

News flash

matplotlib now has a [wiki](#). It's just getting started, so move the ball forward by adding a tip, trick, howto or recipe.

Matplotlib

[Home](#)
[What's New](#)
[Download](#)
[Installing](#)
[Screenshots](#)
[Examples \(zip\)](#)
[Mailing lists](#)

Documentation

[Tutorial](#)
[User's Guide \(pdf\)](#)
[FAQ](#)
[Cookbook / wiki](#)
[pylab interface](#)
[Class library](#)
[Backends](#)
[Toolkits](#)
[Fonts](#)
[Interactive](#)
[Goals](#)

Other

[Credits](#)
[License](#)

Using matplotlib

If you are new to python, I recommend reading as much of the [python tutorial](#) and [manual](#) as possible before working with matplotlib. Otherwise you may get frustrated. If you are comfortable with both, you'll find matplotlib easy.

There are a lot of features under the hood in matplotlib. This tutorial just scratches the surface. If you are finished with it, the next step (other than getting to work on your own figure) is to download the source distribution (*.tar.gz or *.zip) and take a look at the [matplotlib](#) subdirectory. If you are working with date plots you'll find several date demos like [examples/date_demo1.py](#). Likewise, you'll find examples for images [example](#), [contouring](#) [examples/contour_demo.py](#), using matplotlib with a graphical user interface in [examples/embedding_in_wx.py](#) and many more. Because these are not included in the windows installer, they are often overlooked, which is why I emphasize them here.

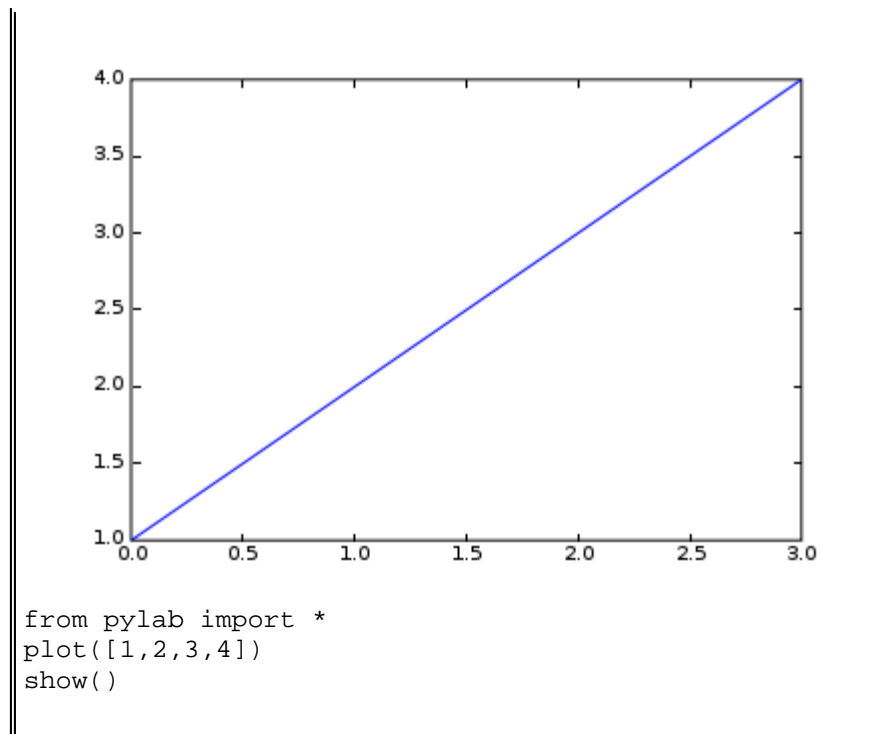
The next place to turn to find the hidden gems is the [what's new](#) page. Every feature introduced into matplotlib is listed on that page with the version it was introduced, usually with a link to the source code and functions. Scrolling through that page is one of the best ways to find out what customizations are supported.

matplotlib is designed to work in a variety of settings: some people use it in a batch server to create images they never look at. Others use graphical user interfaces (GUIs) to create their plots. Thus you must customize matplotlib to work like you want it too. In particular, take a look at the customization file [matplotlibrc](#), in which you can set whether you want to save images or use a GUI (the `backend` setting), and whether you want to work interactively (the `interactive` setting). Also, please read [interactive use](#), [What's up with show window is freezing](#) before trying to type in the examples below, to avoid the confusion of new matplotlib users. Or at least make a mental note to come back and read them when you have problems.

Using matplotlib should come naturally if you have ever plotted with matlab, and is straightforward if you haven't. The basic entity is a figure, which contains axes, windows, and text. The axes are decorated with xlabels, ylabels, titles, ticklabels, and text.

Here is about the simplest script you can use to create a figure with matplotlib

A simple plot



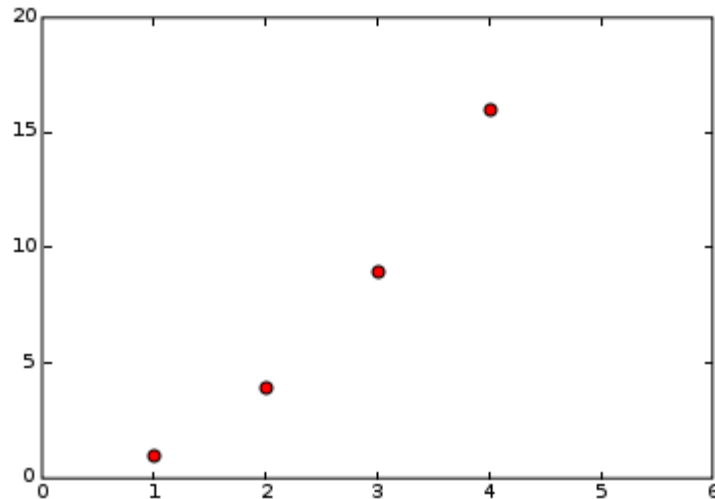
If you are new to python, the first question you are probably asking yourself about does the xaxis range from 0-3 and the yaxis from 1-4." The answer is that if you pass an array to the plot command, matplotlib assumes it a vector of y-values, and automatically generates the x-values for you. Since python ranges start with 0, the default x vector has the same length as the y vector but starts with 0. Hence the x vector is [0,1,2,3]. Of course, if you don't want this default behavior, you can supply the x data explicitly, as in `plot(x,y)` where `x` and `y` are

