None' | ' ' | '']
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
picker	unknown
pickradius	unknown
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

hexbin(**args, **kwargs*)

call signature:

```
hexbin(x, y, C = None, gridsize = 100, bins = None,
       xscale = 'linear', yscale = 'linear',
       cmap=None, norm=None, vmin=None, vmax=None,
       alpha=1.0, linewidths=None, edgecolors='none'
       reduce_C_function = np.mean,
       **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.

Other keyword arguments controlling color mapping and normalization arguments:

***cmap*: [None | Colormap]** a `matplotlib.cm.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** 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:

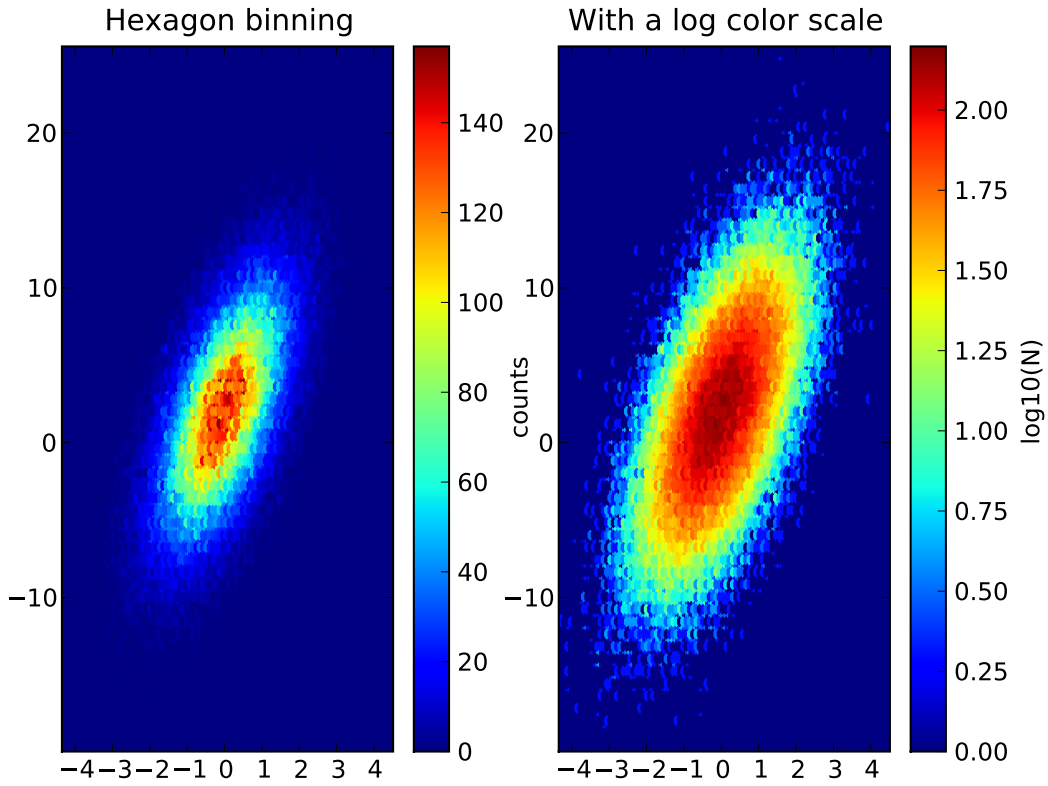
***edgecolors*: [None | mpl color | color sequence]** If 'none', draws the edges in the same color as the fill color. This is the default, as it avoids unsightly unpainted pixels between the hexagons.
If *None*, draws the outlines in the default color.
If a matplotlib color arg or sequence of rgba tuples, draws the outlines in the specified color.

Here are the standard descriptions of all the `Collection` kwargs:

Property	Description
alpha	float
animated	[True | False]
antialiased	Boolean or sequence of booleans
antialiaseds	Boolean or sequence of booleans
array	unknown
axes	an axes instance
clim	a length 2 sequence of floats
clip_box	a `matplotlib.transform.Bbox` instance
clip_on	[True | False]
clip_path	a `Path` instance and a
cmap	a colormap
color	matplotlib color arg or sequence of rgba tuples
colorbar	unknown
contains	unknown
dashes	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
edgecolor	matplotlib color arg or sequence of rgba tuples
edgecolors	matplotlib color arg or sequence of rgba tuples
facecolor	matplotlib color arg or sequence of rgba tuples
facecolors	matplotlib color arg or sequence of rgba tuples
figure	a `matplotlib.figure.Figure` instance
label	any string
linestyle	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
linestyles	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
linewidth	float or sequence of floats
linewidths	float or sequence of floats
lod	[True | False]
lw	float or sequence of floats
norm	unknown
offsets	float or sequence of floats
picker	[None|float|boolean|callable]
pickradius	unknown
transform	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.

Example:



Additional kwargs: hold = [True|False] overrides default hold state

hist(**args, **kwargs*)

call signature:

```
hist(x, bins=10, range=None, normed=False, cumulative=False,
     bottom=None, histtype='bar', align='mid',
     orientation='vertical', rwidth=None, log=False, **kwargs)
```

Compute 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.

Keyword arguments:

***bins*:** either an integer number of bins or a sequence giving the bins. *x* are the data to be binned. *x* can be an array or a 2D array with multiple data in its columns. Note, if *bins* is an integer input argument=numbins, *bins* + 1 bin edges will be returned, compatible with the semantics of `numpy.histogram()` with the *new* = True argument. 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.

***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))
```

***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 one. 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
- **'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 this 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'.

***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.

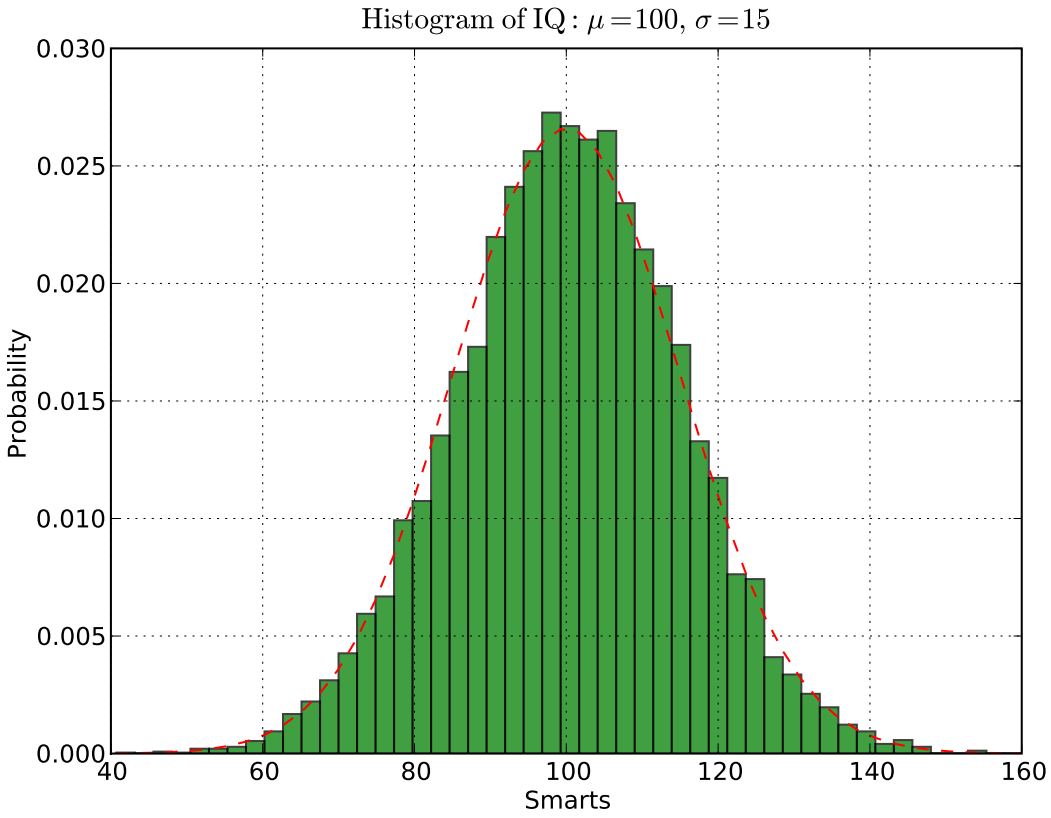
kwargs are used to update the properties of the hist `Rectangle` instances:

Property	Description
aa	[True | False] or None for default
alpha	float
animated	[True | False]
antialiased	[True | False] or None for default
axes	an axes instance
clip_box	a `matplotlib.transform.Bbox` instance
clip_on	[True | False]
clip_path	a `Path` instance and a
contains	unknown
ec	mpl color spec, or None for default, or 'none' for no color
edgecolor	mpl color spec, or None for default, or 'none' for no color
facecolor	mpl color spec, or None for default, or 'none' for no color
fc	mpl color spec, or None for default, or 'none' for no color
figure	a `matplotlib.figure.Figure` instance
fill	[True | False]
hatch	unknown
label	any string
linestyle	['solid' | 'dashed' | 'dashdot' | 'dotted']
linewidth	float or None for default
lod	[True | False]
ls	['solid' | 'dashed' | 'dashdot' | 'dotted']
lw	float or None for default
picker	[None|float|boolean|callable]
transform	unknown
visible	[True | False]
zorder	any number

You can use labels for your histogram, and only the first `Rectangle` gets the label (the others get the magic string '_nolegend_'. This will make the histograms work in the intuitive way for bar charts:

```
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()
```

Example:



Additional kwargs: hold = [True|False] overrides default hold state

hlines(**args, **kwargs*)

call signature:

```
hlines(y, xmin, xmax, colors='k', linestyle='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.

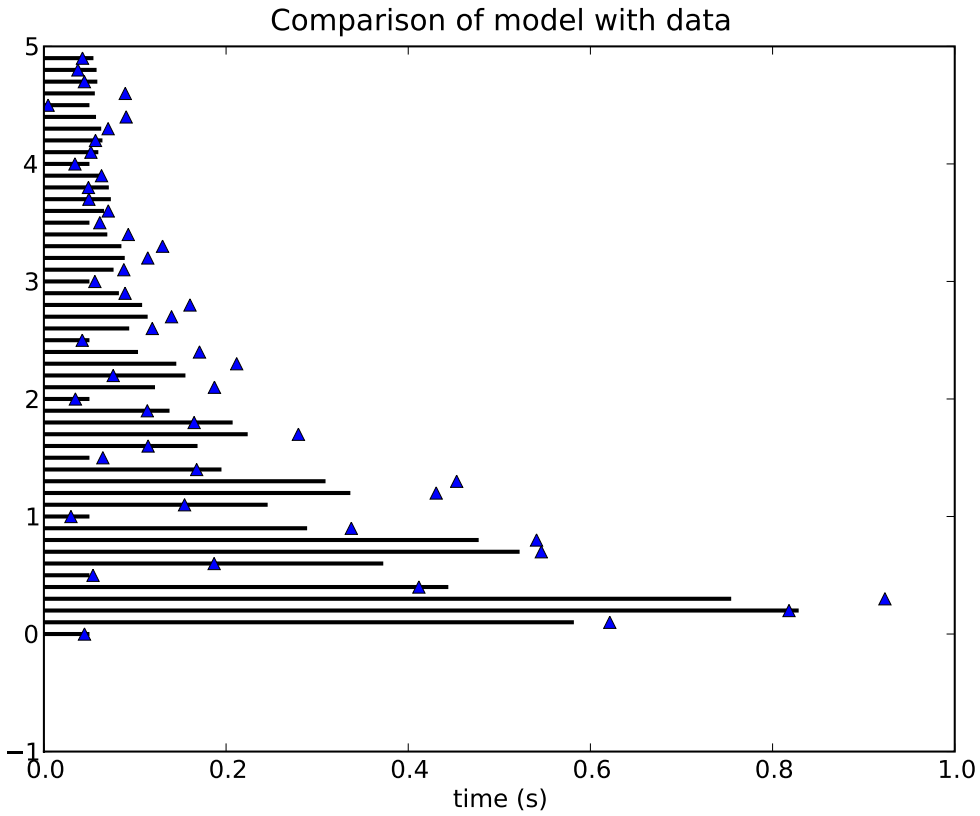
***xmin* and *xmax*:** can be scalars or `len(x)` numpy arrays. If they are scalars, then the respective values are constant, else the widths of the lines are determined by *xmin* and *xmax*.

Optional keyword arguments:

***colors*:** a line collections color argument, either a single color or a `len(y)` list of colors

***linestyle*:** ['solid' | 'dashed' | 'dashdot' | 'dotted']

Example:



Additional kwargs: hold = [True|False] overrides default hold state

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.

hot()

Set the default colormap to hot and apply to current image if any. See `colormaps()` for more information.

hsv()

Set the default colormap to hsv and apply to current image if any. See `colormaps()` for more information.

imread(**args, **kwargs*)

Return image file in *fname* as `numpy.array`.

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()`.

TODO: support RGB and grayscale return values in _image.readpng

imshow(**args, **kwargs*)

call signature:

```
imshow(X, cmap=None, norm=None, aspect=None, interpolation=None,
       alpha=1.0, 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.cm.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*, 'nearest', 'bilinear', 'bicubic', 'spline16', 'spline36', 'hanning', 'hamming', 'hermite', 'kaiser', 'quadric', 'catrom', 'gaussian', 'bessel', 'mitchell', 'sinc', 'lanczos', 'blackman'
If *interpolation* is *None*, default to rc `image.interpolation`. See also the *filternorm* and *filterrad* parameters

***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)

***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)]** Eata values of 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

