LEARNING matplotlib

#matplotlib

Table of Contents

About

Chapter 1: Getting started with matplotlib

Remarks

Overview

matplotlib is a plotting library for Python. It provides object-oriented APIs for embedding plots into applications. It is similar to MATLAB in capacity and syntax.

It was originally written by J.D.Hunter and is actively being developed. It is distributed under a BSD-Style License.

Versions

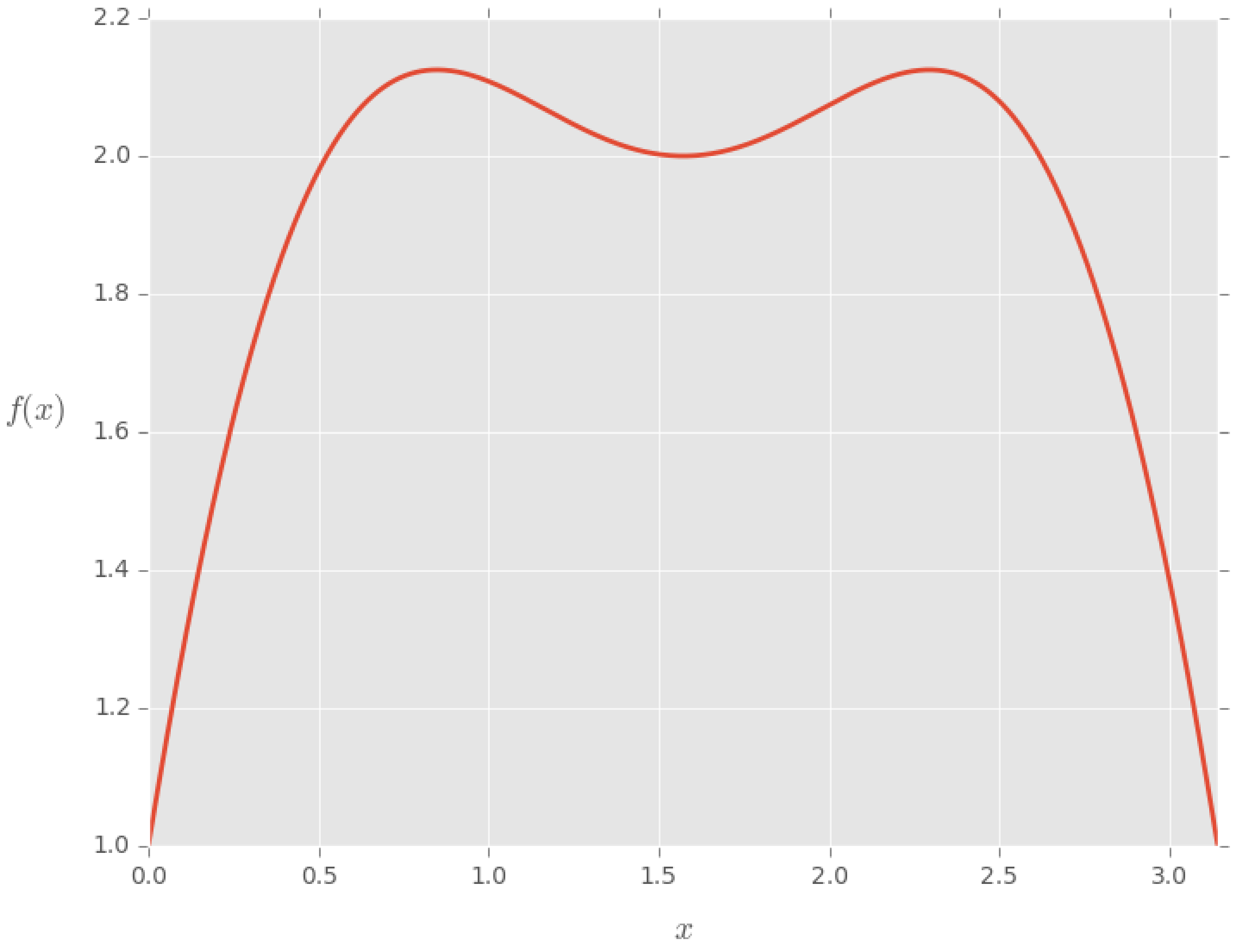
Version	Python Versions Supported	Remarks	Release Date
1.3.1	2.6, 2.7, 3.x	Older Stable Version	2013-10-10
1.4.3	2.6, 2.7, 3.x	Previous Stable Version	2015-07-14
1.5.3	2.7, 3.x	Current Stable Version	2016-01-11
2.x	2.7, 3.x	Latest Development Version	2016-07-25

Examples

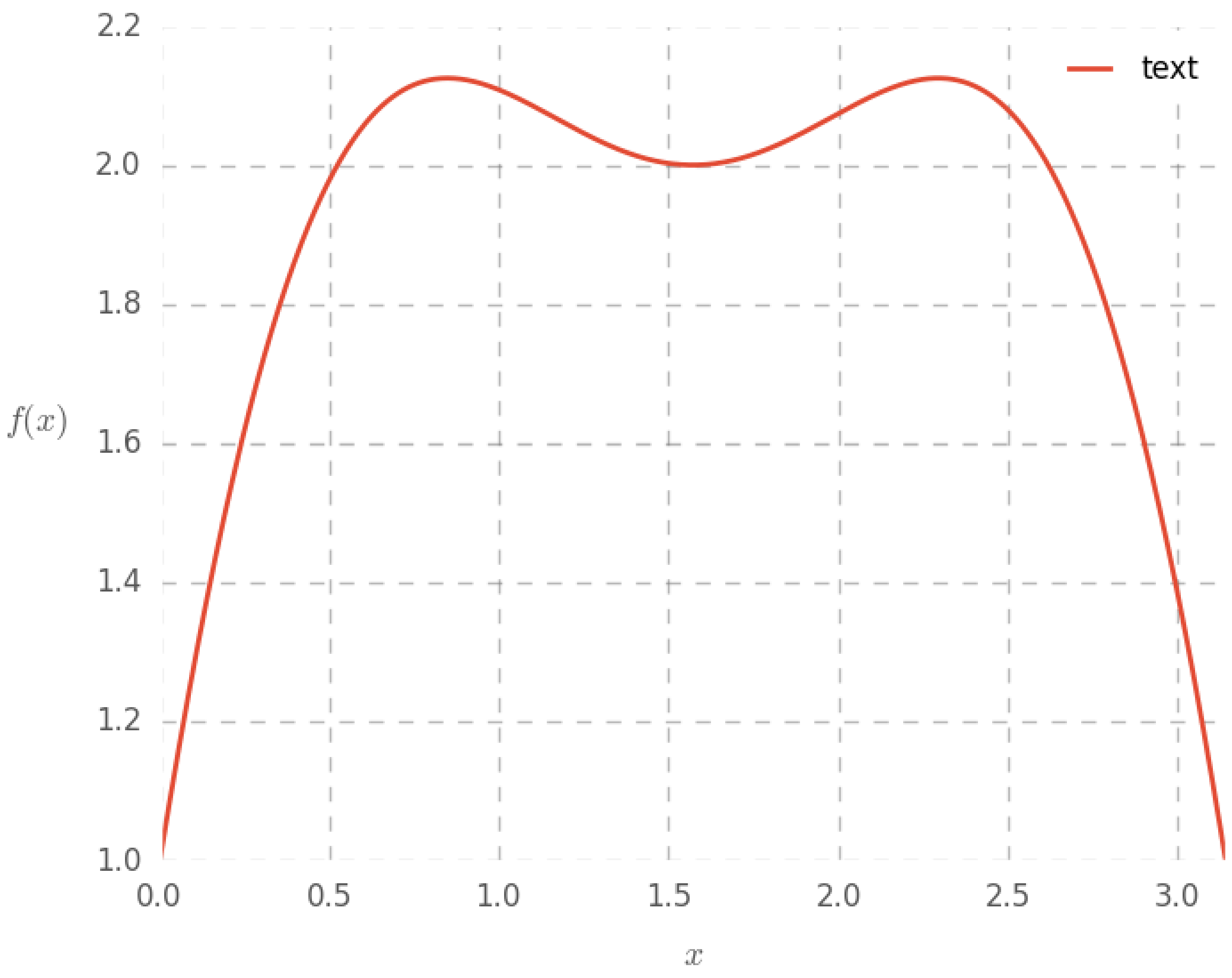
Customizing a matplotlib plot

```
import pylab as plt
import numpy as np

plt.style.use('ggplot')

fig = plt.figure(1)
ax = plt.gca()

# make some testing data
x = np.linspace( 0, np.pi, 1000 )
test_f = lambda x: np.sin(x)*3 + np.cos(2*x)

# plot the test data
ax.plot( x, test_f(x) , lw = 2)

# set the axis labels
ax.set_xlabel(r'$x$', fontsize=14, labelpad=10)
ax.set_ylabel(r'$f(x)$', fontsize=14, labelpad=25, rotation=0)
```

```
# set axis limits
ax.set_xlim(0,np.pi)

plt.draw()
```



```
# Customize the plot
ax.grid(1, ls='--', color='#777777', alpha=0.5, lw=1)
ax.tick_params(labelsize=12, length=0)
ax.set_axis_bgcolor('w')
# add a legend
leg = plt.legend( ['text'], loc=1 )
fr = leg.get_frame()
fr.set_facecolor('w')
fr.set_alpha(.7)
plt.draw()
```



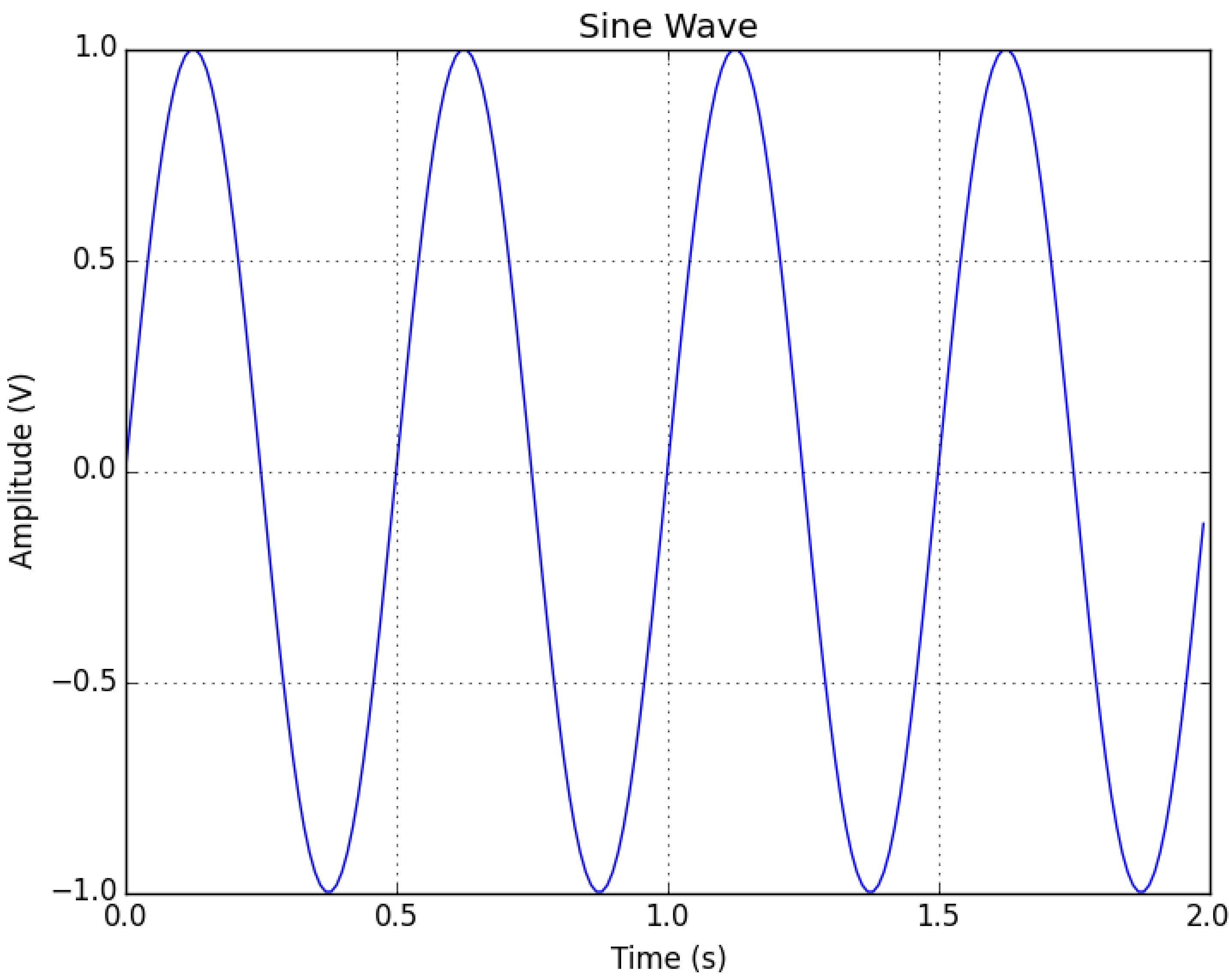
Imperative vs. Object-oriented Syntax

Matplotlib supports both object-oriented and imperative syntax for plotting. The imperative syntax is intentionally designed to be very close to Matlab syntax.

The imperative syntax (sometimes called 'state-machine' syntax) issues a string of commands all of which act on the most recent figure or axis (like Matlab). The object-oriented syntax, on the other hand, explicitly acts on the objects (figure, axis, etc.) of interest. A key point in the zen of Python states that explicit is better than implicit so the object-oriented syntax is more pythonic. However, the imperative syntax is convenient for new converts from Matlab and for writing small, "throwaway" plot scripts. Below is an example of the two different styles.

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

t = np.arange(0, 2, 0.01)
y = np.sin(4 * np.pi * t)

# Imperative syntax
plt.figure(1)
plt.clf()
plt.plot(t, y)
plt.xlabel('Time (s)')
plt.ylabel('Amplitude (V)')
plt.title('Sine Wave')
plt.grid(True)

# Object oriented syntax
fig = plt.figure(2)
fig.clf()
ax = fig.add_subplot(1,1,1)
ax.plot(t, y)
ax.set_xlabel('Time (s)')
ax.set_ylabel('Amplitude (V)')
ax.set_title('Sine Wave')
ax.grid(True)
```

Both examples produce the same plot which is shown below.



Installation and Setup

There are several ways to go about installing matplotlib, some of which will depend on the system you are using. If you are lucky, you will be able to use a package manager to easily install the matplotlib module and its dependencies.

Windows

On Windows machines you can try to use the pip package manager to install matplotlib. See here for information on setting up pip in a Windows environment.

OS X

It is recommended that you use the pip package manager to install matplotlib. If you need to install some of the non-Python libraries on your system (e.g. `libfreetype`) then consider using homebrew.

If you cannot use pip for whatever reason, then try to install from source.

Linux

Ideally, the system package manager or pip should be used to install matplotlib, either by installing the `python-matplotlib` package or by running `pip install matplotlib`.

If this is not possible (e.g. you do not have sudo privileges on the machine you are using), then you can install from source using the `--user` option: `python setup.py install --user`. Typically, this will install matplotlib into `~/.local`.

Debian/Ubuntu

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

Fedora/Red Hat

```
sudo yum install python-matplotlib
```

Troubleshooting

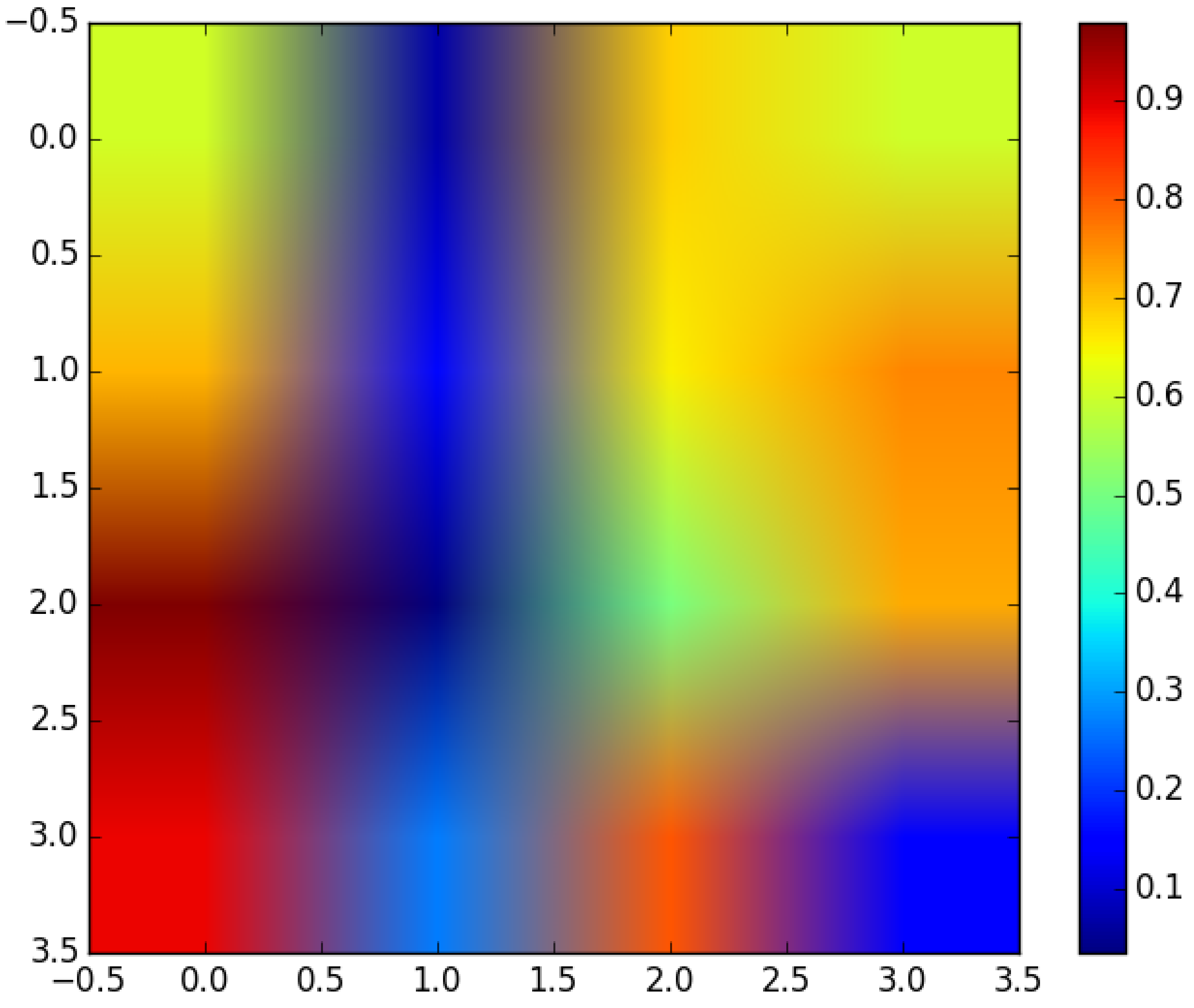
See the matplotlib website for advice on how to fix a broken matplotlib.

Two dimensional (2D) arrays

Display a two dimensional (2D) array on the axes.