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

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

For every `x, y` pair of arguments, there is an optional third argument which is the format string. This indicates the color and line type of the plot. The letters and symbols of the format string are defined in the `matplotlib` documentation, and you concatenate a color string with a line style string. The default format string is `'b'`, which is a solid blue line (don't ask me, talk to [The Mathworks](#)). For example, to plot `y` versus `x` with red circles, you would issue

Using format strings

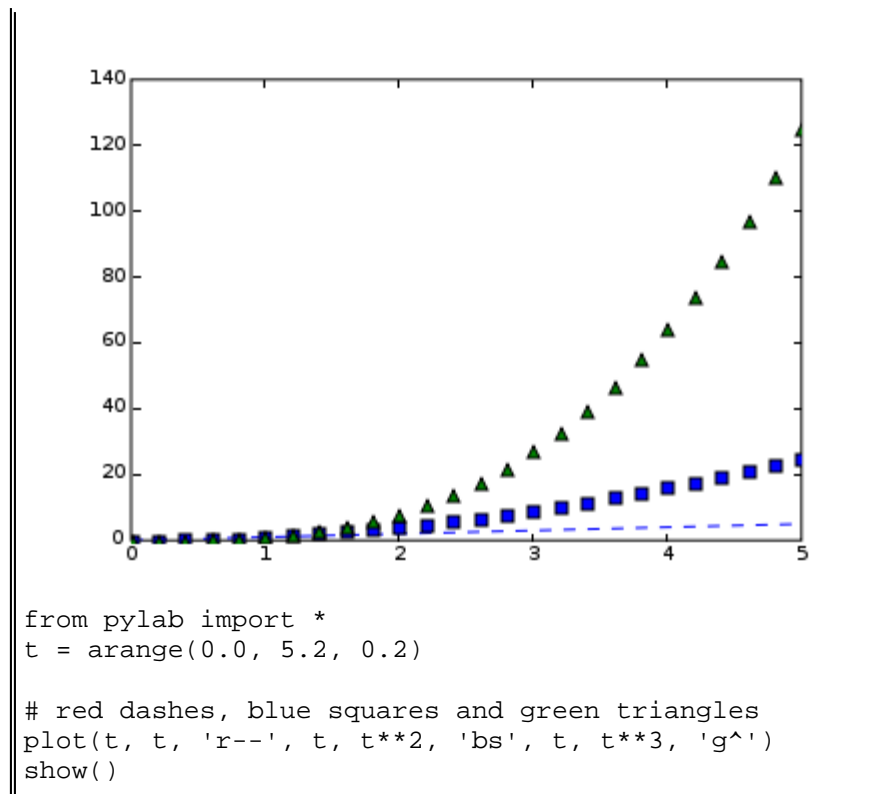


```
from pylab import *  
plot([1,2,3,4], [1,4,9,16], 'ro')  
axis([0, 6, 0, 20])  
savefig('secondfig.png')  
show()
```

See the [plot](#) documentation for a complete list of line styles and format strings. The example above takes a list of `[xmin, xmax, ymin, ymax]` and specifies the

If matplotlib were limited to working with lists, it would be fairly useless for numerical plotting. Generally, you will use [Numeric](#) arrays. Note that matplotlib works equally well with the successor [numarray](#), thanks to the [matplotlib.numerix](#) interface, which enables you to use numarray package transparently. In fact, all sequences are converted to numerix arrays internally. Below illustrates a plotting several lines with different format styles in one command. Note if you are not familiar with Numeric or numarray, now would be a good time to read the documentation at [Numerical python](#) since a lot of the matplotlib examples assume Numeric, and the pylab module imports all the numerical python and associated functionality.

Multiple lines with one plot command



Controlling line properties

Lines have many attributes that you can set: linewidth, dash style, antialiased, etc

There are several ways to set line properties

```

# Use keyword args
plot(x, y, linewidth=2.0)

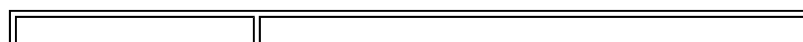
# Use the setter methods of the Line2D instance. plot returns a
# of lines; eg line1, line2 = plot(x1,y1,x2,x2). Below I have on
# one line so it is a list of length 1. I use tuple unpacking in
# line, = plot(x, y, 'o') to get the first element of the list
line, = plot(x, y, 'o')
line.set_antialiased(False)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

# Use the set command. The example below uses matlab handle grap
# style command to set multiple properties on a list of lines. S
# works transparently with a list of objects or a single object
lines = plot(x1, y1, x2, y2)
setp(lines, color='r', linewidth=2.0)

# set also accepts matlab style pairs of arguments with property
# as strings, though I prefer the pythonic keyword arguments above
setp(lines, 'color', 'r', 'linewidth', 2.0)

```

Line Properties



Property	Values
alpha	The alpha transparency on 0-1 scale
antialiased	True False - use antialiased rendering
color	a matplotlib color arg
data_clipping	Whether to use numeric to clip data
label	a string optionally used for legend
linestyle	One of -- : -. -
linewidth	a float, the line width in points
marker	One of + , o . s v x > < ^
markeredgewidth	The line width around the marker symbol
markeredgecolor	The edge color if a marker is used
markerfacecolor	The face color if a marker is used
markersize	The size of the marker in points