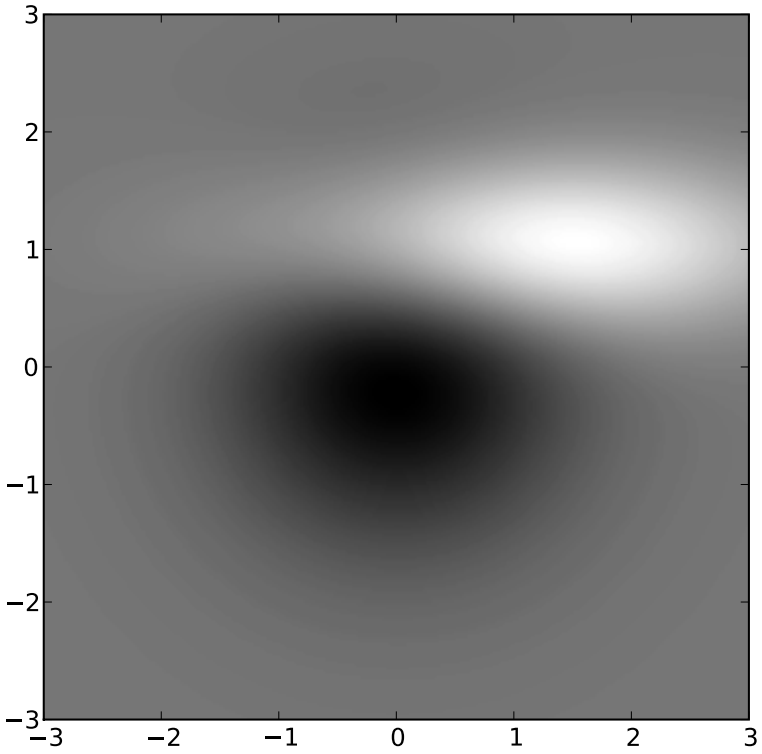
***filternorm*:** A parameter for the antigrain image resize filter. From the antigrain documentation, if *filternorm* = 1, the filter normalizes integer values and corrects the rounding errors. It doesn't do anything with the source floating point values, it corrects only integers according to the rule 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
alpha	float
animated	[True | False]
axes	an axes instance
clip_box	a `matplotlib.transform.Bbox` instance
clip_on	[True | False]
clip_path	a `Path` instance and a
contains	unknown
figure	a `matplotlib.figure.Figure` instance
label	any string
lod	[True | False]
picker	[None|float|boolean|callable]
transform	unknown
visible	[True | False]
zorder	any number

Example:



Additional kwargs: hold = [True|False] overrides default hold state

ioff()
Turn interactive mode off.

ion()
Turn interactive mode on.

ishold()
Return the hold status of the current axes

isinteractive()
Return the interactive status

jet()
Set the default colormap to jet and apply to current image if any. See `colormaps()` for more information.

legend(**args, **kwargs*)
call signature:

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

Place a legend on the current axes 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

If none of these are locations are suitable, loc can be a 2-tuple giving x,y in axes coords, ie:

```
loc = 0, 1 # left top
loc = 0.5, 0.5 # center
```

Keyword arguments:

***isaxes*: [True | False]** Indicates that this is an axes legend

***numpoints*: integer** The number of points in the legend line, default is 4

***prop*: [None | FontProperties]** A `matplotlib.font_manager.FontProperties` instance, or *None* to use rc settings.

***pad*: [None | scalar]** The fractional whitespace inside the legend border, between 0 and 1. If *None*, use rc settings.

***markerscale*: [None | scalar]** The relative size of legend markers vs. original. If *None*, use rc settings.

***shadow*: [None | False | True]** If *True*, draw a shadow behind legend. If *None*, use rc settings.

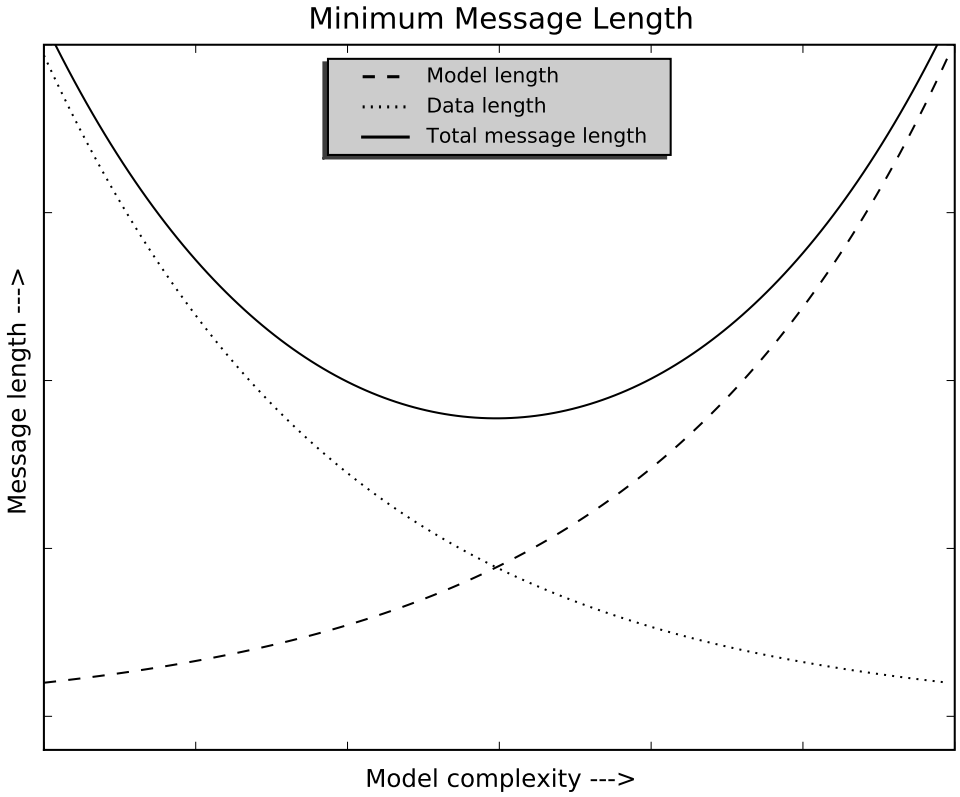
***labelsep*: [None | scalar]** The vertical space between the legend entries. If *None*, use rc settings.

***handlelen*: [None | scalar]** The length of the legend lines. If *None*, use rc settings.

***handletextsep*: [None | scalar]** The space between the legend line and legend text. If *None*, use rc settings.

***axespad*: [None | scalar]** The border between the axes and legend edge. If *None*, use rc settings.

Example:



loglog(**args, **kwargs*)

call signature:

```
loglog(*args, **kwargs)
```

Make a plot with log scaling on the *x* and *y* axis.

loglog() supports all the keyword arguments of plot() and matplotlib.axes.Axes.set_xscale()/matplotlib.axes.Axes.set_yscale().