```
import numpy as np
from matplotlib.pyplot import imshow, show, colorbar
```

```
image = np.random.rand(4,4)
imshow(image)
colorbar()
show()
```



Read Getting started with matplotlib online: http://www.riptutorial.com/matplotlib/topic/881/getting-started-with-matplotlib

Chapter 2: Animations and interactive plotting

Introduction

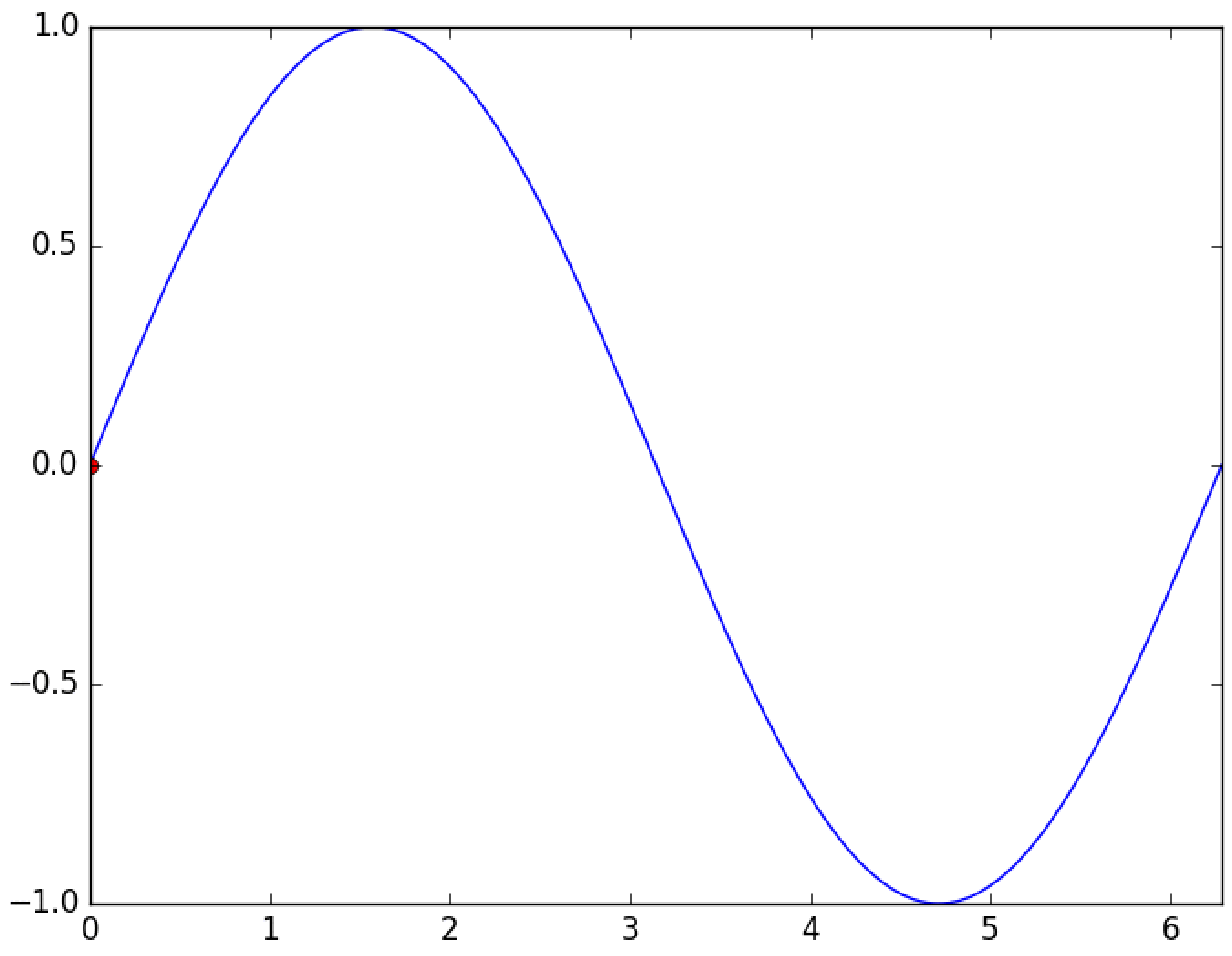
With python matplotlib you can properly make animated graphs.

Examples

Basic animation with FuncAnimation

The matplotlib.animation package offer some classes for creating animations. `FuncAnimation` creates animations by repeatedly calling a function. Here we use a function `animate()` that changes the coordinates of a point on the graph of a sine function.

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

TWOPI = 2*np.pi

fig, ax = plt.subplots()

t = np.arange(0.0, TWOPI, 0.001)
s = np.sin(t)
l = plt.plot(t, s)

ax = plt.axis([0,TWOPI,-1,1])

redDot, = plt.plot([0], [np.sin(0)], 'ro')

def animate(i):
    redDot.set_data(i, np.sin(i))
    return redDot,

# create animation using the animate() function
myAnimation = animation.FuncAnimation(fig, animate, frames=np.arange(0.0, TWOPI, 0.1), \
                                      interval=10, blit=True, repeat=True)

plt.show()
```



Save animation to gif

In this example we use the `save` method to save an `Animation` object using ImageMagick.

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from matplotlib import rcParams

# make sure the full paths for ImageMagick and ffmpeg are configured
rcParams['animation.convert_path'] = r'C:\Program Files\ImageMagick\convert'
rcParams['animation.ffmpeg_path'] = r'C:\Program Files\ffmpeg\bin\ffmpeg.exe'

TWOPI = 2*np.pi

fig, ax = plt.subplots()

t = np.arange(0.0, TWOPI, 0.001)
s = np.sin(t)
l = plt.plot(t, s)
```

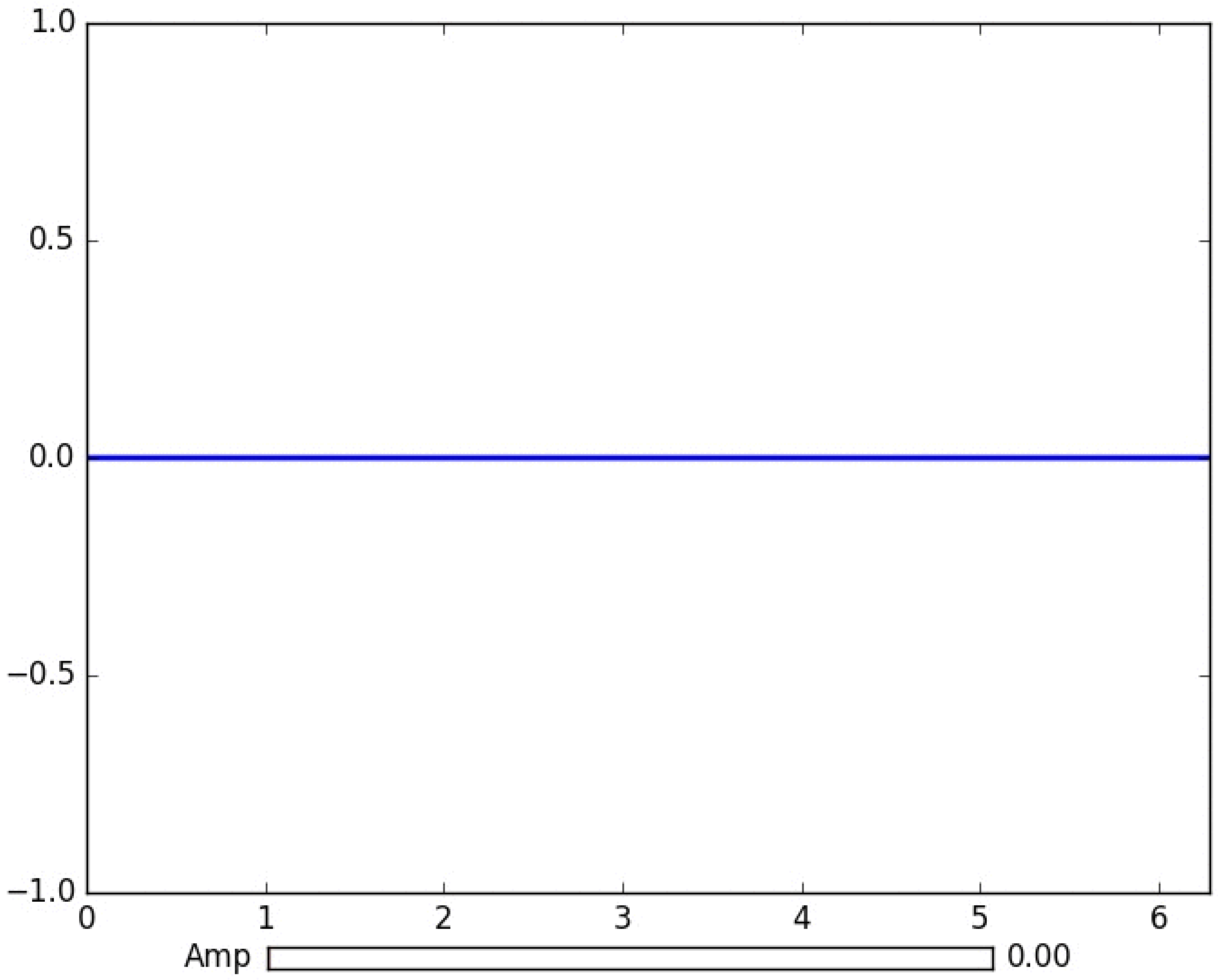
```
ax = plt.axis([0,TWOPI,-1,1])

redDot, = plt.plot([0], [np.sin(0)], 'ro')

def animate(i):
    redDot.set_data(i, np.sin(i))
    return redDot,

# create animation using the animate() function with no repeat
myAnimation = animation.FuncAnimation(fig, animate, frames=np.arange(0.0, TWOPI, 0.1), \
                                      interval=10, blit=True, repeat=False)

# save animation at 30 frames per second
myAnimation.save('myAnimation.gif', writer='imagemagick', fps=30)
```

Interactive controls with matplotlib.widgets

For interacting with plots Matplotlib offers GUI neutral widgets. Widgets require a `matplotlib.axes.Axes` object.

Here's a slider widget demo that ùpdates the amplitude of a sine curve. The update function is triggered by the slider's `on_changed()` event.

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from matplotlib.widgets import Slider

TWOPI = 2*np.pi

fig, ax = plt.subplots()

t = np.arange(0.0, TWOPI, 0.001)
initial_amp = .5
s = initial_amp*np.sin(t)
l, = plt.plot(t, s, lw=2)

ax = plt.axis([0,TWOPI,-1,1])

axamp = plt.axes([0.25, .03, 0.50, 0.02])
# Slider
samp = Slider(axamp, 'Amp', 0, 1, valinit=initial_amp)

def update(val):
    # amp is the current value of the slider
    amp = samp.val
    # update curve
    l.set_ydata(amp*np.sin(t))
    # redraw canvas while idle
    fig.canvas.draw_idle()

# call update function on slider value change
samp.on_changed(update)

plt.show()
```



Other available widgets:

- AxesWidget
- Button
- CheckButtons
- Cursor
- EllipseSelector
- Lasso
- LassoSelector
- LockDraw
- MultiCursor
- RadioButtons
- RectangleSelector
- SpanSelector
- SubplotTool
- ToolHandles

Plot live data from pipe with matplotlib

This can be usefull when you want to visualize incoming data in real-time. This data could, for example, come from a microcontroller that is continuously sampling an analog signal.

In this example we will get our data from a named pipe (also known as a fifo). For this example, the data in the pipe should be numbers separted by newline characters, but you can adapt this to your liking.

Example data:

```
100
123.5
1589
```

More information on named pipes

We will also be using the datatype deque, from the standard library collections. A deque object works quite a lot like a list. But with a deque object it is quite easy to append something to it while still keeping the deque object at a fixed length. This allows us to keep the x axis at a fixed length instead of always growing and squishing the graph together. More information on deque objects

Choosing the right backend is vital for performance. Check what backends work on your operating system, and choose a fast one. For me only qt4agg and the default backend worked, but the default one was too slow. More information on backends in matplotlib

This example is based on the matplotlib example of plotting random data.

None of the ' characters in this code are meant to be removed.

```
import matplotlib
import collections
#selecting the right backend, change qt4agg to your desired backend
matplotlib.use('qt4agg')
import matplotlib.pyplot as plt
import matplotlib.animation as animation

#command to open the pipe
datapipe = open('path to your pipe','r')

#amount of data to be displayed at once, this is the size of the x axis
#increasing this amount also makes plotting slightly slower
data_amount = 1000

#set the size of the deque object
datalist = collections.deque([0]*data_amount,data_amount)

#configure the graph itself
fig, ax = plt.subplots()
line, = ax.plot([0,]*data_amount)

#size of the y axis is set here
ax.set_ylim(0,256)

def update(data):
        line.set_ydata(data)
        return line,
def data_gen():
    while True:
        """
        We read two data points in at once, to improve speed
        You can read more at once to increase speed
        Or you can read just one at a time for improved animation smoothness
        data from the pipe comes in as a string,
        and is seperated with a newline character,
        which is why we use respectively eval and rstrip.
        """
        datalist.append(eval((datapipe.readline()).rstrip('\n')))
        datalist.append(eval((datapipe.readline()).rstrip('\n')))
        yield datalist

ani = animation.FuncAnimation(fig,update,data_gen,interval=0, blit=True)
plt.show()
```

If your plot starts to get delayed after a while, try adding more of the datalist.append data, so that more lines get read each frame. Or choose a faster backend if you can.

This worked with 150hz data from a pipe on my 1.7ghz i3 4005u.