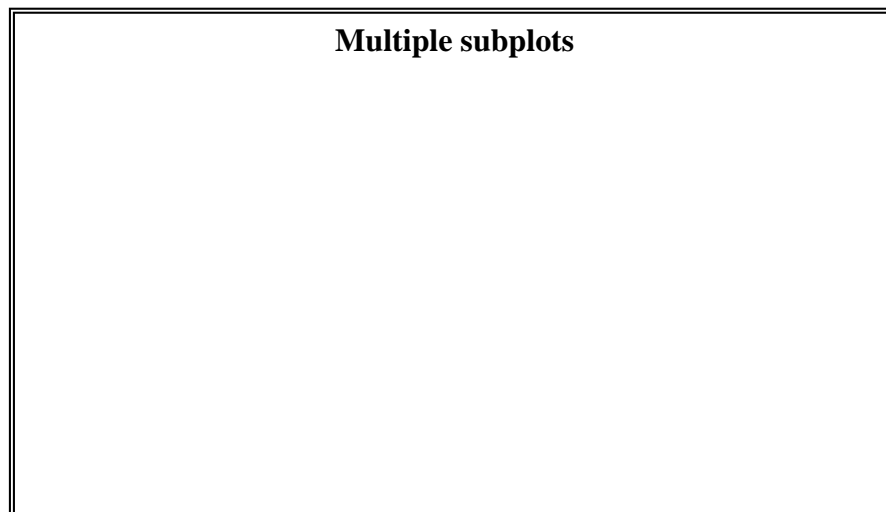
To get a list of settable line properties, call the [setp](#) function with a line or lines as

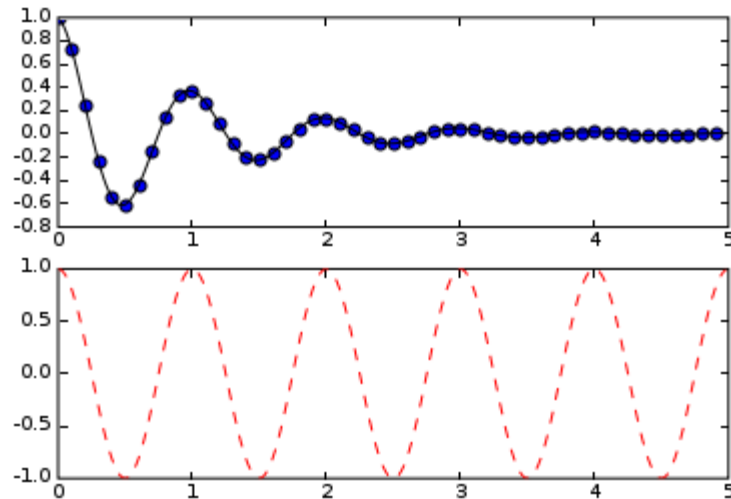
```
>>> lines = plot([1,2,3])
>>> setp(lines)
```

Working with multiple figure and axes

Matlab, and pylab, have the concept of the current figure and the current axes. All commands apply to the current axes. The function [gca](#) returns the current axes as [gcf](#) returns the current figure as a [Figure](#) instance.

Normally, you don't have to worry about this, because it is all taken care of behind an script to create two subplots





```

from pylab import *

def f(t):
    s1 = cos(2*pi*t)
    e1 = exp(-t)
    return multiply(s1,e1)

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

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

subplot(212)
plot(t2, cos(2*pi*t2), 'r--')
show()

```

The [figure](#) command here is optional because 'figure(1)' will be created by default (111) will be created by default if you don't manually specify an axes. The [subplot](#) numrows, numcols, fignum where fignum ranges from 1 to numrows*numcols 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 create subplots manually, ie, not on a rectangular grid, use the [axes](#) command, which allows you to specify axes as axes([left, bottom, width, height]) where all values are in fractional coordinates. See [axes_demo.py](#) for an example of placing axes manually and [line_styles.py](#) for lots-o-subplots.

You can create multiple figures by using multiple [figure](#) calls with an increasing figure number. Each figure can contain as many axes and subplots as your heart desires.

```

from pylab import *

figure(1) # the first figure
plot([1,2,3])
figure(2) # a second figure

```

```

plot([4,5,6])

figure(1)           # figure 1 current
title('Easy as 1,2,3') # figure 1 title
show()

```

You can clear the current figure with [clf](#) and the current axes with [cla](#)

Working with text

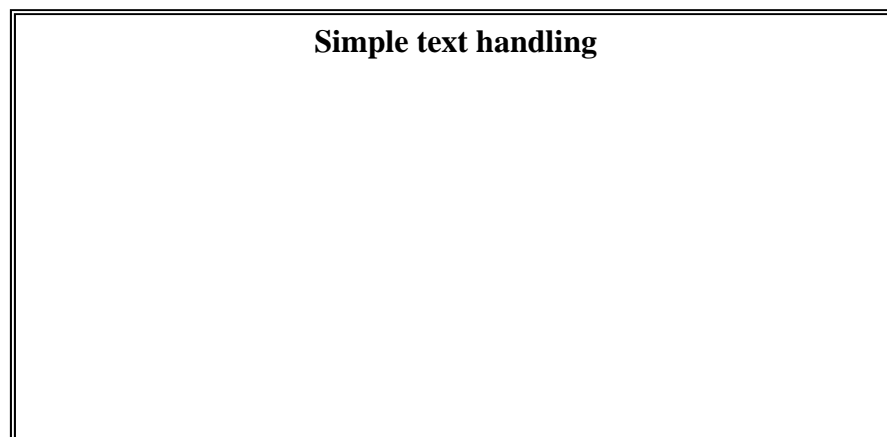
All of the text commands ([xlabel](#), [ylabel](#), [title](#), and [text](#)) take optional keyword arguments and dictionaries to specify the font properties. There are three ways to specify font properties: using [setp](#), object oriented methods, and font dictionaries.