Notable keyword arguments:

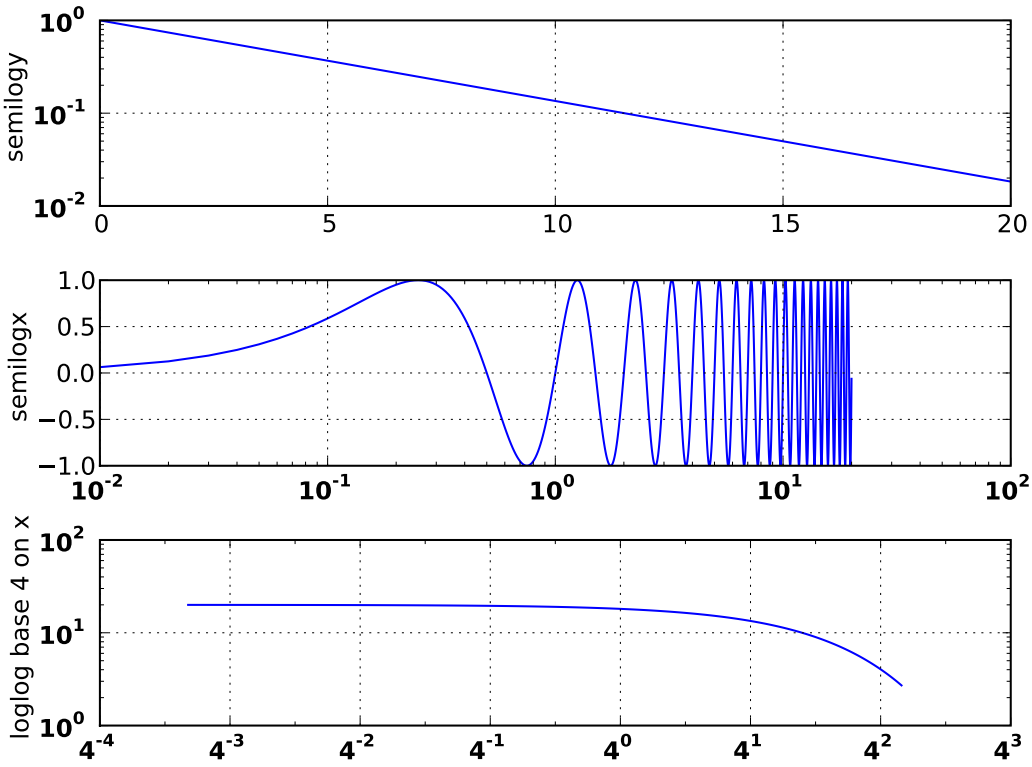
***basex/basey*: scalar > 1** base of the *x*/*y* logarithm

***subsx/subsy*: [None | sequence]** the location of the minor *x*/*y* ticks; *None* defaults to autosubs, which depend on the number of decades in the plot; see matplotlib.axes.Axes.set_xscale()/matplotlib.axes.Axes.set_yscale() for details

The remaining valid kwargs are Line2D properties:

Property	Description
alpha	float
animated	[True | False]
antialiased or aa	[True | False]
axes	unknown
clip_box	a `matplotlib.transform.Bbox` instance
clip_on	[True | False]
clip_path	a `Path` instance and a
color or c	any matplotlib color
contains	unknown
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' | 'steps-pre' | 'steps-mid' | 'steps-post' | 'None' | ' ' | '']
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
picker	unknown
pickradius	unknown
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

Example:



Additional kwargs: hold = [True|False] overrides default hold state

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()`.

***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.

over(*func, *args, **kwargs*)

over calls:

```
func(*args, **kwargs)
```

with `hold(True)` and then restores the hold state.

pcolor(**args, **kwargs*)

call signatures:

```
pcolor(C, **kwargs)
pcolor(X, Y, C, **kwargs)
```

Create a pseudocolor plot of a 2-D array.

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.cm.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 are *None*, the min and max of the color array *C* is used. If you pass a *norm* instance, *vmin* and *vmax* will be ignored.

***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(TM).

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** the alpha blending value

Return value is a `matplotlib.collection.Collection` instance.

The grid orientation follows the Matlab(TM) 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
alpha	float
animated	[True | False]
antialiased	Boolean or sequence of booleans
antialiaseds	Boolean or sequence of booleans
array	unknown
axes	an axes instance
clim	a length 2 sequence of floats
clip_box	a `matplotlib.transform.Bbox` instance
clip_on	[True | False]
clip_path	a `Path` instance and a
cmap	a colormap
color	matplotlib color arg or sequence of rgba tuples
colorbar	unknown
contains	unknown
dashes	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
edgecolor	matplotlib color arg or sequence of rgba tuples
edgecolors	matplotlib color arg or sequence of rgba tuples
facecolor	matplotlib color arg or sequence of rgba tuples
facecolors	matplotlib color arg or sequence of rgba tuples
figure	a `matplotlib.figure.Figure` instance
label	any string
linestyle	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
linestyles	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
linewidth	float or sequence of floats
linewidths	float or sequence of floats
lod	[True | False]
lw	float or sequence of floats
norm	unknown
offsets	float or sequence of floats
picker	[None|float|boolean|callable]
pickradius	unknown
transform	unknown
visible	[True | False]
zorder	any number

Additional kwargs: hold = [True|False] overrides default hold state

pcolormesh(**args, **kwargs*)

call signatures:

```
pcolormesh(C)
pcolormesh(X, Y, C)
pcolormesh(C, **kwargs)
```

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.cm.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 are *None*, the min and max of the color array *C* is used. If you pass a *norm* instance, *vmin* and *vmax* will be ignored.

***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(TM).

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** the alpha blending value

Return value is a `matplotlib.collection.QuadMesh` object.

See `pcolor()` for an explanation of the grid orientation and the expansion of 1-D *X* and/or *Y* to 2-D arrays.

kwargs can be used to control the `matplotlib.collections.QuadMesh` properties:

Property	Description
alpha	float
animated	[True | False]
antialiased	Boolean or sequence of booleans
antialiaseds	Boolean or sequence of booleans
array	unknown
axes	an axes instance
clim	a length 2 sequence of floats
clip_box	a `matplotlib.transform.Bbox` instance
clip_on	[True | False]
clip_path	a `Path` instance and a
cmap	a colormap
color	matplotlib color arg or sequence of rgba tuples
colorbar	unknown
contains	unknown
dashes	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
edgecolor	matplotlib color arg or sequence of rgba tuples
edgecolors	matplotlib color arg or sequence of rgba tuples
facecolor	matplotlib color arg or sequence of rgba tuples
facecolors	matplotlib color arg or sequence of rgba tuples
figure	a `matplotlib.figure.Figure` instance
label	any string
linestyle	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
linestyles	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
linewidth	float or sequence of floats
linewidths	float or sequence of floats
lod	[True | False]
lw	float or sequence of floats
norm	unknown
offsets	float or sequence of floats
picker	[None|float|boolean|callable]
pickradius	unknown
transform	unknown
visible	[True | False]
zorder	any number

Additional kwargs: hold = [True|False] overrides default hold state

pie(**args, **kwargs*)

call signature:

```
pie(x, explode=None, labels=None,
    colors=('b', 'g', 'r', 'c', 'm', 'y', 'k', 'w'),
    autopct=None, pctdistance=0.6, labeldistance=1.1, shadow=False)
```

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.

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.

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

pink()

Set the default colormap to pink and apply to current image if any. See `colormaps()` for more information.

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 the 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.