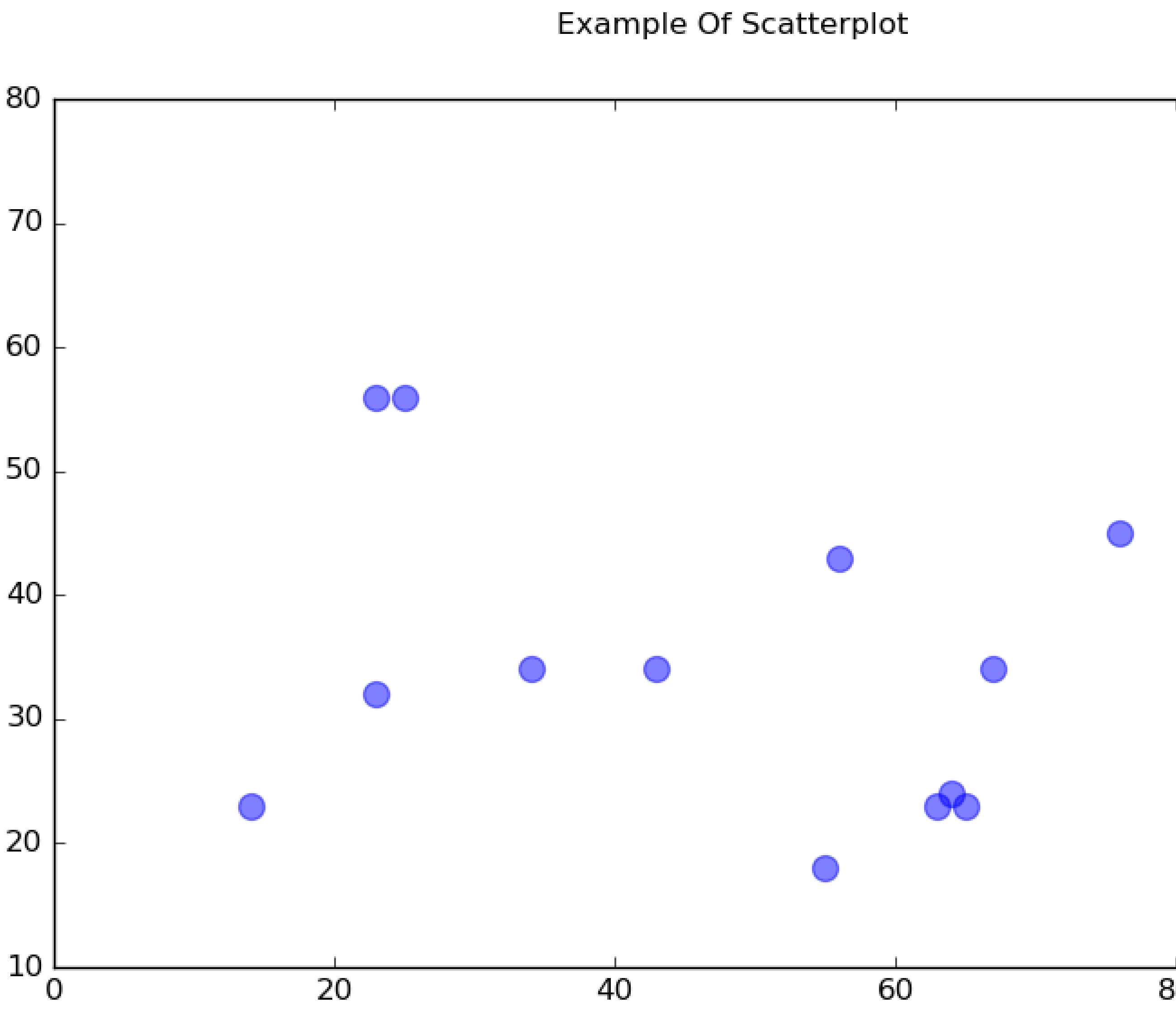
Read Animations and interactive plotting online:
http://www.riptutorial.com/matplotlib/topic/6983/animations-and-interactive-plotting

Chapter 3: Basic Plots

Examples

Scatter Plots

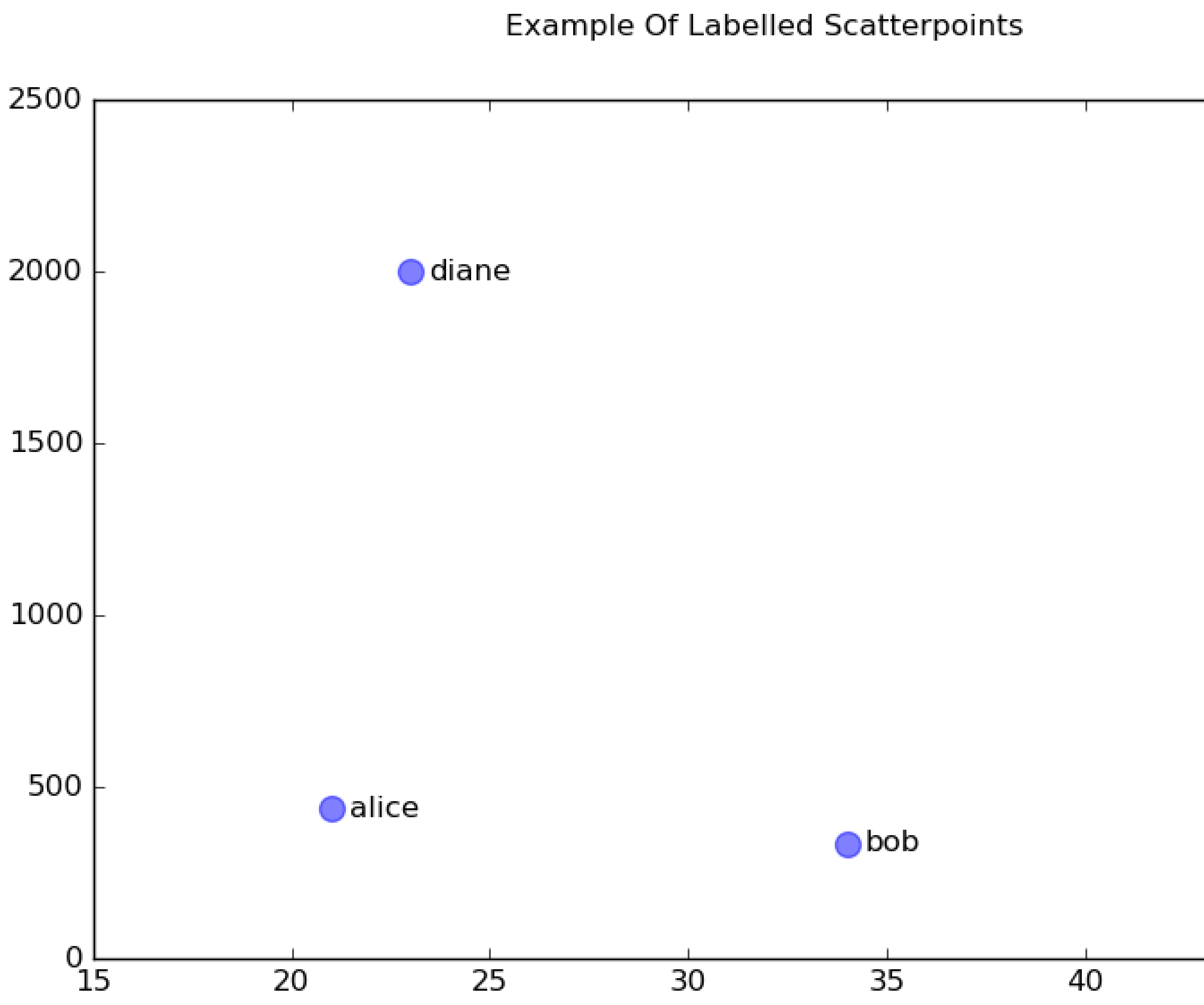
A simple scatter plot



```
import matplotlib.pyplot as plt

# Data
x = [43,76,34,63,56,82,87,55,64,87,95,23,14,65,67,25,23,85]
y = [34,45,34,23,43,76,26,18,24,74,23,56,23,23,34,56,32,23]

fig, ax = plt.subplots(1, figsize=(10, 6))
fig.suptitle('Example Of Scatterplot')
```

```
# Create the Scatter Plot
ax.scatter(x, y,
            color="blue",    # Color of the dots
            s=100,           # Size of the dots
            alpha=0.5,       # Alpha/transparency of the dots (1 is opaque, 0 is transparent)
            linewidths=1)    # Size of edge around the dots

# Show the plot
plt.show()
```

A Scatterplot with Labelled Points

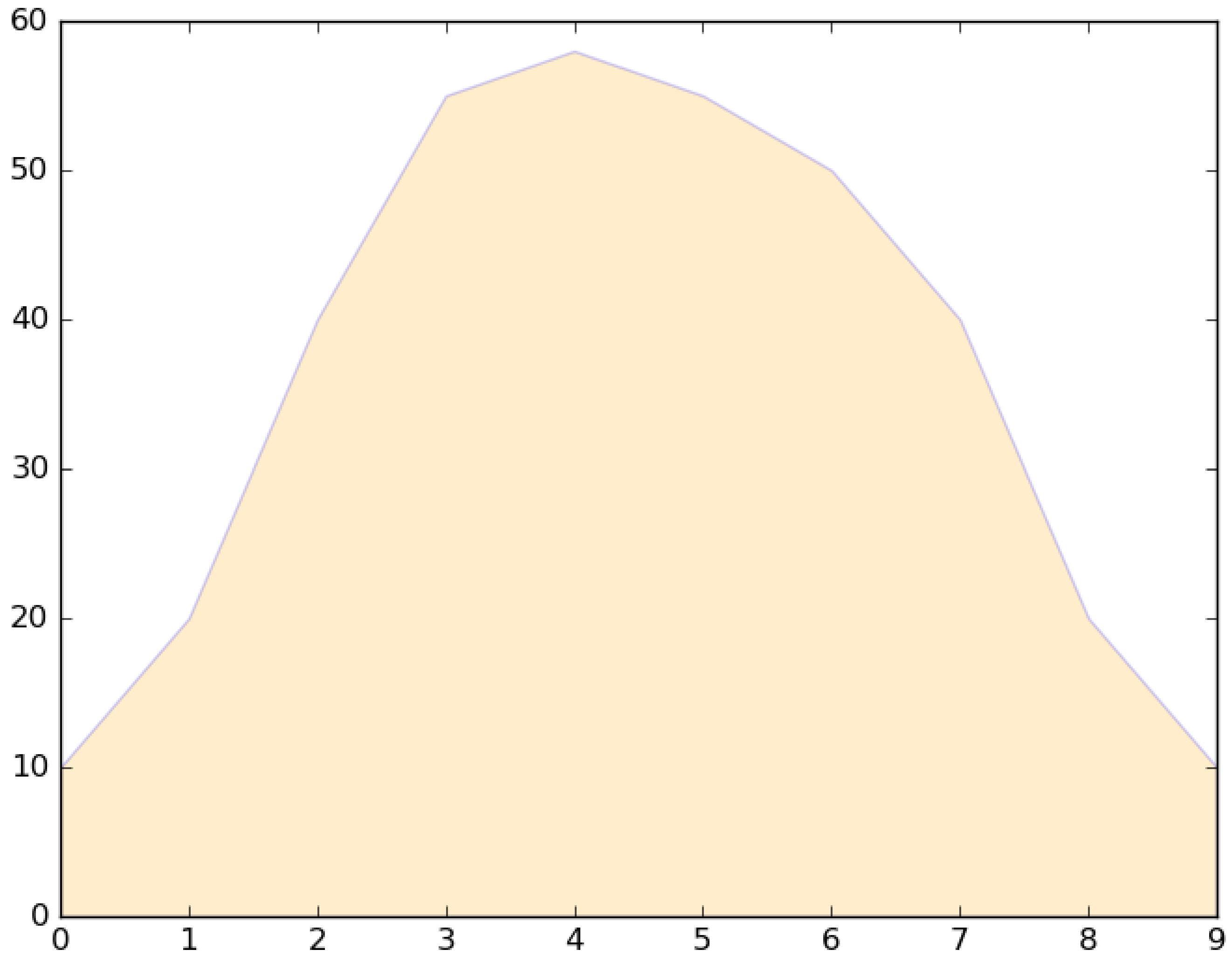


```
import matplotlib.pyplot as plt

# Data
x = [21, 34, 44, 23]
y = [435, 334, 656, 1999]
labels = ["alice", "bob", "charlie", "diane"]

# Create the figure and axes objects
```

```
fig, ax = plt.subplots(1, figsize=(10, 6))
fig.suptitle('Example Of Labelled Scatterpoints')

# Plot the scatter points
ax.scatter(x, y,
           color="blue",  # Color of the dots
           s=100,         # Size of the dots
           alpha=0.5,     # Alpha of the dots
           linewidths=1)  # Size of edge around the dots

# Add the participant names as text labels for each point
for x_pos, y_pos, label in zip(x, y, labels):
    ax.annotate(label,             # The label for this point
                xy=(x_pos, y_pos), # Position of the corresponding point
                xytext=(7, 0),     # Offset text by 7 points to the right
                textcoords='offset points', # tell it to use offset points
                ha='left',         # Horizontally aligned to the left
                va='center')       # Vertical alignment is centered

# Show the plot
plt.show()
```

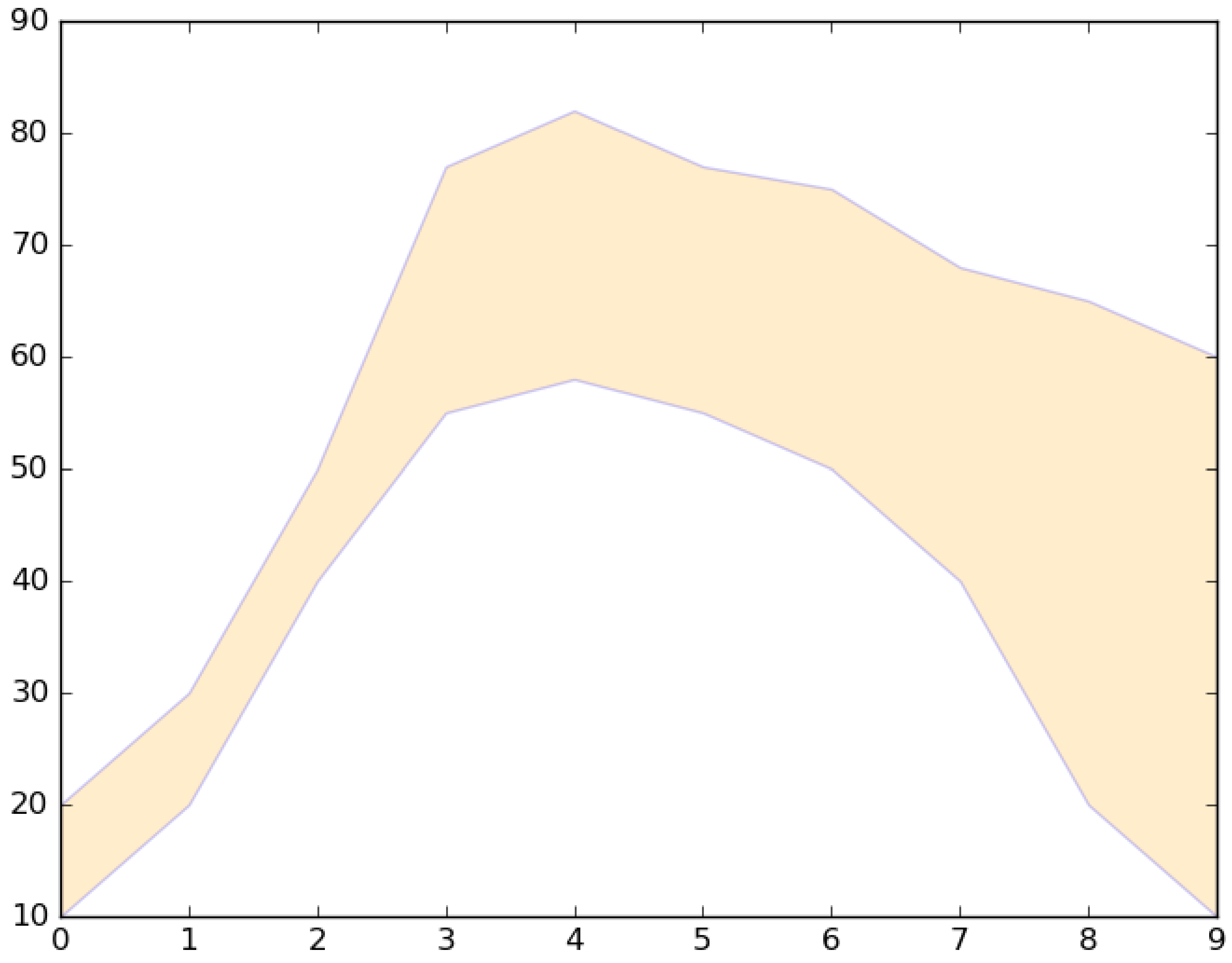
Shaded Plots

Shaded region below a line



```
import matplotlib.pyplot as plt

# Data
x =  [0,1,2,3,4,5,6,7,8,9]
y1 = [10,20,40,55,58,55,50,40,20,10]

# Shade the area between y1 and line y=0
plt.fill_between(x, y1, 0,
                 facecolor="orange", # The fill color
                 color='blue',       # The outline color
                 alpha=0.2)          # Transparency of the fill

# Show the plot
plt.show()
```