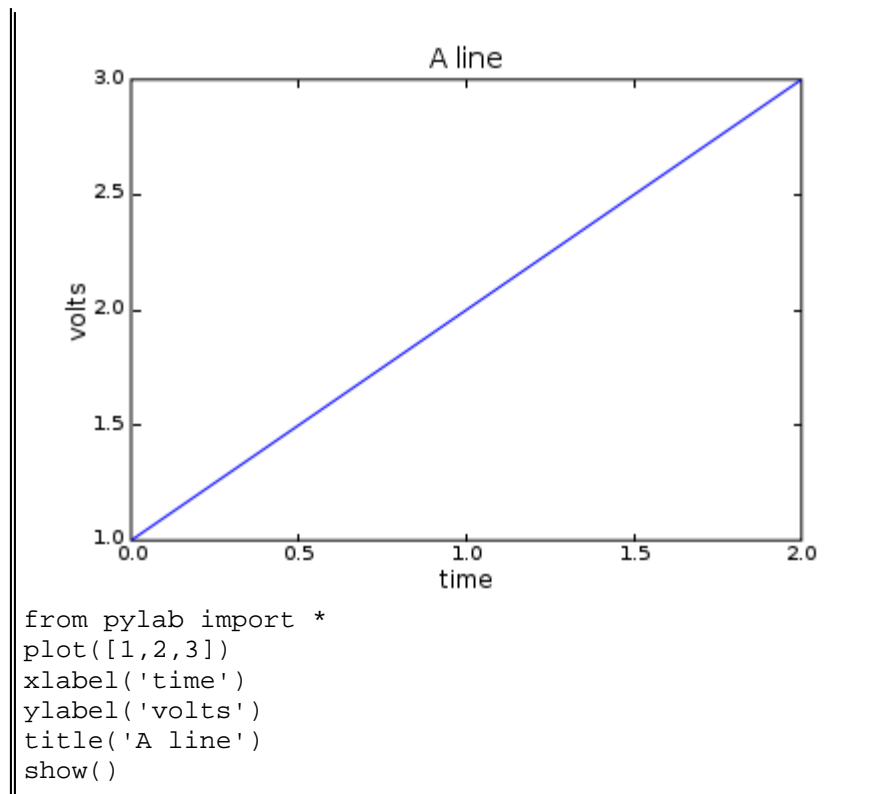
The text commands return an [Text](#) instance (or a list of instances if multiple text objects are created) and the following font properties can be set; these names are compatible with matplotlib's font dictionary for text

Property	Values
alpha	The alpha transparency on 0-1 scale
color	a matplotlib color argument
fontangle	italic normal oblique
fontname	Sans Helvetica Courier Times Others
fontsize	an scalar, eg, 10
fontweight	normal bold light 4
horizontalalignment	left center right
rotation	horizontal vertical
verticalalignment	bottom center top

See [align_text](#) for examples of how to control the alignment and orientation of text

Here is an example adding text to a simple plot





Controlling text properties with handle graphics

If you want to change a text property, and you like to use matlab handle graphics, command to set any of the properties listed in the table above. For example, to make the x-axis label red and bold, you would use

```
t = xlabel('time')
setp(t, color='r', fontweight='bold')
```

Set also works with a list of text instances. The following changes the properties of all x-axis tick labels:

```
labels = getp(gca(), 'xticklabels')
setp(labels, color='r', fontweight='bold')
```

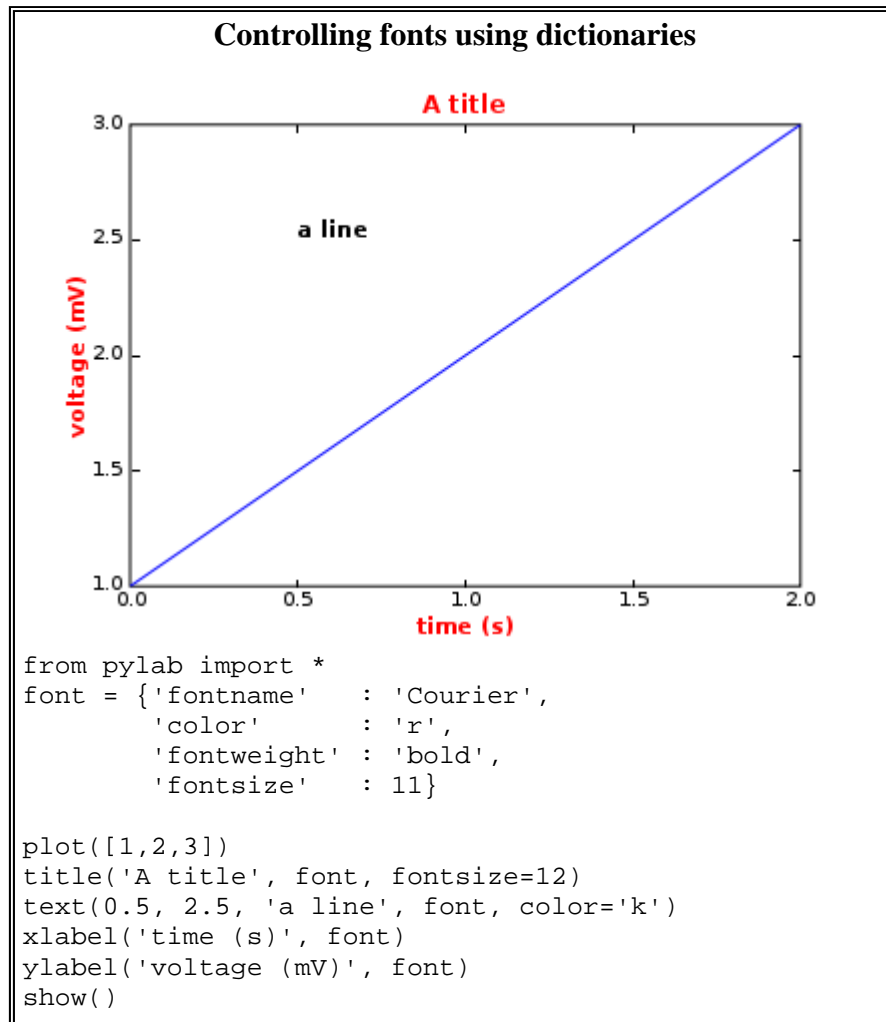
Controlling text using object methods

The [setp](#) command is just a wrapper around the [Text](#) set methods. If you prefer using object methods, you just prepend `set_` to the text property and make a normal python instance method call:

```
t = xlabel('time')
t.set_color('r')
t.set_fontweight('bold')
```

Controlling text using kwargs and dicts

All of the text commands take an optional dictionary and keyword arguments to control text properties. For example, if you want to set a default font theme, and override individual properties, you could do something like



Now, all of the text has the default theme, which is set by the dictionary `font` (bo point), but individual pieces of text can selectively override properties by passing. For example the `text` command uses color black with the color string 'k'.

Writing mathematical expressions

You can use TeX markup in your expressions; see the [mathtext](#) documentation for and backend information.