The following line styles are supported:

```
-     # solid line
--    # dashed line
-.    # dash-dot line
:     # dotted line
.     # points
,     # pixels
o     # circle symbols
^     # triangle up symbols
v     # triangle down symbols
<     # triangle left symbols
>     # triangle right symbols
s     # square symbols
+     # plus symbols
x     # cross symbols
D     # diamond symbols
d     # thin diamond symbols
1     # tripod down symbols
2     # tripod up symbols
3     # tripod left symbols
4     # tripod right symbols
h     # hexagon symbols
H     # rotated hexagon symbols
p     # pentagon symbols
|     # vertical line symbols
_     # horizontal line symbols
steps # use gnuplot style 'steps' # kwarg only
```

The following color abbreviations are supported:

```
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.

The kwargs are Line2D properties:

Property	Description
alpha	float
animated	[True | False]
antialiased or aa	[True | False]
axes	unknown
clip_box	a matplotlib.transform.Bbox instance
clip_on	[True | False]
clip_path	a Path instance and a
color or c	any matplotlib color
contains	unknown
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' | 'steps-pre' | 'steps-mid' | 'steps-post' | 'None' | ' ' | '']
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
picker	unknown
pickradius	unknown
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

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

plot_date(**args, **kwargs*)

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.

See dates for helper functions date2num(), num2date() and drange() for help on creating the required floating point dates.

Keyword arguments:

***fmt*: string** The plot format string.

***tz*: [None | timezone string]** 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.ticker.AutoDateLocator (if the tick locator is not already set to a matplotlib.ticker.DateLocator instance) and the default tick formatter to matplotlib.ticker.AutoDateFormatter (if the tick formatter is not already set to a matplotlib.ticker.DateFormatter instance).

Valid kwargs are Line2D properties:

Property	Description
alpha	float
animated	[True | False]
antialiased or aa	[True | False]
axes	unknown
clip_box	a `matplotlib.transform.Bbox` instance
clip_on	[True | False]
clip_path	a `Path` instance and a
color or c	any matplotlib color
contains	unknown
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' | 'steps-pre' | 'steps-mid' | 'steps-post' | 'None' | ' ' | '']
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
picker	unknown
pickradius	unknown
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

Additional kwargs: hold = [True|False] overrides default hold state

plotfile(*fname, cols=(0,), plotfuncs=None, comments='#', skiprows=0, checkrows=5, delimiter=', ', **kwargs*)

Plot the data in *fname*

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

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.

comments, *skiprows*, *checkrows*, and *delimiter* are all passed on to `matplotlib.pylab.csv2rec()` to load the data into a record array.

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'})
```

plotting()
Plotting commands

Command	Description
axes	Create a new axes
axis	Set or return the current axis limits
bar	make a bar chart
boxplot	make a box and whiskers chart
cla	clear current axes
clabel	label a contour plot
clf	clear a figure window
close	close a figure window
colorbar	add a colorbar to the current figure
cohere	make a plot of coherence
contour	make a contour plot
contourf	make a filled contour plot
csd	make a plot of cross spectral density
draw	force a redraw of the current figure
errorbar	make an errorbar graph
figlegend	add a legend to the figure
figimage	add an image to the figure, w/o resampling
figtext	add text in figure coords
figure	create or change active figure
fill	make filled polygons
gca	return the current axes
gcf	return the current figure
gci	get the current image, or None
getp	get a handle graphics property
hist	make a histogram
hold	set the hold state on current axes
legend	add a legend to the axes
loglog	a log log plot
imread	load image file into array
imshow	plot image data
matshow	display a matrix in a new figure preserving aspect
pcolor	make a pseudocolor plot
plot	make a line plot
plotfile	plot data from a flat file
psd	make a plot of power spectral density
quiver	make a direction field (arrows) plot
rc	control the default params
savefig	save the current figure
scatter	make a scatter plot
setp	set a handle graphics property
semilogx	log x axis
semilogy	log y axis
show	show the figures
specgram	a spectrogram plot
stem	make a stem plot
subplot	make a subplot (numrows, numcols, axesnum)
table	add a table to the axes
text	add some text at location x,y to the current axes
title	add a title to the current axes
xlabel	add an xlabel to the current axes
ylabel	add a ylabel to the current axes

The following commands will set the default colormap accordingly:

- autumn
- bone
- cool
- copper
- flag
- gray
- hot
- hsv
- jet
- pink
- prism
- spring
- summer
- winter
- spectral

polar(**args, **kwargs*)
call signature:

```
polar(theta, r, **kwargs)
```

Make a polar plot. Multiple *theta*, *r* arguments are supported, with format strings, as in `plot()`.

prism()
Set the default colormap to prism and apply to current image if any. See `colormaps()` for more information.

psd(**args, **kwargs*)
call signature:

```
psd(x, NFFT=256, Fs=2, Fc=0, detrend=mlab.detrend_none,
    window=mlab.window_hanning, noverlap=0, **kwargs)
```

The power spectral density by Welches average periodogram method. The vector *x* is divided into *NFFT* length segments. Each segment is detrended by function *detrend* and windowed by function *window*. *noperlap* 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 length of the fft segment, must be a power of 2

***Fs*: integer** The sampling frequency.

***Fc*: integer** The center frequency of *x* (defaults to 0), which offsets the yextents of the image to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

***detrend*:** 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*:** The function used to window the segments. *window* is a function, unlike in matlab where it is a vector. `pylab` defines `window_none()`, and `window_hanning()`, but you can use a custom function as well.

***noverlap*: integer** Gives the length of the overlap between segments.

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
alpha	float
animated	[True | False]
antialiased or aa	[True | False]
axes	unknown
clip_box	a `matplotlib.transform.Bbox` instance
clip_on	[True | False]
clip_path	a `Path` instance and a
color or c	any matplotlib color
contains	unknown
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' | 'steps-pre' | 'steps-mid' | 'steps-post' | 'None' | ' ' | '']
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
picker	unknown
pickradius	unknown
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

Additional kwargs: hold = [True|False] overrides default hold state

quiver(**args, **kwargs*)

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*, ** are not supported at present.

Keyword arguments:

***units*: ['width' | 'height' | 'dots' | 'inches' | 'x' | 'y']** 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' or 'y': *X* or *Y* data units

In all cases 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.
The arrows scale differently depending on the units, however. 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.