Shaded Region between two lines

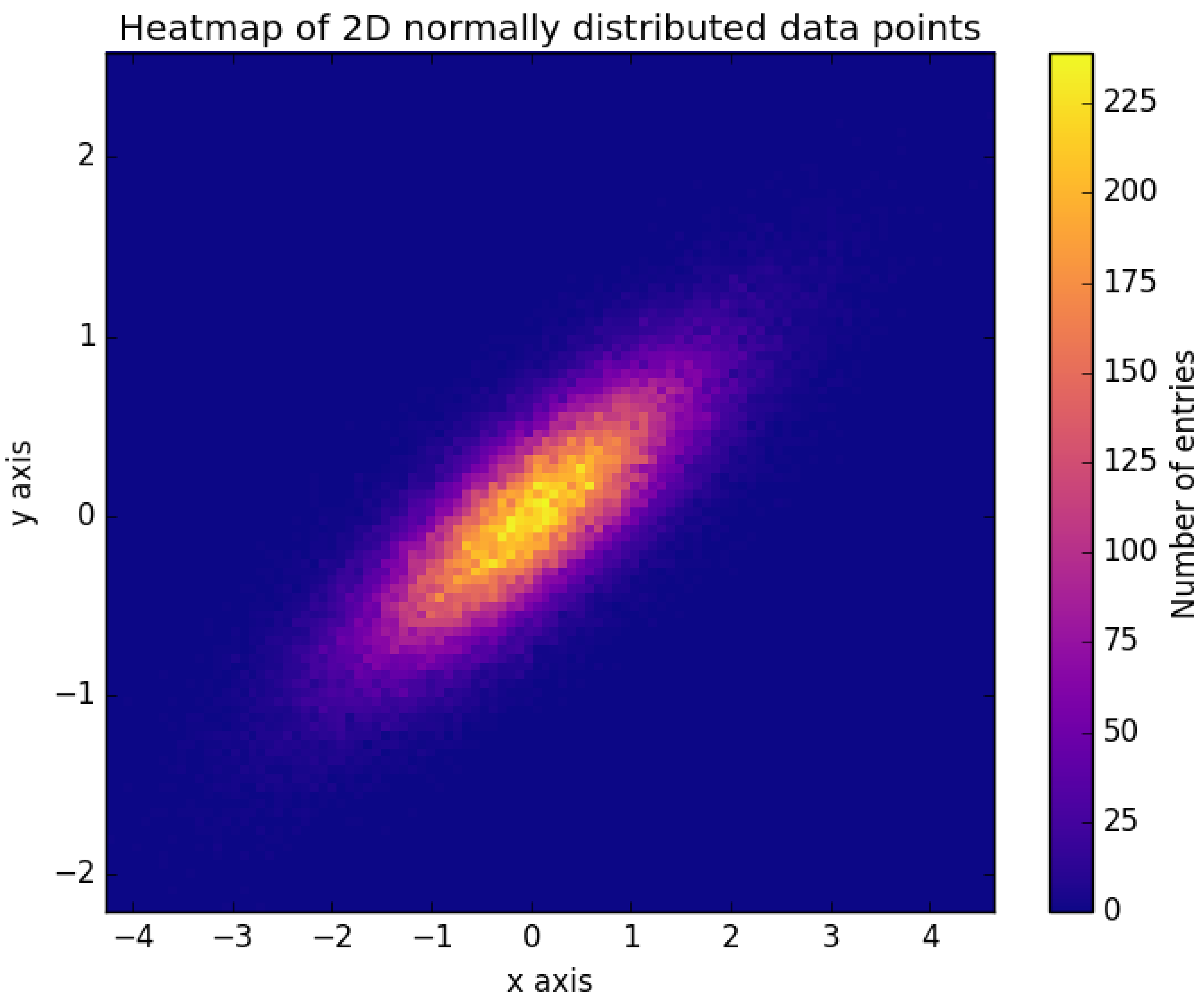


```
import matplotlib.pyplot as plt

# Data
x =  [0,1,2,3,4,5,6,7,8,9]
y1 = [10,20,40,55,58,55,50,40,20,10]
y2 = [20,30,50,77,82,77,75,68,65,60]

# Shade the area between y1 and y2
plt.fill_between(x, y1, y2,
                 facecolor="orange", # The fill color
                 color='blue',       # The outline color
                 alpha=0.2)          # Transparency of the fill

# Show the plot
plt.show()
```

Heatmap

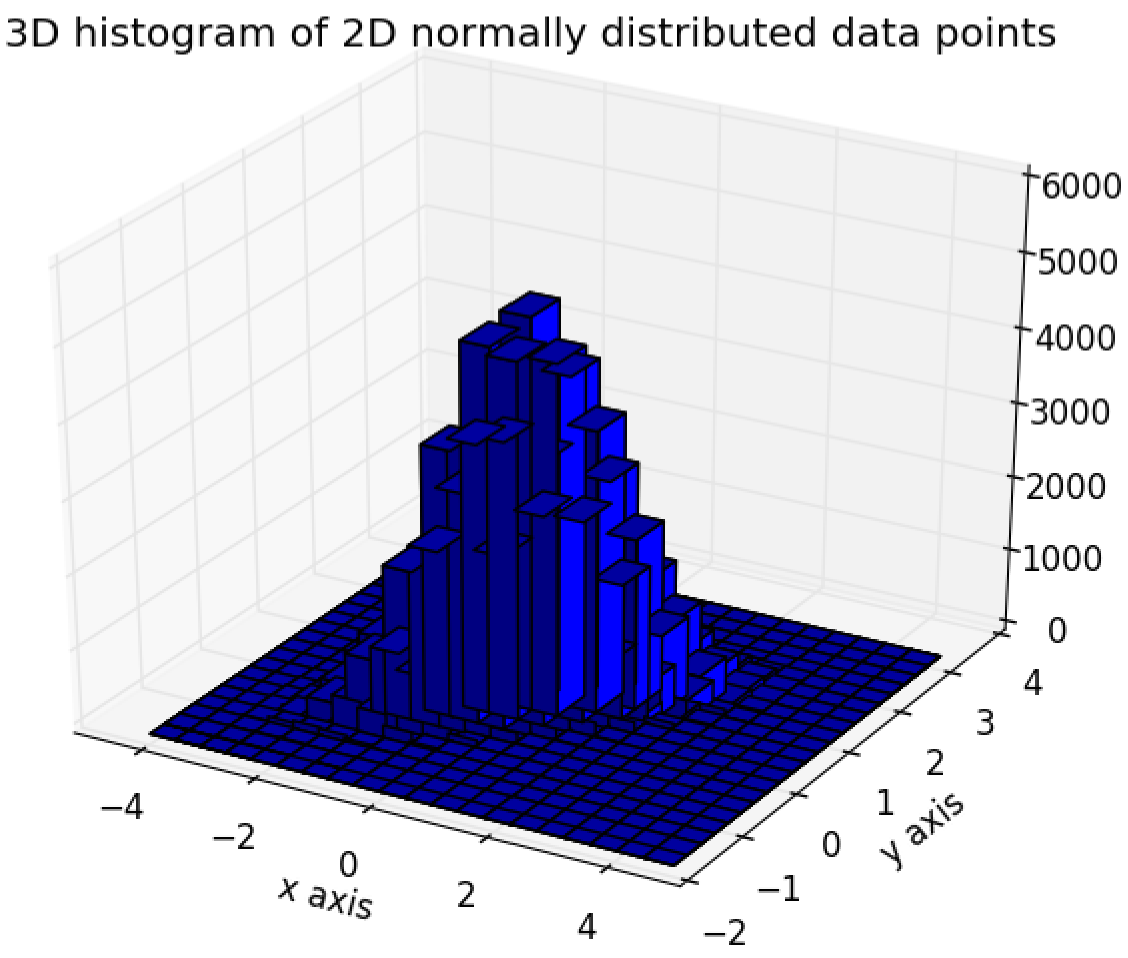
Heatmaps are useful for visualizing scalar functions of two variables. They provide a “flat” image of two-dimensional histograms (representing for instance the density of a certain area).

The following source code illustrates heatmaps using bivariate normally distributed numbers centered at 0 in both directions (means `[0.0, 0.0]`) and a with a given covariance matrix. The data is generated using the numpy function numpy.random.multivariate_normal; it is then fed to the `hist2d` function of pyplot matplotlib.pyplot.hist2d.



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

# Define numbers of generated data points and bins per axis.
N_numbers = 100000
N_bins = 100

# set random seed
np.random.seed(0)

# Generate 2D normally distributed numbers.
x, y = np.random.multivariate_normal(
        mean=[0.0, 0.0],      # mean
        cov=[[1.0, 0.4],
             [0.4, 0.25]],    # covariance matrix
        size=N_numbers
        ).T                    # transpose to get columns


# Construct 2D histogram from data using the 'plasma' colormap
plt.hist2d(x, y, bins=N_bins, normed=False, cmap='plasma')

# Plot a colorbar with label.
cb = plt.colorbar()
cb.set_label('Number of entries')

# Add title and labels to plot.
plt.title('Heatmap of 2D normally distributed data points')
plt.xlabel('x axis')
plt.ylabel('y axis')

# Show the plot.
plt.show()
```

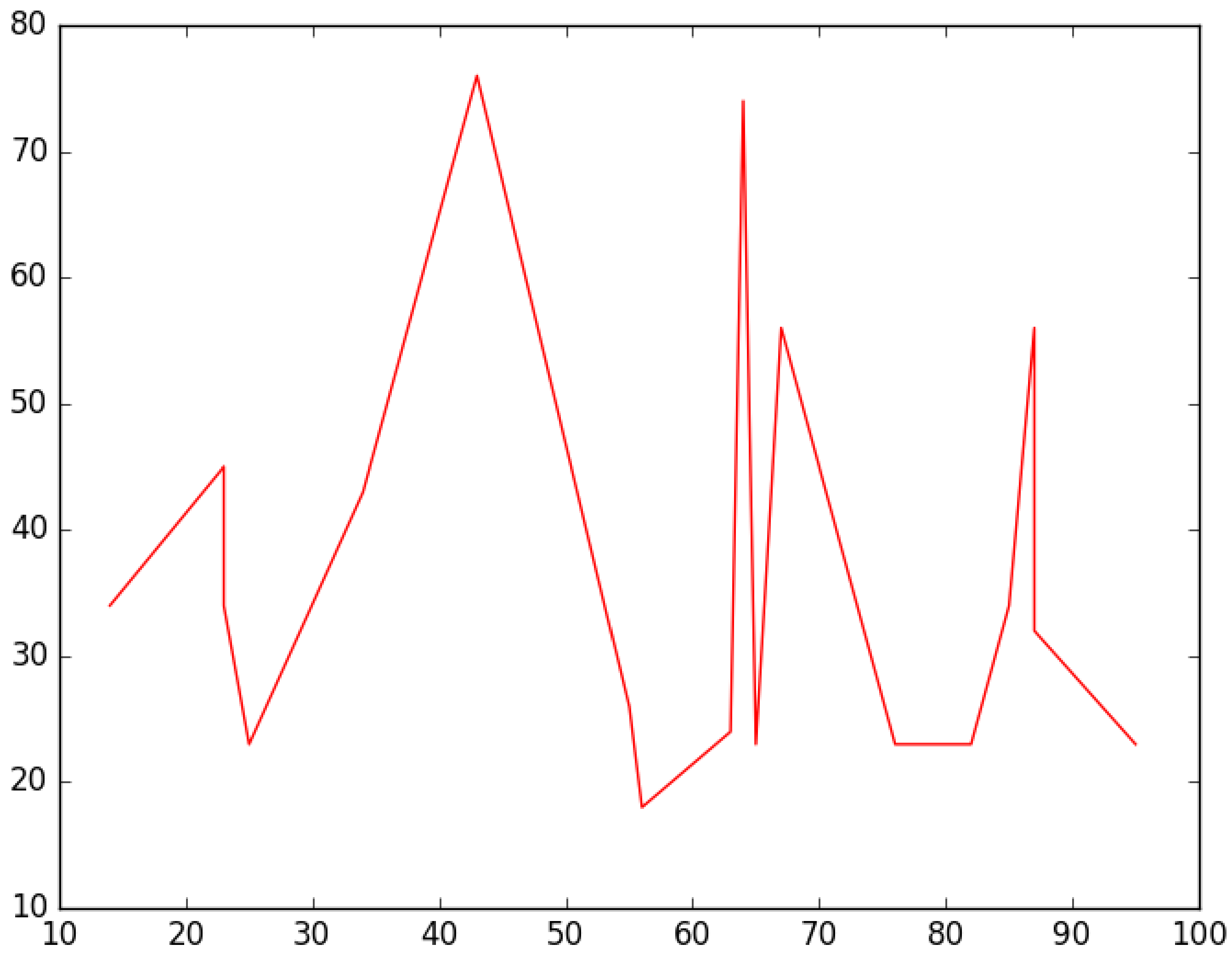
Here is the same data visualized as a 3D histogram (here we use only 20 bins for efficiency). The code is based on this matplotlib demo.



```
from mpl_toolkits.mplot3d import Axes3D
import numpy as np
import matplotlib
import matplotlib.pyplot as plt

# Define numbers of generated data points and bins per axis.
N_numbers = 100000
N_bins = 20

# set random seed
np.random.seed(0)

# Generate 2D normally distributed numbers.
x, y = np.random.multivariate_normal(
        mean=[0.0, 0.0],      # mean
        cov=[[1.0, 0.4],
             [0.4, 0.25]],    # covariance matrix
        size=N_numbers
        ).T                   # transpose to get columns

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
hist, xedges, yedges = np.histogram2d(x, y, bins=N_bins)

# Add title and labels to plot.
plt.title('3D histogram of 2D normally distributed data points')
plt.xlabel('x axis')
plt.ylabel('y axis')

# Construct arrays for the anchor positions of the bars.
# Note: np.meshgrid gives arrays in (ny, nx) so we use 'F' to flatten xpos,
# ypos in column-major order. For numpy >= 1.7, we could instead call meshgrid
# with indexing='ij'.
xpos, ypos = np.meshgrid(xedges[:-1] + 0.25, yedges[:-1] + 0.25)
xpos = xpos.flatten('F')
ypos = ypos.flatten('F')
zpos = np.zeros_like(xpos)

# Construct arrays with the dimensions for the 16 bars.
dx = 0.5 * np.ones_like(zpos)
dy = dx.copy()
dz = hist.flatten()

ax.bar3d(xpos, ypos, zpos, dx, dy, dz, color='b', zsort='average')

# Show the plot.
plt.show()
```

Line plots

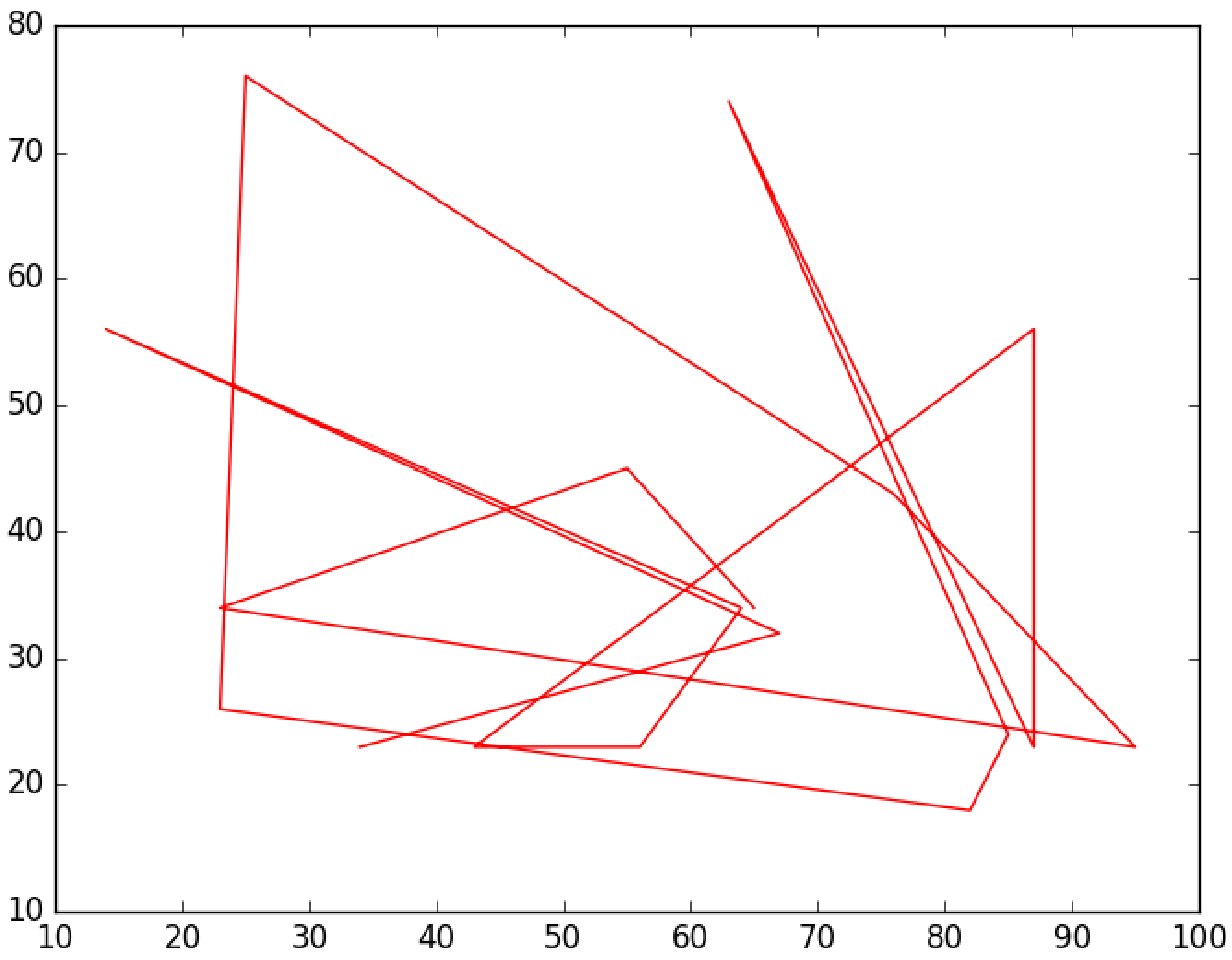
Simple line plot



```
import matplotlib.pyplot as plt

# Data
x = [14,23,23,25,34,43,55,56,63,64,65,67,76,82,85,87,87,95]
y = [34,45,34,23,43,76,26,18,24,74,23,56,23,23,34,56,32,23]