Any text element can use math text. You need to use raw strings (precede the quote surround the string text with dollar signs, as in TeX.

```

# plain text
title('alpha > beta')

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

```

To make subscripts and superscripts use the '_' and '^' symbols, as in

```

title(r'$\alpha_i > \beta_i$')

```

You can also use a large number of the TeX symbols, as in `\infty`, `\leftarrow` [mathtext](#) for a complete list. The over/under subscript/superscript style is also supported. For example, to write the sum of x_i from 0 to infinity, you could do

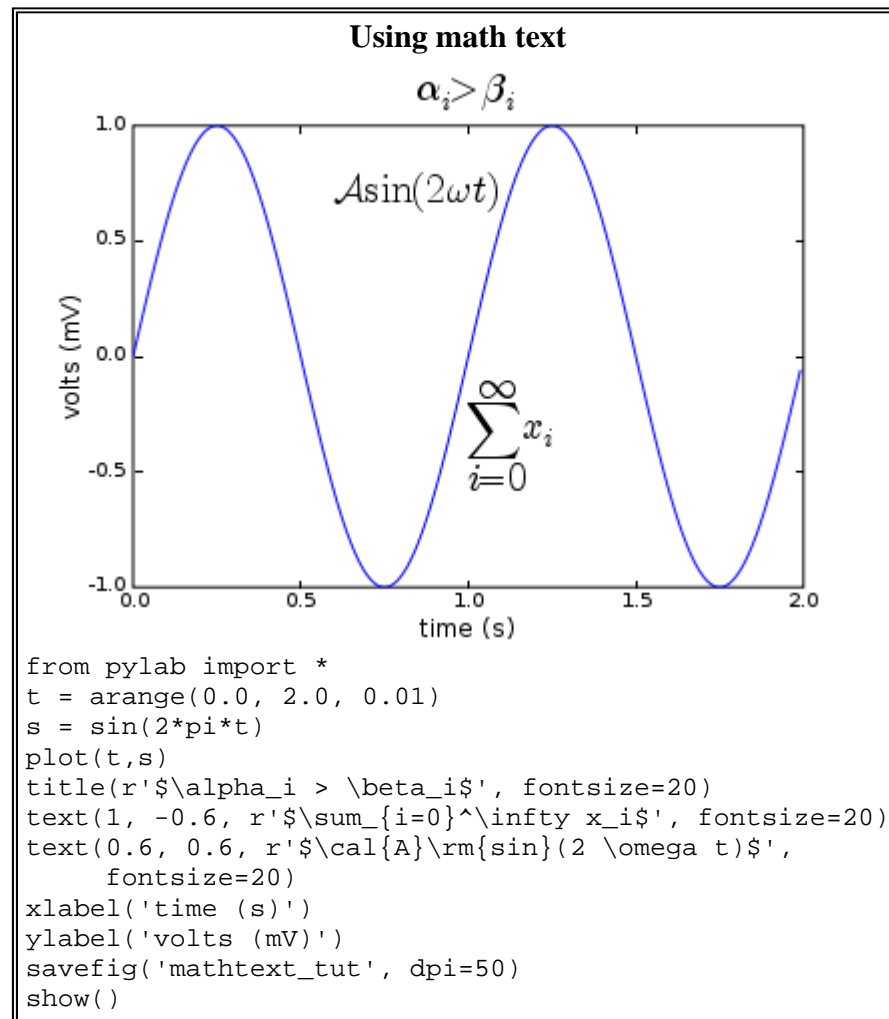
```
text(1, -0.6, r'$\sum_{i=0}^{\infty} x_i$')
```

The default font is *italics* for mathematical symbols. To change fonts, eg, to write $s(t) = A \sin(2 \omega t)$, you could do

```
text(1,2, r's(t) = $\cal{A}\rm{sin}(2 \omega t)$')
```

Here "s" and "t" are variable in italics font (default), "sin" is in roman font, and the amplitude "A" is in caligraphy font. The font choices are roman `\rm`, italics `\it`, caligraphy `\cal`, and

The following accents are provided: `\hat`, `\breve`, `\grave`, `\bar`, `\acute`, `\tilde`. All of them have the same syntax, eg to make an overbar you do `\bar{o}` or to make a double dot you do `\ddot{o}`.