***scale*: [None | float]** data units per arrow 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.

***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
alpha	float
animated	[True | False]
antialiased	Boolean or sequence of booleans
antialiaseds	Boolean or sequence of booleans
array	unknown
axes	an axes instance
clim	a length 2 sequence of floats
clip_box	a `matplotlib.transform.Bbox` instance
clip_on	[True | False]
clip_path	a `Path` instance and a
cmap	a colormap
color	matplotlib color arg or sequence of rgba tuples
colorbar	unknown
contains	unknown
dashes	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
edgecolor	matplotlib color arg or sequence of rgba tuples
edgecolors	matplotlib color arg or sequence of rgba tuples
facecolor	matplotlib color arg or sequence of rgba tuples
facecolors	matplotlib color arg or sequence of rgba tuples
figure	a `matplotlib.figure.Figure` instance
label	any string
linestyle	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
linestyles	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
linewidth	float or sequence of floats
linewidths	float or sequence of floats
lod	[True | False]
lw	float or sequence of floats
norm	unknown
offsets	float or sequence of floats
picker	[None|float|boolean|callable]
pickradius	unknown
transform	unknown
visible	[True | False]
zorder	any number

Additional kwargs: hold = [True|False] overrides default hold state

quiverkey(**args, **kwargs*)

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

`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.

rcdefaults()
Restore the default rc params - the ones that were created at matplotlib load time.

rgrids(**args, **kwargs*)
Set/Get the radial locations of the gridlines and ticklabels 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' )
```

savefig(**args, **kwargs*)
call signature:

```
savefig(fname, dpi=None, facecolor='w', edgecolor='w',
        orientation='portrait', papertype=None, format=None,
        transparent=False):
```

Save the current figure.

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.

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 figure patch and axes patches will all be transparent. 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.

`scatter`(**args, **kwargs*)

call signatures:

```
scatter(x, y, s=20, c='b', marker='o', cmap=None, norm=None,
        vmin=None, vmax=None, alpha=1.0, linewidths=None,
        verts=None, **kwargs)
```

Make a scatter plot of *x* versus *y*, where *x*, *y* are 1-D sequences 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:

Value	Description
's'	square
'o'	circle
'^'	triangle up
'>'	triangle right
'v'	triangle down
'<'	triangle left
'd'	diamond
'p'	pentagram
'h'	hexagon
'8'	octagon
'+'	plus
'x'	cross

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

Finally, *marker* can be (*verts*, 0): *verts* is a sequence of (*x*, *y*) vertices for a custom scatter symbol. Alternatively, use the kwarg combination *marker* = *None*, *verts* = *verts*.

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. 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** 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*:** 'none' to plot faces with no outlines

***facecolors*:** 'none' to plot unfilled outlines

Here are the standard descriptions of all the `Collection` kwargs:

Property	Description
alpha	float
animated	[True | False]
antialiased	Boolean or sequence of booleans
antialiaseds	Boolean or sequence of booleans
array	unknown
axes	an axes instance
clim	a length 2 sequence of floats
clip_box	a `matplotlib.transform.Bbox` instance
clip_on	[True | False]
clip_path	a `Path` instance and a
cmap	a colormap
color	matplotlib color arg or sequence of rgba tuples
colorbar	unknown
contains	unknown
dashes	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
edgecolor	matplotlib color arg or sequence of rgba tuples
edgecolors	matplotlib color arg or sequence of rgba tuples
facecolor	matplotlib color arg or sequence of rgba tuples
facecolors	matplotlib color arg or sequence of rgba tuples
figure	a `matplotlib.figure.Figure` instance
label	any string
linestyle	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
linestyles	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
linewidth	float or sequence of floats
linewidths	float or sequence of floats
lod	[True | False]
lw	float or sequence of floats
norm	unknown
offsets	float or sequence of floats
picker	[None|float|boolean|callable]
pickradius	unknown
transform	unknown
visible	[True | False]
zorder	any number

A `Collection` instance is returned.

Additional kwargs: hold = [True|False] overrides default hold state

sci(*im*)

Set the current image (target of colormap commands like `jet()`, `hot()` or `clim()`).

semilogx(**args, **kwargs*)

call signature:

```
semilogx(*args, **kwargs)
```

Make a plot with log scaling on the *x* axis.

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.

The remaining valid kwargs are Line2D properties:

Property	Description
alpha	float
animated	[True | False]
antialiased or aa	[True | False]
axes	unknown
clip_box	a matplotlib.transform.Bbox instance
clip_on	[True | False]
clip_path	a Path instance and a
color or c	any matplotlib color
contains	unknown
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' | 'steps-pre' | 'steps-mid' | 'steps-post' | 'None' | ' ' | '']
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
picker	unknown
pickradius	unknown
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

See loglog() for example code and figure

Additional kwargs: hold = [True|False] overrides default hold state

semilogy(**args, **kwargs*)

call signature:

```
semilogy(*args, **kwargs)
```

Make a plot with log scaling on the *y* axis.

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.

The remaining valid kwargs are Line2D properties:

Property	Description
alpha	float
animated	[True | False]
antialiased or aa	[True | False]
axes	unknown
clip_box	a `matplotlib.transform.Bbox` instance
clip_on	[True | False]
clip_path	a `Path` instance and a
color or c	any matplotlib color
contains	unknown
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' | 'steps-pre' | 'steps-mid' | 'steps-post' | 'None' | ' ' | '']
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
picker	unknown
pickradius	unknown
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

See `loglog()` for example code and figure

Additional kwargs: hold = [True|False] overrides default hold state

setp(**args, **kwargs*)

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(TM) 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
```

specgram(**args, **kwargs*)

call signature:

```
specgram(x, NFFT=256, Fs=2, Fc=0, detrend=mlab.detrend_none,
         window = mlab.window_hanning, noverlap=128,
         cmap=None, xextent=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*.

Keyword arguments:

***cmap*:** A `matplotlib.cm.Colormap` instance; if *None* use default determined by rc

***xextent*:** The image extent in the xaxes xextent=xmin, xmax default 0, max(bins), 0, max(freqs) where bins is the return value from mlab.specgram

See `psd()` for information on the other keyword arguments.

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 a len(times) x len(freqs) array of power
- *im* is a `matplotlib.image.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.

Additional kwargs: hold = [True|False] overrides default hold state

spectral()
Set the default colormap to spectral and apply to current image if any. See `colormaps()` for more information.

spring()
Set the default colormap to spring and apply to current image if any. See `colormaps()` for more information.

spy(**args, **kwargs*)
call signature:

```
spy(Z, precision=None, marker=None, markersize=None,
    aspect='equal', **kwargs)
```

`spy(Z)` plots the sparsity pattern of the 2-D array *Z*.

If *precision* is *None*, any non-zero value will be plotted; else, values of |*Z*| > *precision* will be plotted.

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 documentation for `imshow()` for details.

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*

See documentation for `plot()` for details.

Useful values for *marker* include:

- 's' square (default)
- 'o' circle
- '.' point
- ',' pixel

Additional kwargs: hold = [True|False] overrides default hold state

stem(**args, **kwargs*)

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 this document for details and `examples/pylab_examples/stem_plot.py` for a demo.

Additional kwargs: hold = [True|False] overrides default hold state

step(**args, **kwargs*)

call signature:

```
step(x, y, *args, **kwargs)
```

Make a step plot. 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]
If 'post', that interval has level y[i+1]
If 'mid', the jumps in *y* occur half-way between the *x*-values.

Additional kwargs: hold = [True|False] overrides default hold state

subplot(**args, **kwargs*)

Create a subplot command, 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.

The background color of the subplot can be specified via keyword argument *axisbg*, which takes a color string as value, as in:

```
subplot(211, axisbg='y')
```

See axes() for additional information on axes() and subplot() keyword arguments.

New subplots that overlap old will delete the old axes. If you do not want this behavior, use matplotlib.figure.Figure.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_tool(*targetfig=None*)

Launch a subplot tool window for *targetfig* (default gcf).

A matplotlib.widgets.SubplotTool instance is returned.

subplots_adjust(**args, **kwargs*)

call signature:

```
subplots_adjust(left=None, bottom=None, right=None, top=None,
                wspace=None, hspace=None)
```

Tune the subplot layout via the matplotlib.figure.SubplotParams mechanism. 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

summer()

Set the default colormap to summer and apply to current image if any. See colormaps() for more information.

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 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.subtitle('this is the figure title', fontsize=12)
```

switch_backend(*newbackend*)

Switch the default backend to newbackend. 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.

table(**args, **kwargs*)

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):
```

Add a table to the current axes. 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
alpha	float
animated	[True | False]
axes	an axes instance
clip_box	a `matplotlib.transform.Bbox` instance
clip_on	[True | False]
clip_path	a `Path` instance and a
contains	unknown
figure	a `matplotlib.figure.Figure` instance
fontsize	a float in points
label	any string
lod	[True | False]
picker	[None|float|boolean|callable]
transform	unknown
visible	[True | False]
zorder	any number

text(**args, **kwargs*)

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 `matplotlib.text.Text` properties:

Property	Description
alpha	float
animated	[True | False]
axes	an axes instance
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
color	any matplotlib color
contains	unknown
family	['serif' | 'sans-serif' | 'cursive' | 'fantasy' | 'monospace']
figure	a `matplotlib.figure.Figure` instance
fontproperties	a matplotlib.font_manager.FontProperties instance
horizontalalignment or ha	['center' | 'right' | 'left']
label	any string
linespacing	float
lod	[True | False]
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	unknown
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

thetagrids(**args, **kwargs*)

Set/Get the theta locations of the gridlines and ticklabels.

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') )
```

title(*s, *args, **kwargs*)

Set the title of the current axis to *s*.

Default font override is:

```
override = {'fontsize': 'medium',
            'verticalalignment': 'bottom',
            'horizontalalignment': 'center'}
```

See the `text()` docstring for information of how override and the optional args work.

twinx(*ax=None*)

Make a second axes overlay *ax* (or the current axes if *ax* is *None*) sharing the xaxis. The ticks for *ax2* will be placed on the right, and the *ax2* instance is returned.

See `examples/pylab_examples/two_scales.py`

twiny(*ax=None*)

Make a second axes overlay *ax* (or the current axes if *ax* is *None*) sharing the yaxis. The ticks for *ax2* will be placed on the top, and the *ax2* instance is returned.

vlines(*args, **kwargs*)

call signature:

```
vlines(x, ymin, ymax, color='k')
```

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 is a line collections color args, either a single color or a len(*x*) list of colors

linestyle is one of ['solid' | 'dashed' | 'dashdot' | 'dotted']

Returns the `matplotlib.collections.LineCollection` that was added.

kwargs are `LineCollection` properties:

Property	Description
alpha	float
animated	[True | False]
antialiased	Boolean or sequence of booleans
antialiaseds	Boolean or sequence of booleans
array	unknown
axes	an axes instance
clim	a length 2 sequence of floats
clip_box	a `matplotlib.transform.Bbox` instance
clip_on	[True | False]
clip_path	a `Path` instance and a
cmap	a colormap
color	matplotlib color arg or sequence of rgba tuples
colorbar	unknown
contains	unknown
dashes	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
edgecolor	matplotlib color arg or sequence of rgba tuples
edgecolors	matplotlib color arg or sequence of rgba tuples
facecolor	matplotlib color arg or sequence of rgba tuples
facecolors	matplotlib color arg or sequence of rgba tuples
figure	a `matplotlib.figure.Figure` instance
label	any string
linestyle	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
linestyles	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq)]
linewidth	float or sequence of floats
linewidths	float or sequence of floats
lod	[True | False]
lw	float or sequence of floats
norm	unknown
offsets	float or sequence of floats
picker	[None|float|boolean|callable]
pickradius	unknown
segments	unknown
transform	unknown
verts	unknown
visible	[True | False]
zorder	any number

Additional kwargs: hold = [True|False] overrides default hold state

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.

winter()

Set the default colormap to winter and apply to current image if any. See colormaps() for more information.

xcorr(**args, **kwargs*)

call signature:

```
xcorr(x, y, normed=False, detrend=mlab.detrend_none,
      usevlines=False, **kwargs):
```

Plot the cross correlation between *x* and *y*. If *normed* = *True*, normalize the data but 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*:

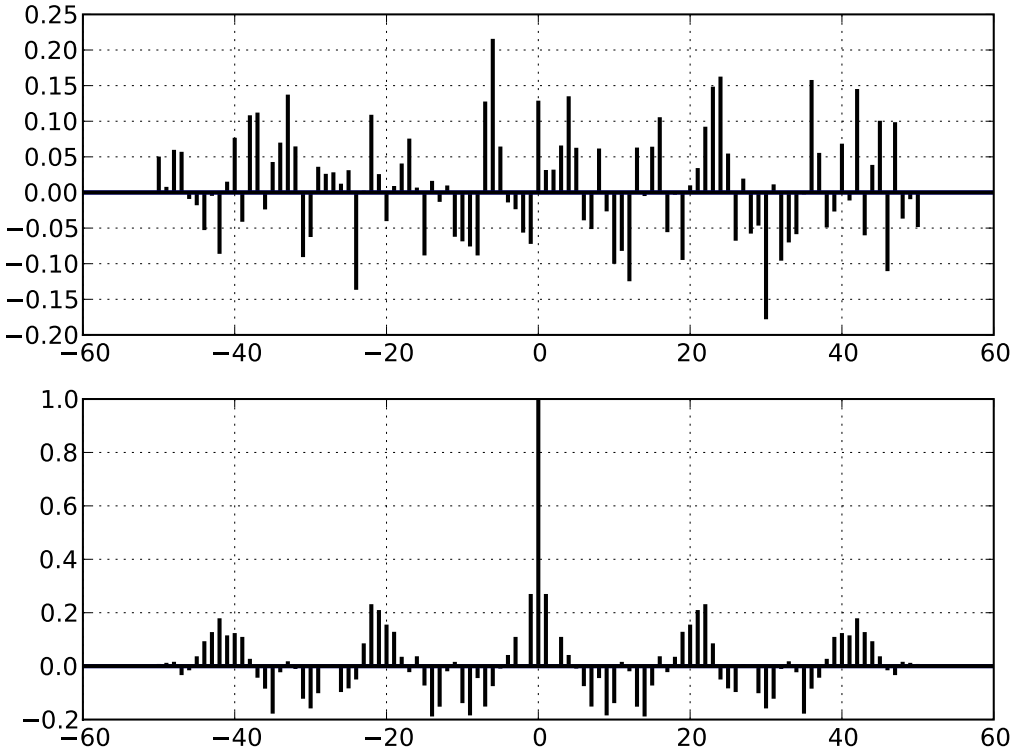
> vlines() rather than plot() is used to draw vertical lines from the origin to the xcorr. Otherwise the plotstyle is determined by the kwargs, which are Line2D properties.
>
> The return value is a tuple (*lags*, *c*, *linecol*, *b*) where *linecol* is the matplotlib.collections.LineCollection instance 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() above, and acorr() below.