# Create the plot
plt.plot(x, y, 'r-')
# r- is a style code meaning red solid line

# Show the plot
plt.show()
```

Note that in general `y` is not a function of `x` and also that the values in `x` do not need to be sorted. Here's how a line plot with unsorted x-values looks like:

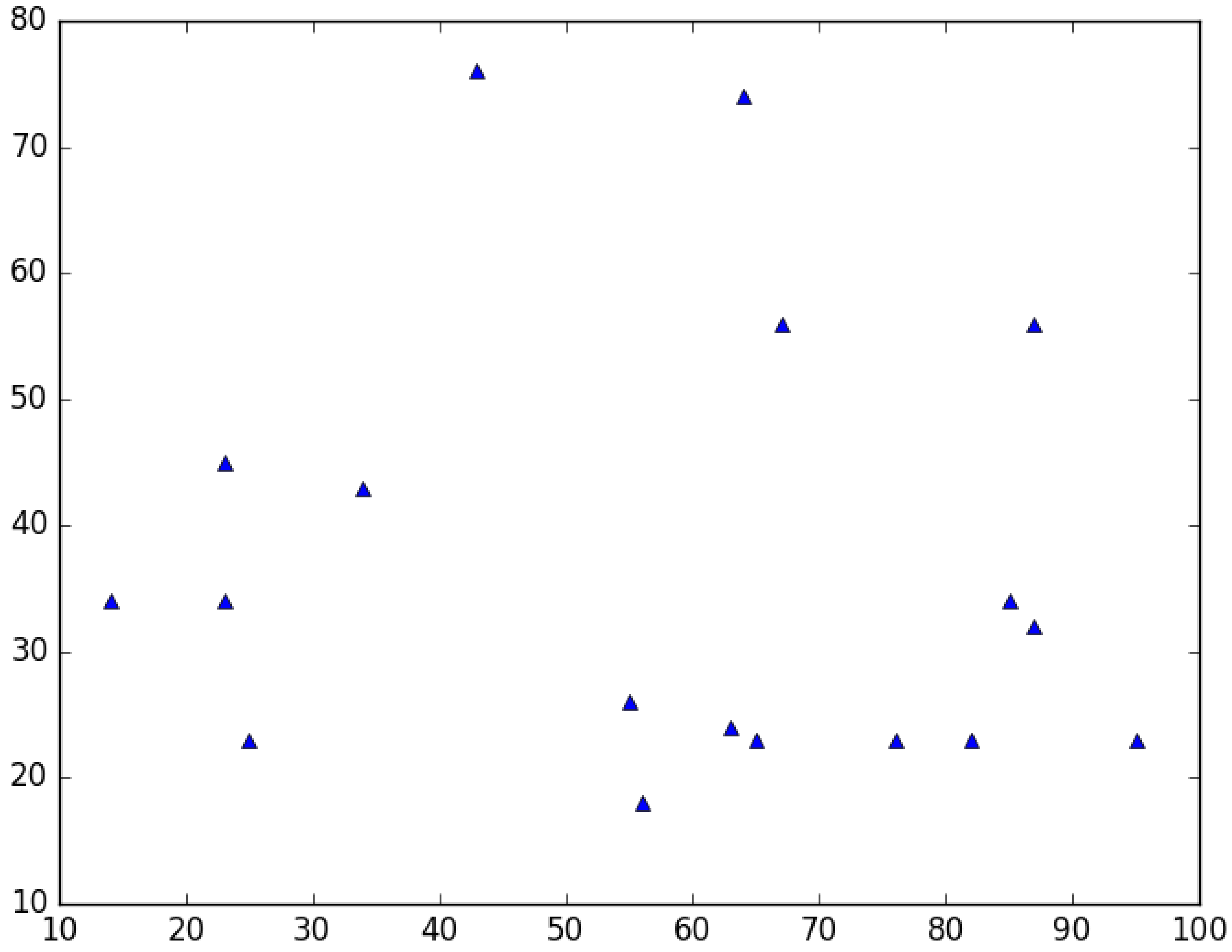
```
# shuffle the elements in x
np.random.shuffle(x)
plt.plot(x, y, 'r-')
plt.show()
```



Data plot

This is similar to a scatter plot, but uses the `plot()` function instead. The only difference in the code here is the style argument.

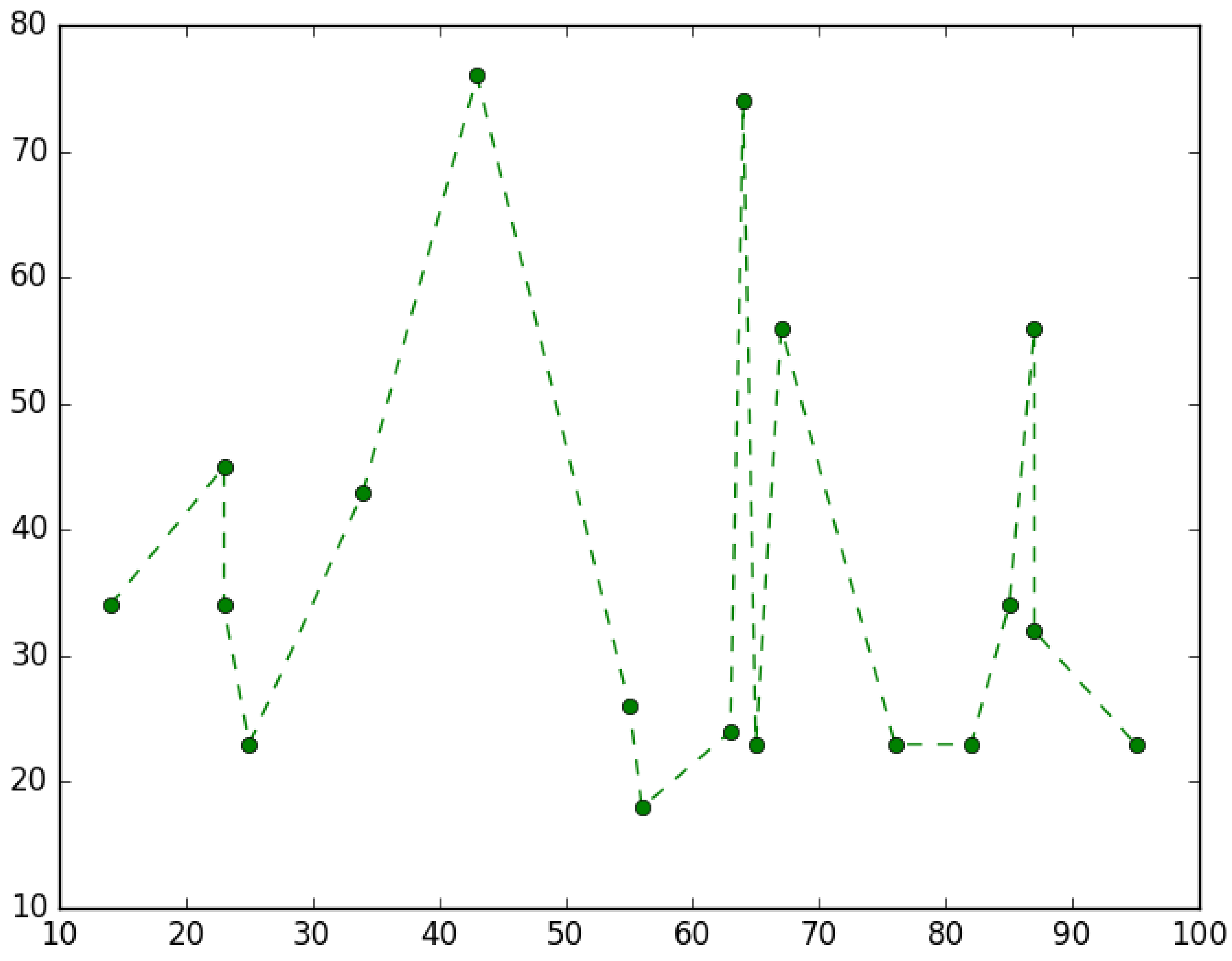
```
plt.plot(x, y, 'b^')
# Create blue up-facing triangles
```



Data and line

The style argument can take symbols for both markers and line style:

```
plt.plot(x, y, 'go--')
# green circles and dashed line
```



Read Basic Plots online: http://www.riptutorial.com/matplotlib/topic/3266/basic-plots

Chapter 4: Boxplots

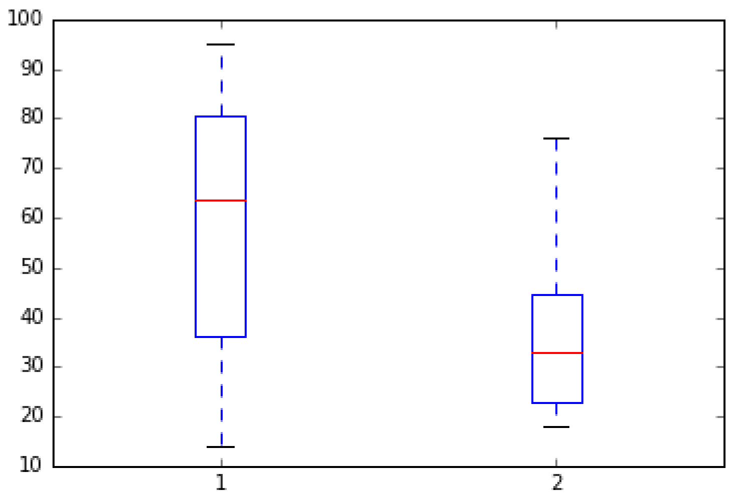
Examples

Basic Boxplots

Boxplots are descriptive diagrams that help to compare the distribution of different series of data. They are *descriptive* because they show measures (e.g. the *median*) which do not assume an underlying probability distribution.

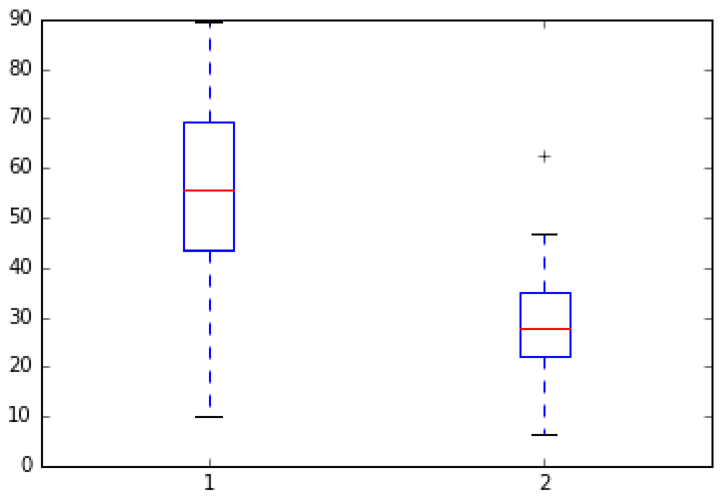
The most basic example of a boxplot in matplotlib can be achieved by just passing the data as a list of lists:

```
import matplotlib as plt

dataline1 = [43,76,34,63,56,82,87,55,64,87,95,23,14,65,67,25,23,85]
dataline2 = [34,45,34,23,43,76,26,18,24,74,23,56,23,23,34,56,32,23]
data = [ dataline1, dataline2 ]

plt.boxplot( data )
```



However, it is a common practice to use `numpy` arrays as parameters to the plots, since they are often the result of previous calculations. This can be done as follows:

```
import numpy as np
import matplotlib as plt

np.random.seed(123)
dataline1 = np.random.normal( loc=50, scale=20, size=18 )
dataline2 = np.random.normal( loc=30, scale=10, size=18 )
data = np.stack( [ dataline1, dataline2 ], axis=1 )

plt.boxplot( data )
```



Read Boxplots online: http://www.riptutorial.com/matplotlib/topic/6086/boxplots

Chapter 5: Boxplots

Examples

Boxplot function

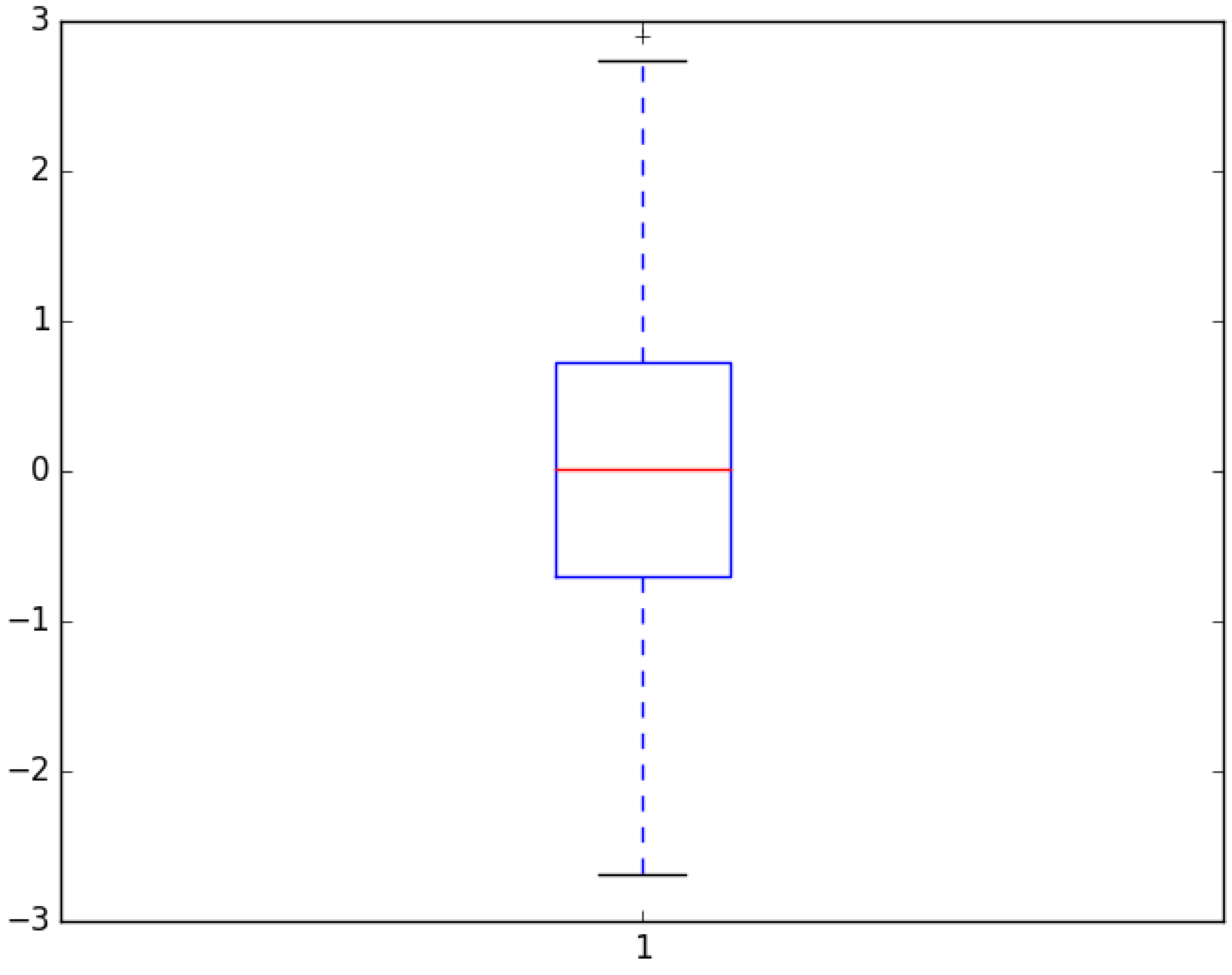
Matplotlib has its own implementation of boxplot. The relevant aspects of this function is that, by default, the boxplot is showing the median (percentile 50%) with a red line. The box represents Q1 and Q3 (percentiles 25 and 75), and the whiskers give an idea of the range of the data (possibly at Q1 - 1.5*IQR; Q3 + 1.5*IQR; being IQR the interquartile range, but this lacks confirmation). Also notice that samples beyond this range are shown as markers (these are named fliers).

> **NOTE:** Not all implementations of *boxplot* follow the same rules. Perhaps the most common boxplot diagram uses the whiskers to represent the minimum and maximum (making fliers non-existent). Also notice that this plot is sometimes called *box-and-whisker plot* and *box-and-whisker diagram*.

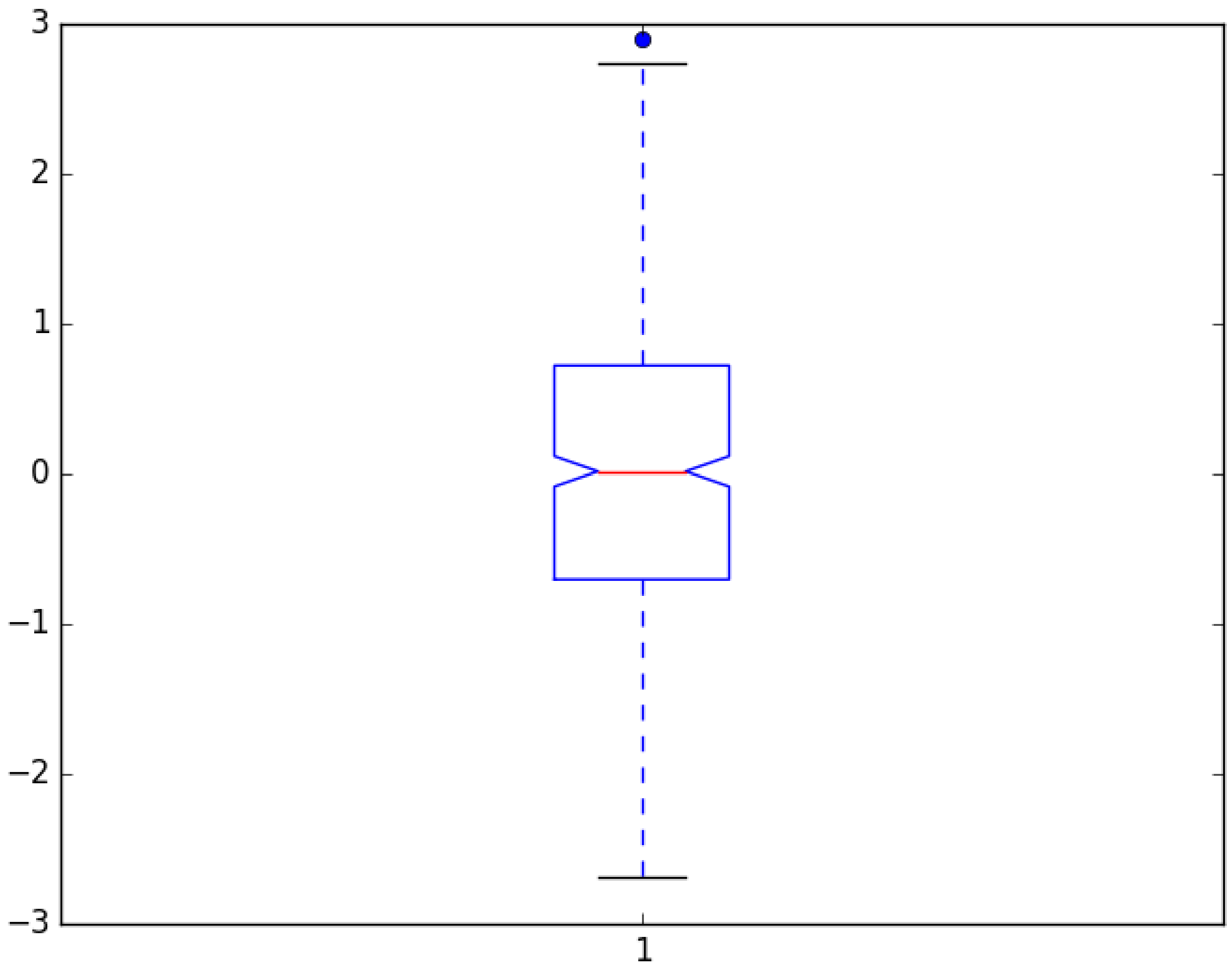
The following recipe show some of the things you can do with the current matplotlib implementation of boxplot:

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

X1 = np.random.normal(0, 1, 500)
X2 = np.random.normal(0.3, 1, 500)

# The most simple boxplot
plt.boxplot(X1)
plt.show()

# Changing some of its features
plt.boxplot(X1, notch=True, sym="o") # Use sym="" to shown no fliers; also showfliers=False
plt.show()

# Showing multiple boxplots on the same window
plt.boxplot((X1, X2), notch=True, sym="o", labels=["Set 1", "Set 2"])
plt.show()

# Hidding features of the boxplot
plt.boxplot(X2, notch=False, showfliers=False, showbox=False, showcaps=False, positions=[4],
labels=["Set 2"])
plt.show()

# Advanced customization of the boxplot
line_props = dict(color="r", alpha=0.3)
bbox_props = dict(color="g", alpha=0.9, linestyle="dashdot")
flier_props = dict(marker="o", markersize=17)
plt.boxplot(X1, notch=True, whiskerprops=line_props, boxprops=bbox_props,
flierprops=flier_props)
plt.show()
```