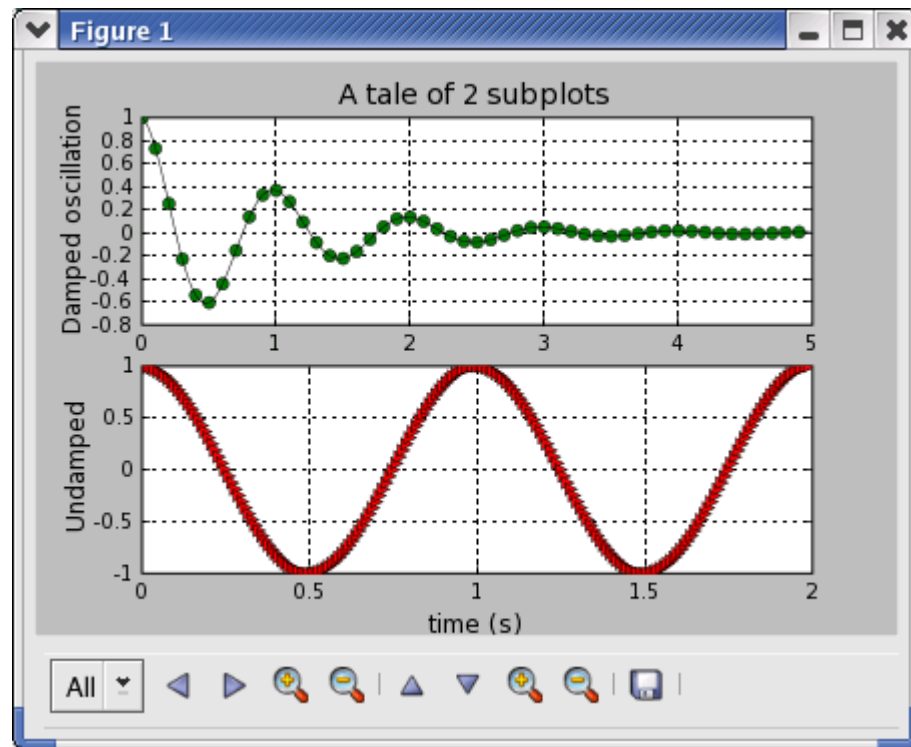


Interactive navigation

All figure windows come with a navigation toolbar, which can be used to navigate. You can select either the "classic" or newfangled toolbar "toolbar2" in your `matplotlib.toolbar` setting. If you want to interact with very large data sets, matplotlib supports where the data is clipped with `numarray` before they are plotted to the active view. For a very large data set, plot the whole thing, set the view limits to a narrow range, and scroll through the data with good interactive refresh rates. I wrote an EEG viewer (see [this](#) in matplotlib and routinely plot 25 MB data sets in matplotlib with good performance. See the [stock_demo.py](#) that comes with the matplotlib src for an example of a long limited data view -- only the first 3 of 60 days worth of minute by minute stock quotes for Apple are initially shown in the view port. As in this example, data clipping is a feature explicitly turned on.

Classic toolbar

You can pan and zoom on the X and Y axis for any combination of the axes that you have a wheel mouse, you can move bidirectionally by scrolling the wheel over the plot. In the examples, the wheel mouse can be used to pan left or right by scrolling over *either* the left or right arrow buttons, so you never have to move the mouse to pan the x-axis left or right. If you don't have a wheel mouse, buy one!



The left widget that says 'All' on the controls on the bottom of the figure is a drop down menu used to select which axes the controls affect. You can select all, none, single, or combinations of axes. The first set of 4 controls are used to pan left, pan right, zoom in and zoom out on the x axes. The second set are used to pan up, pan down, zoom in and zoom out on the y axes. The remaining buttons are used to redraw the figure, save (PNG or JPEG) the figure, or to close the figure window.

toolbar2

The toolbar2 buttons behave very differently from the classic the classic matplotlib introduce a new one!) despite the visual similarity of the forward and back button details.

The Forward and Back buttons are akin to the web browser forward and back buttons to navigate back and forth between previously defined views. They have no meaning if you have already navigated somewhere else using the pan and zoom buttons. This is analogous to Back on your web browser before visiting a new page --nothing happens. Home at first, default view of your data. For Home, >Forward and Back, think web browser buttons for web pages. Use the pan and zoom to rectangle to define new views.

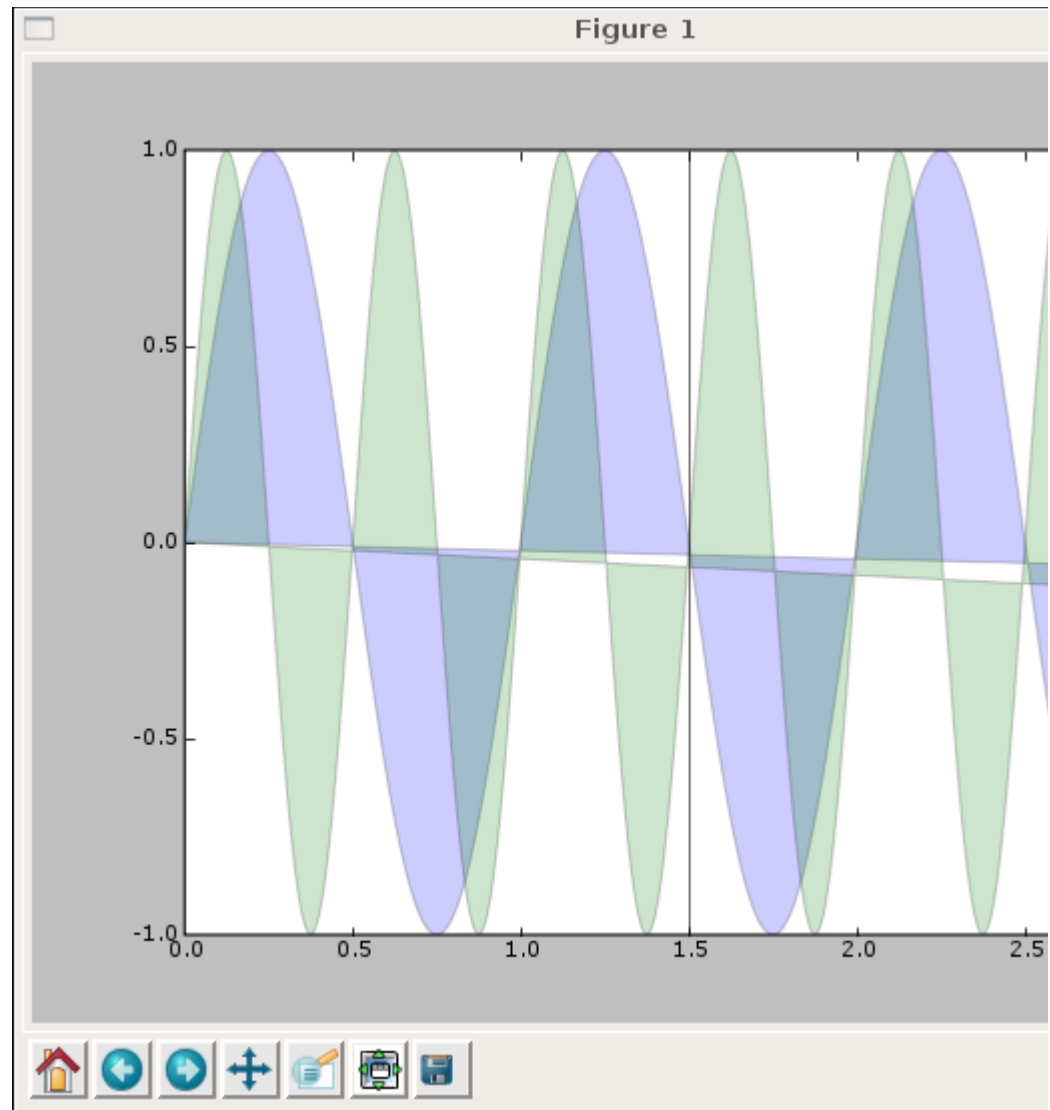
The Pan/Zoom button has two modes: pan and zoom. Click this toolbar button to activate it. Then put your mouse somewhere over an axes.