Example:



Additional kwargs: hold = [True|False] overrides default hold state

xlabel(*s, *args, **kwargs*)

Set the *x* axis label of the current axis to *s*

Default override is:

```
override = {
    'fontsize'            : 'small',
    'verticalalignment'   : 'top',
    'horizontalalignment' : 'center'
    }
```

See `text()` for information of how override and the optional args work

xlim(**args, **kwargs*)

Set/Get the xlimits 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
```

The new axis limits are returned as a length 2 tuple.

xscale(**args, **kwargs*)

call signature:

```
xscale(scale, **kwargs)
```

Set the scaling for the x-axis: 'linear' | 'log' | 'symlog'

Different keywords may be accepted, depending on the scale:

'linear'

'log'

***basex/basey*:** The base of the logarithm

***subsx/subsy*:** Where to place the subticks between each major tick. Should be a sequence of integers. For example, in a log10 scale: `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`
will place 10 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: `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`
will place 10 logarithmically spaced minor ticks between each major tick.

xticks(**args, **kwargs*)

Set/Get the xlimits of the current ticklocs 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.

ylabel(*s, *args, **kwargs*)

Set the *y* axis label of the current axis to *s*.

Defaults override is:

```
override = {
   'fontsize'            : 'small',
   'verticalalignment'   : 'center',
   'horizontalalignment' : 'right',
   'rotation'='vertical' : }
```

See `text()` for information on how override and the optional args work.

ylim(**args, **kwargs*)

Set/Get the ylimits 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
```

The new axis limits are returned as a length 2 tuple.

yscale(**args, **kwargs*)

call signature:

```
xscale(scale, **kwargs)
```

Set the scaling for the y-axis: 'linear' | 'log' | 'symlog'

Different keywords may be accepted, depending on the scale:

'linear'

'log'

***basex/basey*:** The base of the logarithm

***subsx/subsy*:** Where to place the subticks between each major tick. Should be a sequence of integers. For example, in a log10 scale: `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`
will place 10 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: `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`
will place 10 logarithmically spaced minor ticks between each major tick.

yticks(**args, **kwargs*)

Set/Get the ylimits of the current ticklocs 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.

CHAPTER

THIRTYONE

Matplotlib backends

31.1 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.

class `Cursors`()

class `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 `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 `FigureCanvasBase`(*figure*)

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, 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.

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

flush_events()
Flush the GUI events for the figure. Implemented only for backends with GUIs.

get_default_filetype()

get_supported_filetypes()

get_supported_filetypes_grouped()

get_width_height()
return the figure width and height in points or pixels (depending on the backend), truncated to integers

idle_event(*guiEvent=None*)
call 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

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'

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)
```

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

print_pdf(**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*)

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:

Literal block expected; none found.

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 FigureManagerBase(*canvas, num*)

Helper class for matlab mode, wraps everything up into a neat bundle

Public attibutes:

canvas A FigureCanvasBase instance

num The figure nuamber

destroy()

full_screen_toggle()

key_press(*event*)

resize(*w, h*)

For gui backends: resize window in pixels

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).

show_popup(*msg*)
Display message in a popup – GUI only

class **GraphicsContextBase**()
An abstract base class that provides color, line styles, etc...

copy_properties(*gc*)
Copy properties from gc to self

get_alpha()
Return the alpha value used for blending - not supported on all backends

get_antialiased()
Return true if the object should try to do antialiased rendering

get_capstyle()
Return the capstyle as a string in ('butt', 'round', 'projecting')

get_clip_path()
Return the clip path in the form (path, transform), where path is a `Path` instance, and transform is an affine transform to apply to the path before clipping.

get_clip_rectangle()
Return the clip rectangle as a `Bbox` instance

get_dashes()
Return the dash information as an offset dashlist tuple The dash list is a even size list that gives the ink on, ink off in pixels. See p107 of to postscript BLUEBOOK for more info

Default value is None

get_hatch()
Gets the current hatch style

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 floats from 0-1. color can be a matlab format string, a html hex color string, or a rgb tuple

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 unit tuple, or a float between 0 and 1. In the latter case, grayscale is used.

The GraphicsContextBase converts colors to rgb internally. If you know the color is rgb already, you can set `isRGB=True` to avoid the performace hit of the conversion

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').

set_linewidth(*w*)
Set the linewidth in points

class 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 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 pressed: None, chr(range(255), shift, win, or control

This interface may change slightly when better support for modifier keys is included.

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 LocationEvent(*name, canvas, x, y, guiEvent=None*)

Bases: matplotlib.backend_bases.Event

A event that has a screen location

The following additional attributes are defined and shown with their default values

In addition to the Event attributes, the following event attributes are defined:

x x position - pixels from left of canvas

y y position - pixels from bottom of canvas

inaxes the Axes instance if mouse is over axes

xdata x coord of mouse in data coords

ydata y coord of mouse in data coords

x, *y* in figure coords, 0,0 = bottom, left

class MouseEvent(*name, canvas, x, y, button=None, key=None, step=0, 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 pressed: None, chr(range(255), 'shift', 'win', or 'control'

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'

class NavigationToolbar2(*canvas*)

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

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*)
this will be 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

update()
reset the axes stack

zoom(**args*)
activate zoom to rect mode

class 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

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 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_image(*x, y, im, bbox, clippath=None, clippath_trans=None*)
Draw the image instance into the current axes;

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

bbox a `matplotlib.transforms.Bbox` instance for clipping, or None

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(*master_transform, cliprect, clippath, clippath_trans, paths, all_transforms, offsets, offsetTrans, facecolors, edgecolors, linewidths, linestyles, antialiaseds*)

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.

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(*master_transform, cliprect, clippath, clippath_trans, meshWidth, meshHeight, coordinates, offsets, offsetTrans, facecolors, antialiased, showedges*)

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*)
Open a grouping element with label *s*. Is only currently used by backend_svg

option_image_nocomposite()
overwrite this method for renderers that do not necessarily want to rescale and composite raster images. (like SVG)

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_rasterizing()

stop_rasterizing()

strip_math(*s*)

class 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

31.2 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.

31.3 matplotlib.backends.backend_qt4agg

autodoc can't import/find module 'matplotlib.backends.backend_qt4agg', it reported error: "cannot import name QtCore",please check your spelling and sys.path

31.4 matplotlib.backends.backend_wxagg

class 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.

print_figure(*filename, *args, **kwargs*)

class FigureFrameWxAgg(*num, fig*)

Bases: matplotlib.backends.backend_wx.FigureFrameWx

get_canvas(*fig*)

class NavigationToolbar2WxAgg(*canvas*)

Bases: matplotlib.backends.backend_wx.NavigationToolbar2Wx

get_canvas(*frame, fig*)

new_figure_manager(*num, *args, **kwargs*)

Create a new figure manager instance

Part V

Glossary

AGG The Anti-Grain Geometry (Agg) rendering engine, capable of rendering high-quality images

Cairo The Cairo graphics engine

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.

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.

MODULE INDEX

M

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z