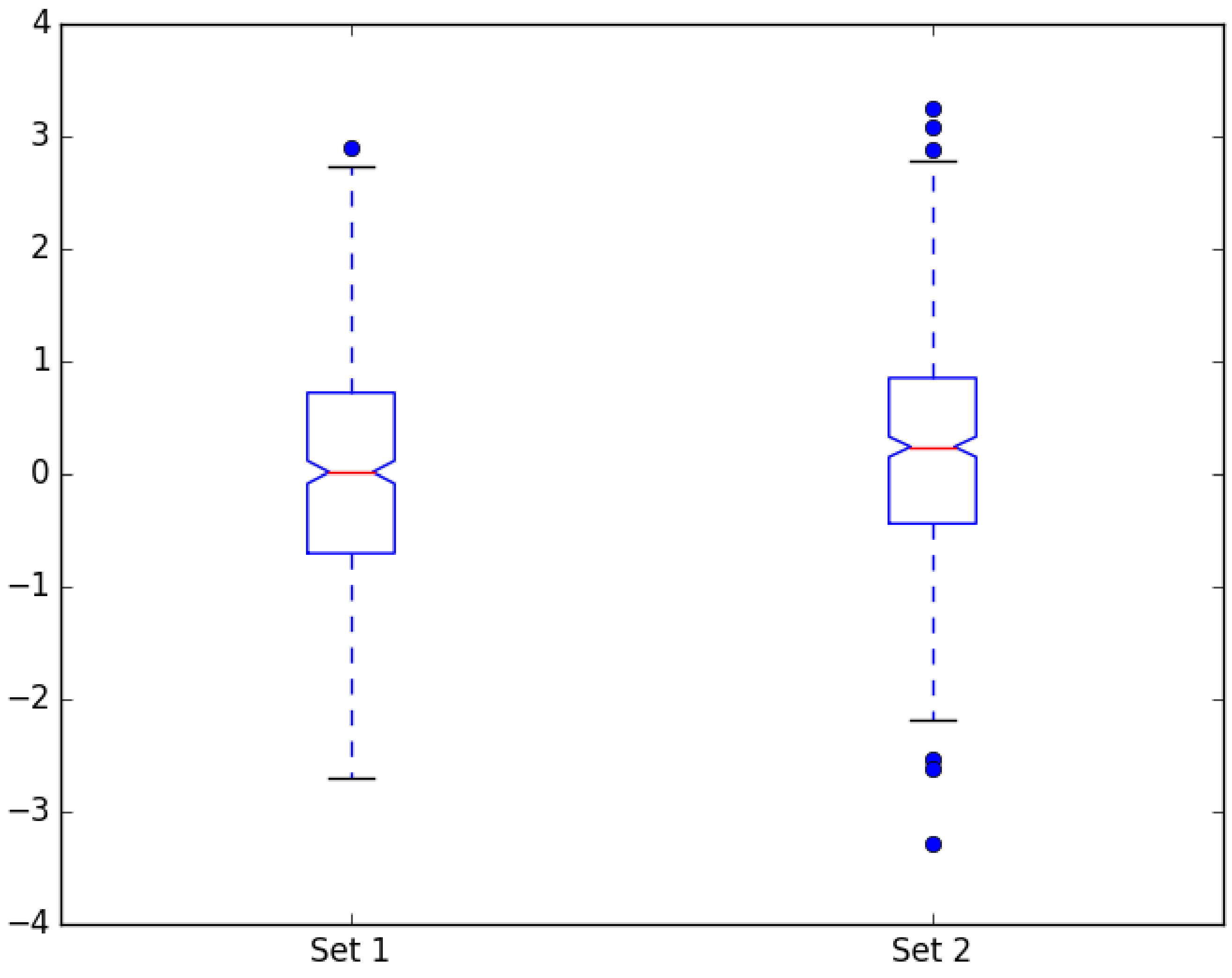
This result in the following plots:



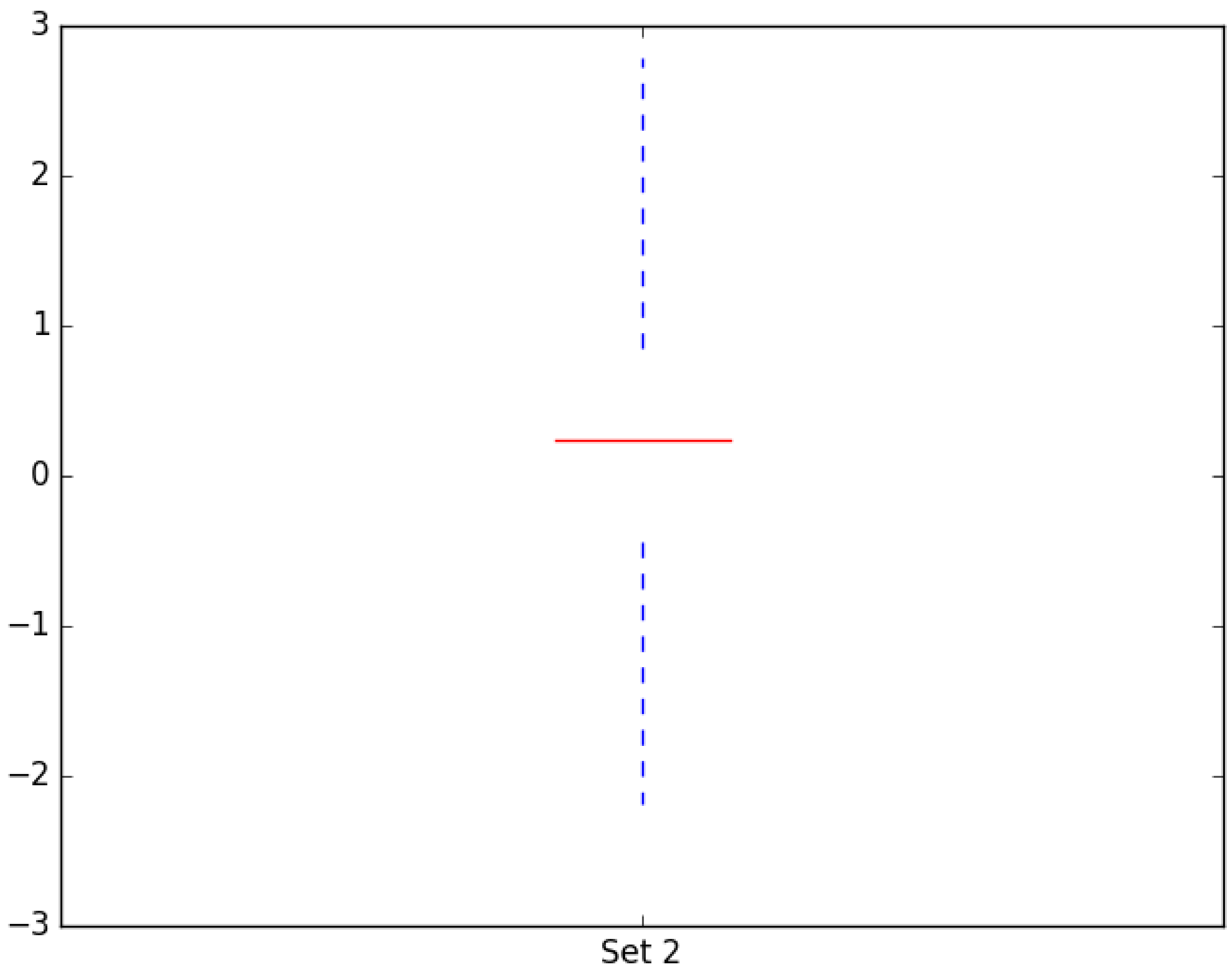
1. *Default matplotlib boxplot*



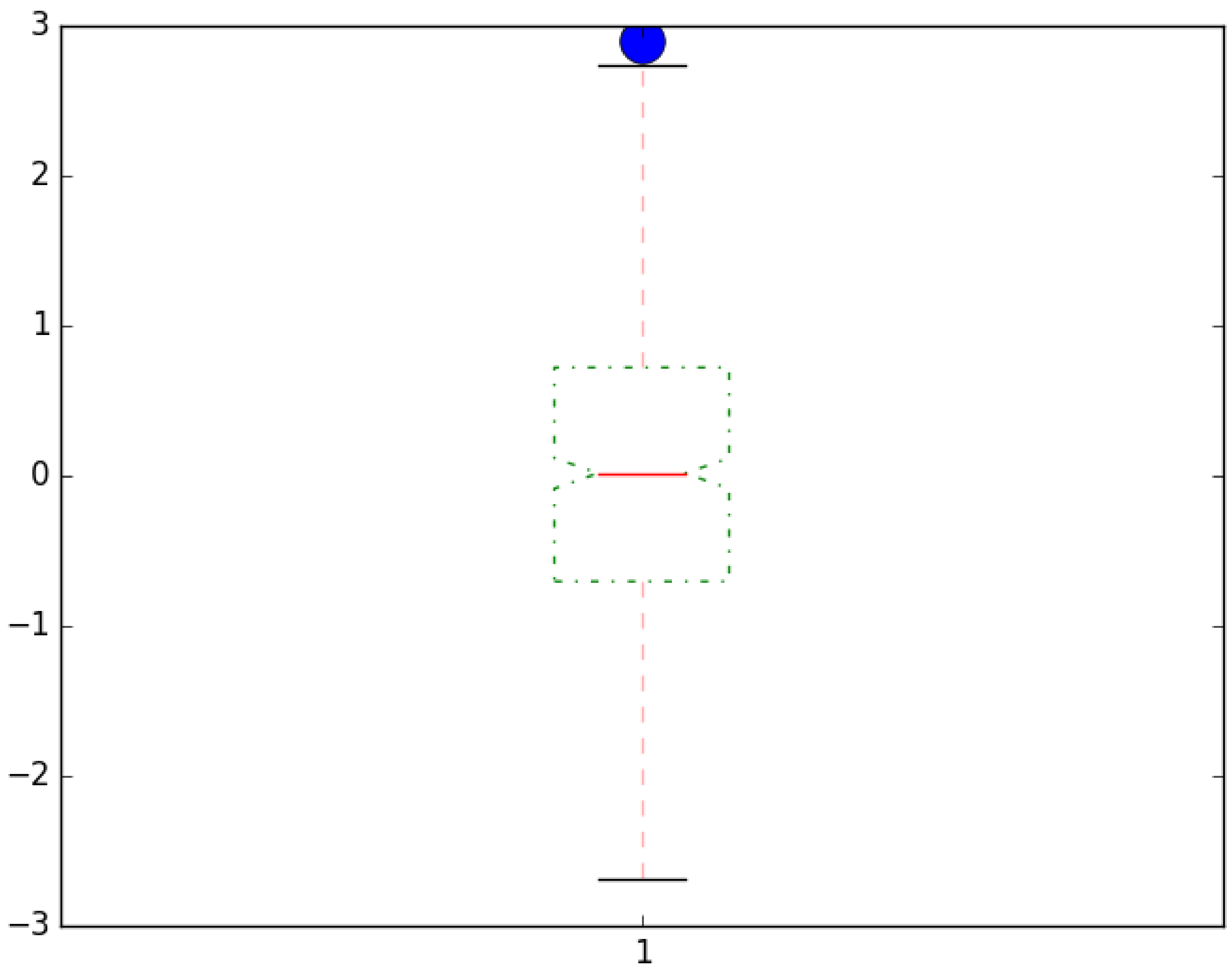
2. *Changing some features of the boxplot using function arguments*



3. *Multiple boxplot in the same plot window*



4. *Hidding some features of the boxplot*



5. *Advanced customization of a boxplot using props*

If you intend to do some advanced customization of your boxplot you should know that the props dictionaries you build (for example):

```
line_props = dict(color="r", alpha=0.3)
bbox_props = dict(color="g", alpha=0.9, linestyle="dashdot")
flier_props = dict(marker="o", markersize=17)
plt.boxplot(X1, notch=True, whiskerprops=line_props, boxprops=bbox_props,
flierprops=flier_props)
plt.show()
```

...refer mostly (if not all) to Line2D objects. This means that only arguments available in that class are changeable. You will notice the existence of keywords such as `whiskerprops`, `boxprops`, `flierprops`, and `capprops`. These are the elements you need to provide a props dictionary to further customize it.

NOTE: Further customization of the boxplot using this implementation might prove difficult. In some instances the use of other matplotlib elements such as patches to build ones own boxplot can be advantageous (considerable changes to the box element, for example).