- **Pan Mode:** Press the left mouse button and hold it, dragging it to a new position. When you release it, the data under the point where you pressed will be moved to the new position. If you press 'x' or 'y' while panning, the motion will be constrained to the x or y axis respectively.
- **Zoom Mode:** Press the right mouse button, dragging it to a new position. The plot is zoomed in proportionate to the rightward movement and zoomed out proportionate to leftward movement. Ditto for the yaxis and up/down motions. The point where you begin the zoom remains stationary, allowing you to zoom to an arbitrary location. You can use the modifier keys 'x', 'y' or 'CONTROL' to constrain the zoom to the x or y axis, or aspect ratio preserve, respectively.

The Zoom to rectangle button: Click this toolbar button to activate this mode. Click somewhere over an axes and press the left mouse button. Drag the mouse to a new location and release. The axes view limits will be zoomed to the rectangle defined by the mouse. There is also an experimental 'zoom out to rectangle' in this mode with the right button. Click to place your entire axes in the region defined by the zoom out rectangle.

Subplot configuration Use this tool to configure the parameters of the subplot: title, position, bottom, space between the rows and space between the columns.

Save: click this button to launch a file save dialog. All the *Agg backends know the following image types: PNG, PS, EPS, SVG. There is no support currently in Agg for TIFF (the regular wx and gtk backends handle these types). It is possible to use it to convert agg images to one of these other formats if required. I can provide a recipe for saving PNG over JPG and TIFF, which is why I haven't worked too hard to include these in agg.



The new toolbar2

Customizing matplotlib

matplotlib uses an rc configuration file [matplotlibrc](#). At installation time, this is placed in a system path, eg `C:\Python23\share\matplotlib\matplotlibrc` on windows or `/usr/local/share/matplotlib/matplotlibrc` on linux and friends. Every time you install matplotlib, this file will be overwritten, so if you want your customizations to be persistent, you should copy this file to your HOME directory and make sure the HOME environment variable is set to that directory.

You can control the defaults of almost every property in matplotlib: figure size and position, color and style, axes, axis and grid properties, text and font properties and so on. This is documented within it, so please see [matplotlibrc](#) for more information.

You can also dynamically change the defaults in a python script or interactively using the `rc` command. For example to change the default line properties, you could

```
>>> rc('lines', linewidth=2, color='r')
```

And the default lines would be thicker and red. All rc parameters except backend, interactive, toolbar, timezone and datapath can be customized this way for more information.

Controlling axes properties

The [axes](#) and [subplot](#) commands return the Axes instance that is created, and you can control the properties of the axis, such as ticklines, ticklabels, gridlines, etc, using the set method of the [Axis](#) API. Or if you prefer matlab handle graphics commands, you can use the [set](#) command to set properties.

```
subplot(111)
plot([1,2,3])
glines = getp(gca(), 'gridlines')
# make the gridlines blue and thicker
setp(glines, 'color', 'b', 'linewidth', 2)

# get the patches.Rectangle instance
# increase the linewidth of the rectangular axis frame
frame = gca(gca(), 'frame')
setp(frame, 'linewidth', 2)
```

One thing that comes up a lot in my plots when I have multiple subplots is that I don't want the xticklabels on all except the lowest subplot, if the scaling is the same. Here's how to handle graphics; recall that [gca](#) returns a handle to the current axis

```
subplot(211)
plot([1,2,3], [1,2,3])
setp(gca(), xticklabels=[])

subplot(212)
plot([1,2,3], [1,4,9])
```

and the same with instance methods

```
a1 = subplot(211)
plot([1,2,3], [1,2,3])
a1.set_xticklabels([])

subplot(212)
plot([1,2,3], [1,4,9])
```

Using the interactive shell

You can work with matplotlib interactively from the python shell. The recommended option is [ipython](#), which includes explicit support for all of the matplotlib backends and the `pylab` option.

For other interactive environments, there are known incompatibilities with some of the IDEs, because they use different GUI event handlers. If you want to use an IDE, please consult the backends documentation for compatibility information. In an unmodified python shell, you must use the TkAgg backend. To work interactively,

shells or IDEs, please consult [backends](#) and [interactive](#).

Event handling

When visualizing data, it's often helpful to get some interactive input from the user. GUIs provide event handling to determine things like key presses, mouse button clicks. matplotlib supports a number of GUIs, and provides an interface to handling via the [connect](#) and [disconnect](#) methods of the pylab interface. API users can use their GUIs event handling directly, but do have the option of using their `FigureCanvas.mpl_connect` method.

matplotlib uses a callback event handling mechanism. The basic idea is that you register what you want to listen for, and the figure canvas will call a user defined function when that event occurs. For example, if you want to know where the user clicks a mouse on your figure, you can define a function

```
def click(event):
    print 'you clicked', event.x, event.y

#register this function with the event handler
connect('button_press_event', click)
```

Then whenever the user clicks anywhere on the figure canvas, your function will be called with an [MplEvent](#) instance. The event instance will have the following attributes defined.

Property	Meaning
x	x position - pixels from left of canvas
y	y position - pixels from bottom of canvas
button	button pressed None, 1, 2, 3
inaxes	the Axes instance if mouse is over axes (or None)
xdata	x coord of mouse in data coords (None if mouse isn't over axes)
ydata	y coord of mouse in data coords (None if mouse isn't over axes)
name	The string name of the event
canvas	The FigureCanvas instance the event occurred in
key	The key press if any, eg 'a', 'b', '1'. Also records 'control' and 'shift'

You can connect to the following events: 'button_press_event', 'button_release_event', 'motion_notify_event'. Here's an example to get the mouse location in data coordinates when the mouse moves

```
from pylab import *

plot(arange(10))

def on_move(event):
    # get the x and y pixel coords
    x, y = event.x, event.y
```

```

    if event.inaxes:
        print 'data coords', event.xdata, event.ydata

connect('motion_notify_event', on_move)

show()

```

Plotting dates

matplotlib handles date plotting by providing converter functions to convert python datetime objects to and from floating point numbers. These numbers represent days in the Proleptic Gregorian (UTC) time. The advantage of using UTC is that there is no daylight savings time (which facilitates time calculation). However, this is just the internal format matplotlib uses. Timezone support is provided. The supported date range is the same as the python datetime range: years 0001-9999.