Read Boxplots online: http://www.riptutorial.com/matplotlib/topic/6368/boxplots

Chapter 6: Closing a figure window

Syntax

- plt.close() # closes the current active figure
- plt.close(fig) # closes the figure with handle 'fig'
- plt.close(num) # closes the figure number 'num'
- plt.close(name) # closes the figure with the label 'name'
- plt.close('all') # closes all figures

Examples

Closing the current active figure using pyplot

The pyplot interface to `matplotlib` might be the simplest way to close a figure.

```
import matplotlib.pyplot as plt
plt.plot([0, 1], [0, 1])
plt.close()
```

Closing a specific figure using plt.close()

A specific figure can be closed by keeping its handle

```
import matplotlib.pyplot as plt

fig1 = plt.figure() # create first figure
plt.plot([0, 1], [0, 1])

fig2 = plt.figure() # create second figure
plt.plot([0, 1], [0, 1])

plt.close(fig1) # close first figure although second one is active
```

Read Closing a figure window online: http://www.riptutorial.com/matplotlib/topic/6628/closing-a-figure-window

Chapter 7: Colormaps

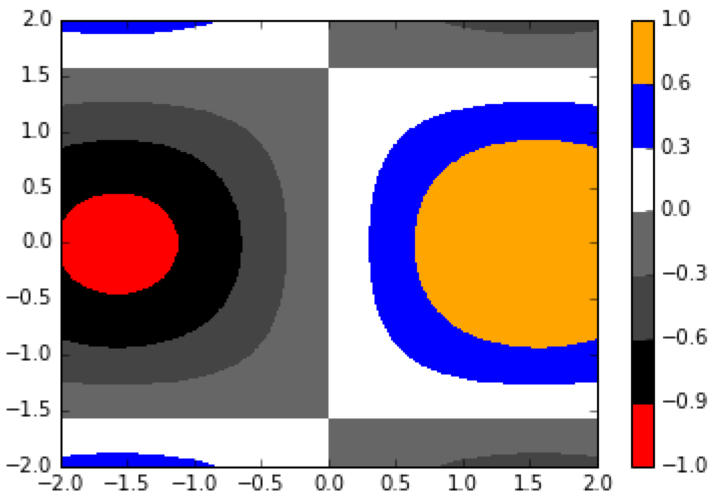
Examples

Custom discrete colormap

If you have predefined ranges and want to use specific colors for those ranges you can declare custom colormap. For example:

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

x = np.linspace(-2,2,500)
y = np.linspace(-2,2,500)
XX, YY = np.meshgrid(x, y)
Z = np.sin(XX) * np.cos(YY)

cmap = colors.ListedColormap(['red', '#000000','#444444', '#666666', '#ffffff', 'blue',
'orange'])
boundaries = [-1, -0.9, -0.6, -0.3, 0, 0.3, 0.6, 1]
norm = colors.BoundaryNorm(boundaries, cmap.N, clip=True)

plt.pcolormesh(x,y,Z, cmap=cmap, norm=norm)
plt.colorbar()
plt.show()
```

Produces

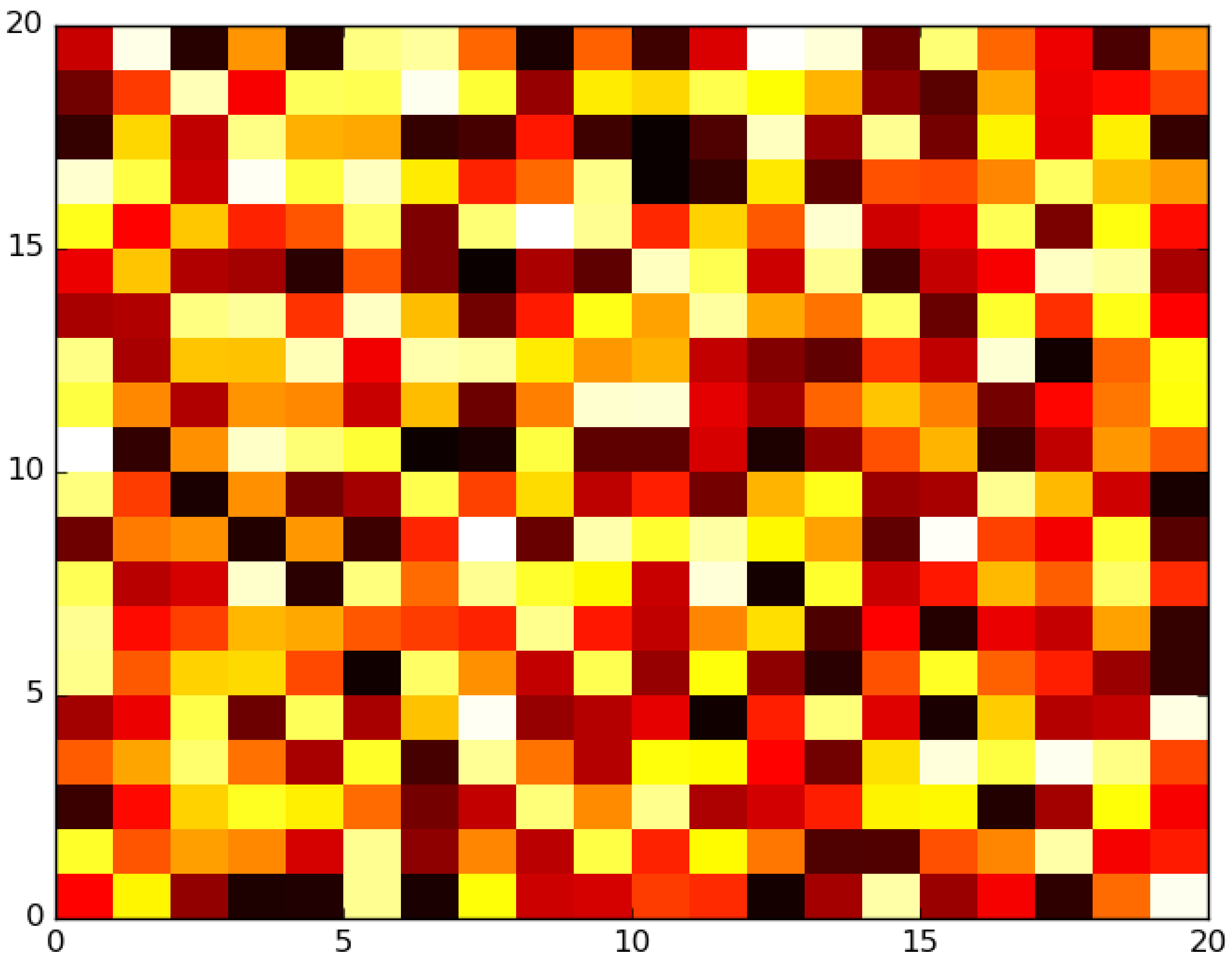


Color *i* will be used for values between boundary *i* and *i+1*. Colors can be specified by names (`'red'`, `'green'`), HTML codes (`'#ffaa44'`, `'#441188'`) or RGB tuples (`(0.2, 0.9, 0.45)`).

Basic usage

Using built-in colormaps is as simple as passing the name of the required colormap (as given in the colormaps reference) to the plotting function (such as `pcolormesh` or `contourf`) that expects it, usually in the form of a `cmap` keyword argument:

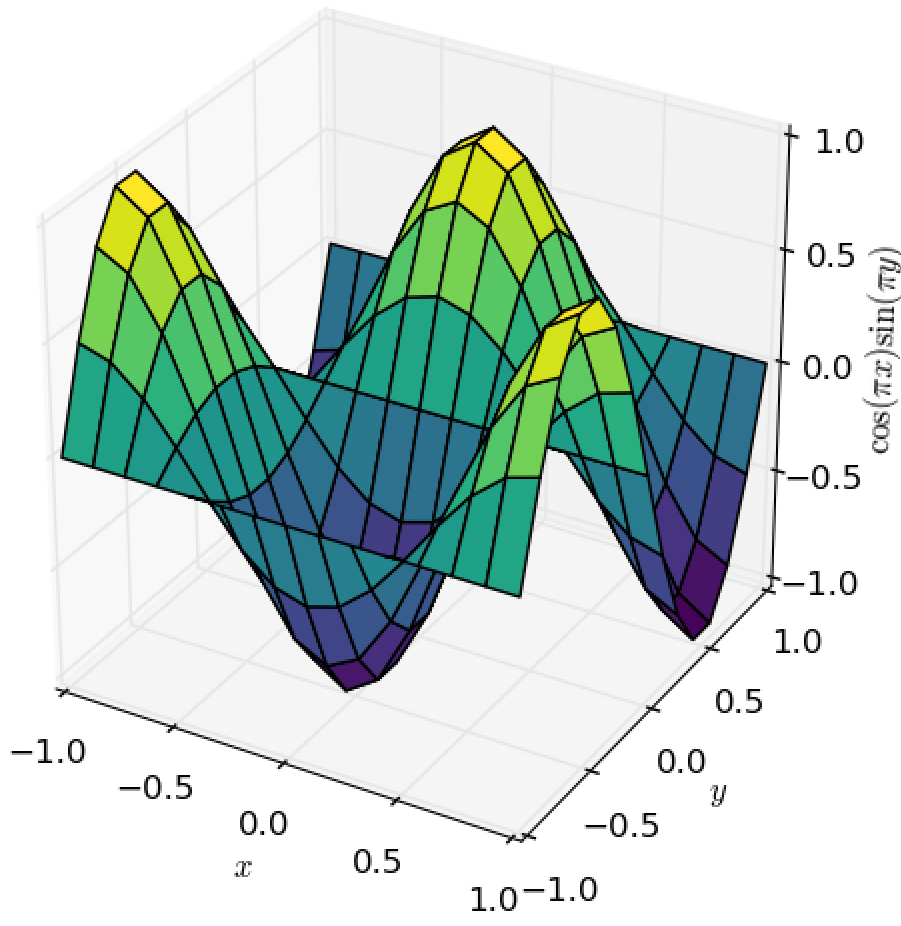
```
import matplotlib.pyplot as plt
import numpy as np

plt.figure()
plt.pcolormesh(np.random.rand(20,20),cmap='hot')
plt.show()
```



Colormaps are especially useful for visualizing three-dimensional data on two-dimensional plots, but a good colormap can also make a proper three-dimensional plot much clearer:

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.ticker import LinearLocator

# generate example data
import numpy as np
```

```
x,y = np.meshgrid(np.linspace(-1,1,15),np.linspace(-1,1,15))
z = np.cos(x*np.pi)*np.sin(y*np.pi)

# actual plotting example
fig = plt.figure()
ax1 = fig.add_subplot(121, projection='3d')
ax1.plot_surface(x,y,z,rstride=1,cstride=1,cmap='viridis')
ax2 = fig.add_subplot(122)
cf = ax2.contourf(x,y,z,51,vmin=-1,vmax=1,cmap='viridis')
cbar = fig.colorbar(cf)
cbar.locator = LinearLocator(numticks=11)
cbar.update_ticks()
for ax in {ax1, ax2}:
    ax.set_xlabel(r'$x$')
    ax.set_ylabel(r'$y$')
    ax.set_xlim([-1,1])
    ax.set_ylim([-1,1])
    ax.set_aspect('equal')

ax1.set_zlim([-1,1])
ax1.set_zlabel(r'$\cos(\pi x) \sin(\p    i y)$')

plt.show()
```

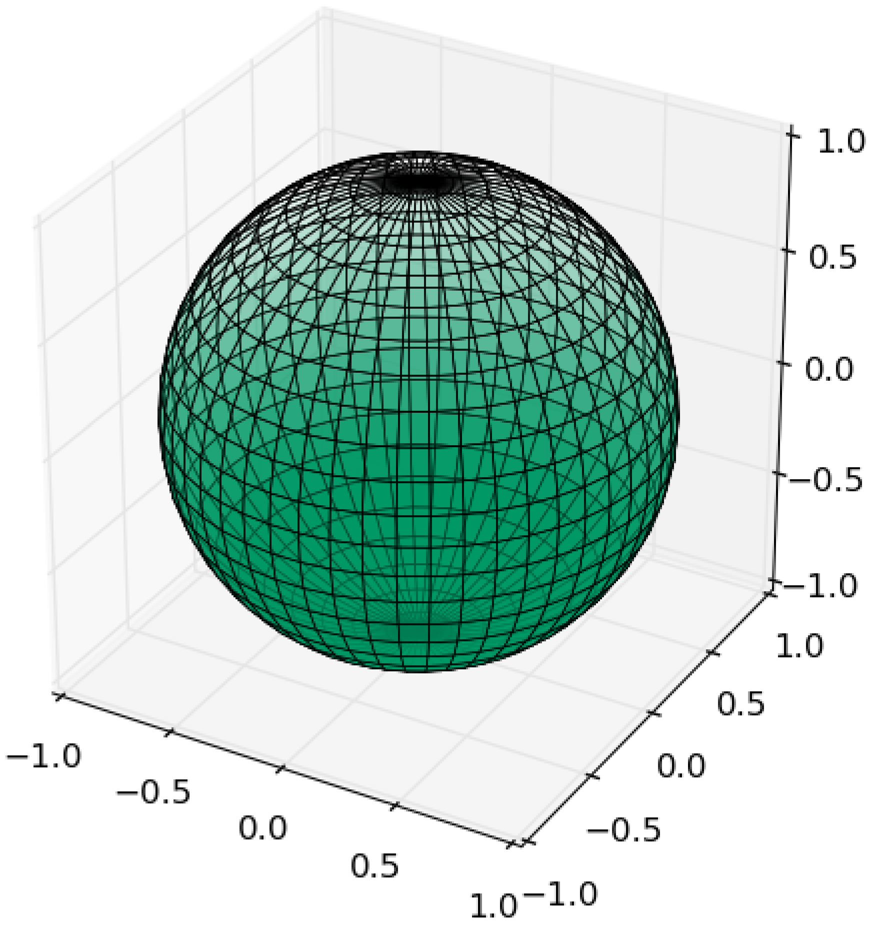


Using custom colormaps

Apart from the built-in colormaps defined in the colormaps reference (and their reversed maps, with `'_r'` appended to their name), custom colormaps can also be defined. The key is the `matplotlib.cm` module.

The below example defines a very simple colormap using `cm.register_cmap`, containing a single colour, with the opacity (alpha value) of the colour interpolating between fully opaque and fully transparent in the data range. Note that the important lines from the point of view of the colormap are the import of `cm`, the call to `register_cmap`, and the passing of the colormap to `plot_surface`.

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.cm as cm

# generate data for sphere
from numpy import pi,meshgrid,linspace,sin,cos
th,ph = meshgrid(linspace(0,pi,25),linspace(0,2*pi,51))
x,y,z = sin(th)*cos(ph),sin(th)*sin(ph),cos(th)

# define custom colormap with fixed colour and alpha gradient
# use simple linear interpolation in the entire scale
cm.register_cmap(name='alpha_gradient',
                 data={'red':   [(0.,0,0),
                                 (1.,0,0)],

                       'green': [(0.,0.6,0.6),
                                 (1.,0.6,0.6)],

                       'blue':  [(0.,0.4,0.4),
                                 (1.,0.4,0.4)],

                       'alpha': [(0.,1,1),
                                 (1.,0,0)]})

# plot sphere with custom colormap; constrain mapping to between |z|=0.7 for enhanced effect
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(x,y,z,cmap='alpha_gradient',vmin=-
0.7,vmax=0.7,rstride=1,cstride=1,linewidth=0.5,edgecolor='b')
ax.set_xlim([-1,1])
ax.set_ylim([-1,1])
ax.set_zlim([-1,1])
ax.set_aspect('equal')