Note that before matplotlib 0.63, dates were represented as seconds since the epoch. This did not have timezone support. The code was rewritten to use datetime objects. This change is required to migrate old code. Also note that many of the date locator functions have the same name now but have a different (and more general) constructor syntax.

There are three datetime conversion functions: [date2num](#), [num2date](#) and [drange](#). `date2num` takes a python datetime (or sequence of datetimes) and returns a float or sequence of floats. `num2date` is the inverse. `drange` takes a start date, end date and datetime timedelta object and returns a sequence of floats.

```

date1 = datetime.date( 1952, 1, 1 )
date2 = datetime.date( 2004, 4, 12 )
delta = datetime.timedelta(days=100)
dates = drange(date1, date2, delta)

```

The function [plot_date](#) is used to plot a series of y-values against floating point dates. If you have just a sequence of floats, you could just as easily use [plot](#). In fact, this is just what `plot_date` does under the hood. The only difference is that `plot_date` picks a date tick locator and tries to make an intelligent default choice based on the range of dates present. If you want custom date tickers and formatters, you can just as easily use the `plot` command with

```

# make up some random y values
s = rand(len(dates))
plot_date(dates, s)

```

date ticking with dateutils

Another new change in date plotting with matplotlib-0.63 is that the powerful [dateutil](#) package is included to support date tick locating. `dateutil` is included with the matplotlib distribution and installed automatically if you don't already have it. The `dateutil` package provides a powerful set of tools for generating recurrence rules. Thus you can place ticks every Monday at 5PM, every second Tuesday at specified hours, weekdays etc. All of matplotlib's date tick locators utilize `dateutil` (see [matplotlib.ticker](#) and [matplotlib.dates](#) for more information on date ticking).

Here is an example setting up a ticker to make major ticks every year, minor ticks

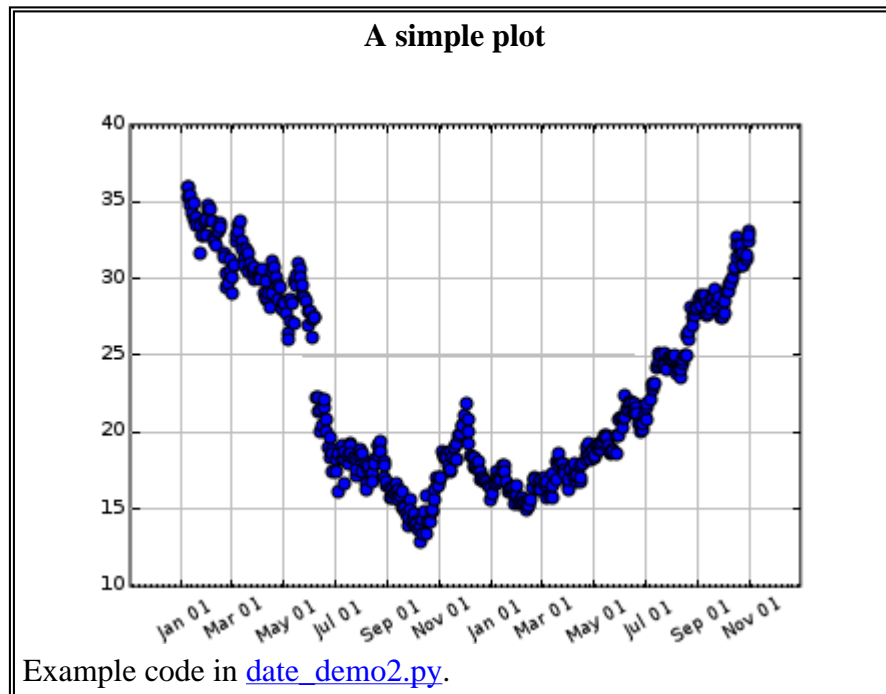
```
years      = YearLocator()    # every year
months     = MonthLocator()  # every month
yearsFmt   = DateFormatter('%Y')
ax.xaxis.set_major_locator(years)
ax.xaxis.set_major_formatter(yearsFmt)
ax.xaxis.set_minor_locator(months)
```

The [DateFormatter](#) class takes an [strftime](#) format string. The example above used constructors for [YearLocator](#) and [MonthLocator](#). But you can customize this behavior to make major ticks every 5 years, and minor ticks on March, June, September and December instead

```
# every 5 years
years      = YearLocator(5)

# every quarter
months     = MonthLocator(range(3,13,3))
```

The date tick locators [MinuteLocator](#), [HourLocator](#), [DayLocator](#), [WeekdayLocator](#), [YearLocator](#) are provided, and are simple interfaces to the [rrule](#) generator. See [date_demo2.py](#). For full control of date ticking, you can use your own [rrule](#) with [rrule](#) class. See [date_demo_rrule.py](#).



timezones with pytz

Timezone support is provided via the [pytz](#) module. This module provides python implementations of 546 of the Olsen database timezones. [pytz](#) is included in the [r](#) and will be installed if you don't already have it. Thus you can reside in Zimbabwe to plot financial quotes in London or New York, with correct handling of idiosyncratic

savings time. The default timezone is set by the `timezone` parameter in your `matplotlibrc` file. If a `timezone` is specified as a string acceptable to the `pytz.timezone` constructor, eg `'UTC'` or `'Europe/London'`. The default timezone in the `matplotlibrc` file that ships with matplotlib is `'UTC'`. matplotlib supports per directory `matplotlibrc` files, so if you have a project in a timezone other than the default you can place an `matplotlibrc` file in a working directory for that project to use that as the default.

All of the matplotlib date ticking classes and functions, except for `Drange` and `Date2Num`, have an optional argument `tz` which is a `tzinfo` instance; the reason `Drange` and `Date2Num` don't have this is because they can infer it from the `tzinfo` of the supplied `datetime` instances. If `tz` is not specified, the `matplotlibrc` value will be used. See [date_demo_convert.py](#) for an example date plot using `Drange` and `Date2Num`.

Matlab® is a registered trademark of The MathWorks

Powered by
[YAPTU!](#)