plt.show()
```



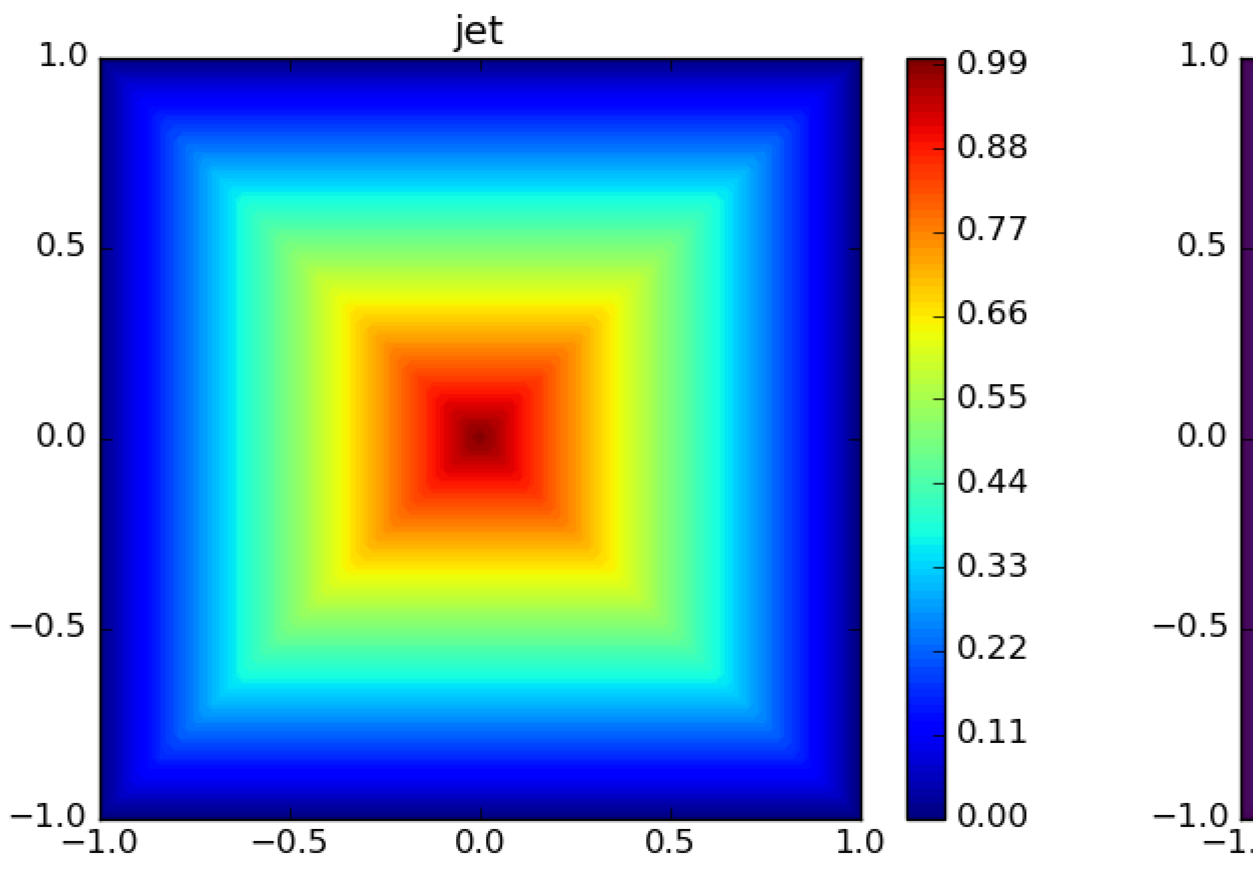
In more complicated scenarios, one can define a list of R/G/B(/A) values into which matplotlib interpolates linearly in order to determine the colours used in the corresponding plots.

Perceptually uniform colormaps

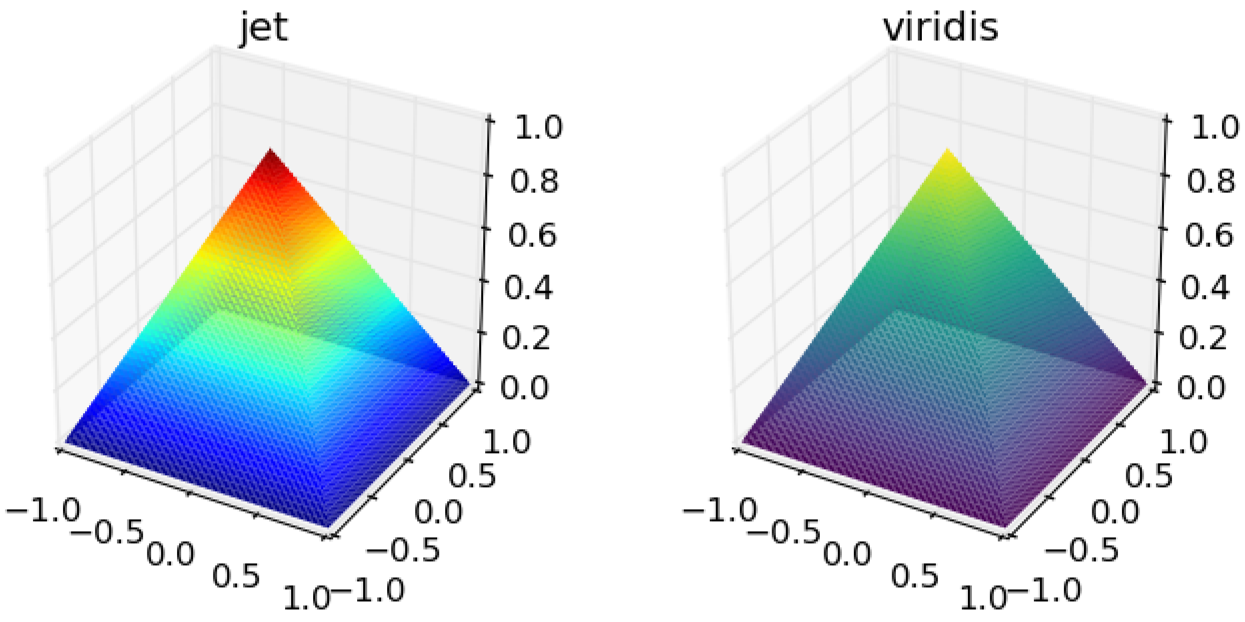
The original default colourmap of MATLAB (replaced in version R2014b) called `jet` is ubiquitous due to its high contrast and familiarity (and was the default of matplotlib for compatibility reasons). Despite its popularity, traditional colormaps often have deficiencies when it comes to representing data accurately. The percieved change in these colormaps does not correspond to changes in data; and a conversion of the colormap to greyscale (by, for instance, printing a figure using a black-and-white printer) might cause loss of information.

Perceptually uniform colormaps have been introduced to make data visualization as accurate and accessible as possible. Matplotlib introduced four new, perceptually uniform colormaps in version 1.5, with one of them (named `viridis`) to be the default from version 2.0. These four colormaps (`viridis`, `inferno`, `plasma` and `magma`) are all optimal from the point of view of perception, and these should be used for data visualization by default unless there are very good reasons not to do so. These colormaps introduce as little bias as possible (by not creating features where there aren't any to begin with), and they are suitable for an audience with reduced color perception.

As an example for visually distorting data, consider the following two top-view plots of pyramid-like objects:



Which one of the two is a proper pyramid? The answer is of course that both of them are, but this is far from obvious from the plot using the `jet` colormap:



This feature is at the core of perceptual uniformity.

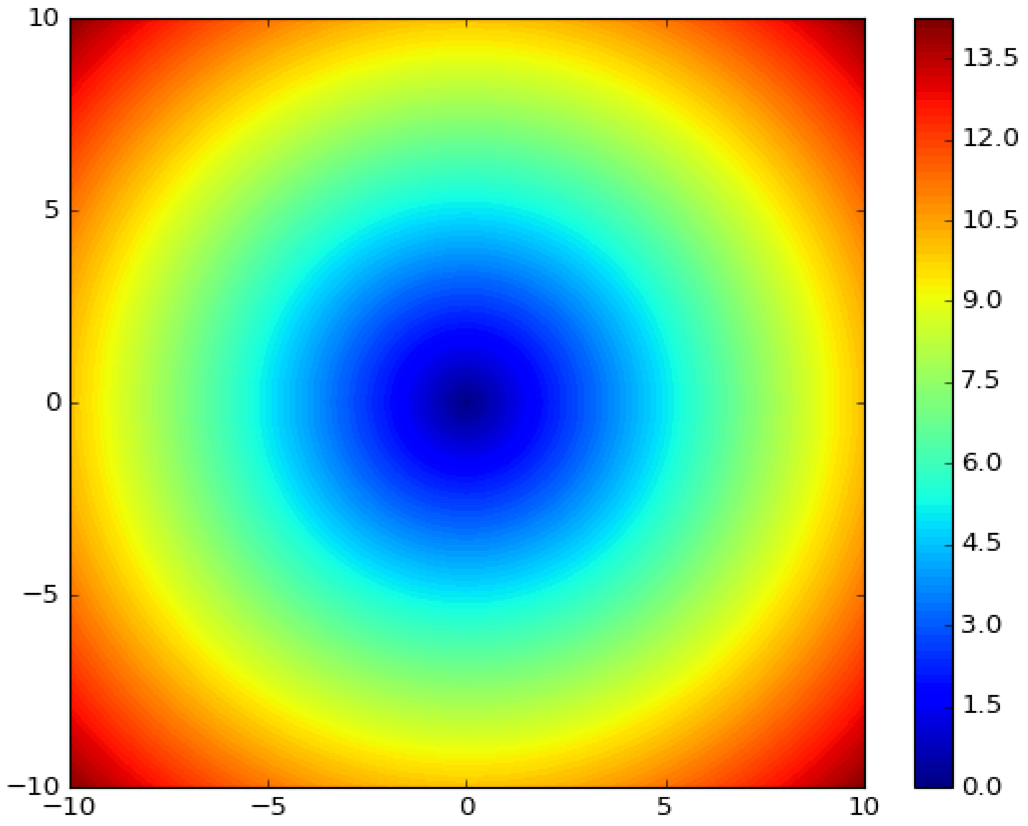
Read Colormaps online: http://www.riptutorial.com/matplotlib/topic/3385/colormaps

Chapter 8: Contour Maps

Examples

Simple filled contour plotting

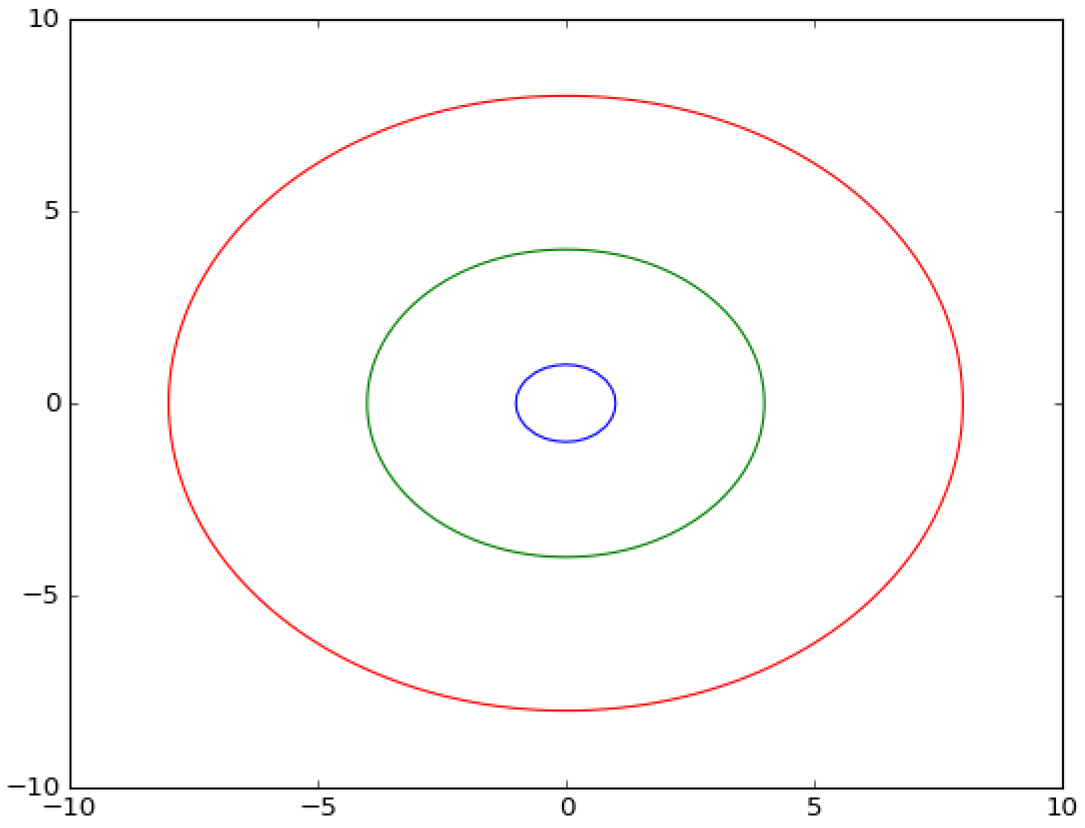
```
import matplotlib.pyplot as plt
import numpy as np

# generate 101 x and y values between -10 and 10
x = np.linspace(-10, 10, 101)
y = np.linspace(-10, 10, 101)

# make X and Y matrices representing x and y values of 2d plane
X, Y = np.meshgrid(x, y)

# compute z value of a point as a function of x and y (z = l2 distance form 0,0)
Z = np.sqrt(X ** 2 + Y ** 2)

# plot filled contour map with 100 levels
cs = plt.contourf(X, Y, Z, 100)

# add default colorbar for the map
plt.colorbar(cs)
```



Result:

Simple contour plotting

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

# generate 101 x and y values between -10 and 10
x = np.linspace(-10, 10, 101)
y = np.linspace(-10, 10, 101)

# make X and Y matrices representing x and y values of 2d plane
X, Y = np.meshgrid(x, y)

# compute z value of a point as a function of x and y (z = l2 distance form 0,0)
Z = np.sqrt(X ** 2 + Y ** 2)

# plot contour map with 3 levels
# colors: up to 1 - blue,  from 1 to 4 - green, from 4 to 8 - red
plt.contour(X, Y, Z, [1, 4, 8], colors=['b', 'g', 'r'])
```



Result:

Read Contour Maps online: http://www.riptutorial.com/matplotlib/topic/8644/contour-maps

Chapter 9: Coordinates Systems

Remarks

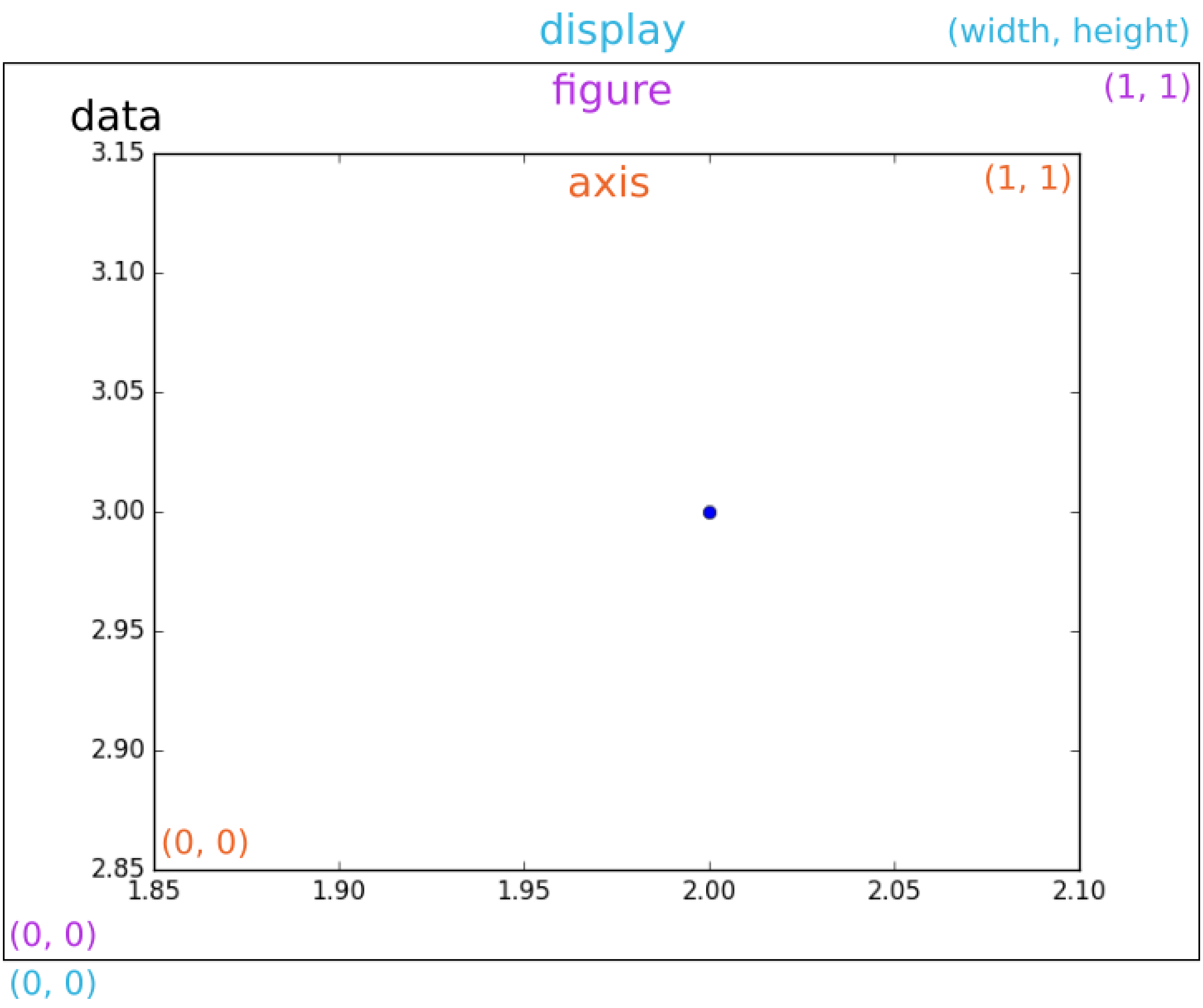
Matplotlib has four distinct coordinate systems which can be leveraged to ease the positioning of different object, e.g., text. Each system has a corresponding transformation object which transform coordinates from that system to the so called display coordinate system.

Data coordinate system is the system defined by the data on the respective axes. It is useful when trying to position some object relative to the data plotted. The range is given by the `xlim` and `ylim` properties of `Axes`. Its corresponding transformation object is `ax.transData`.

Axes coordinate system is the system tied to its `Axes` object. Points (0, 0) and (1, 1) define the bottom-left and top-right corners of the axes. As such it is useful when positioning relative to the axes, like top-center of the plot. Its corresponding transformation object is `ax.transAxes`.

Figure coordinate system is analogous to the axes coordinate system, except that it is tied to the `Figure`. Points (0, 0) and (1, 1) represent the bottom-left and top-right corners of the figure. It is useful when trying to position something relative to the whole image. Its corresponding transformation object is `fig.transFigure`.

Display coordinate system is the system of the image given in pixels. Points (0, 0) and (width, height) are the bottom-left and top-right pixels of image or display. It can be used for positioning absolutely. Since transformation objects transform coordinates into this coordinate system, display system has no transformation object associated with it. However, `None` or `matplotlib.transforms.IdentityTransform()` can be used when necessary.



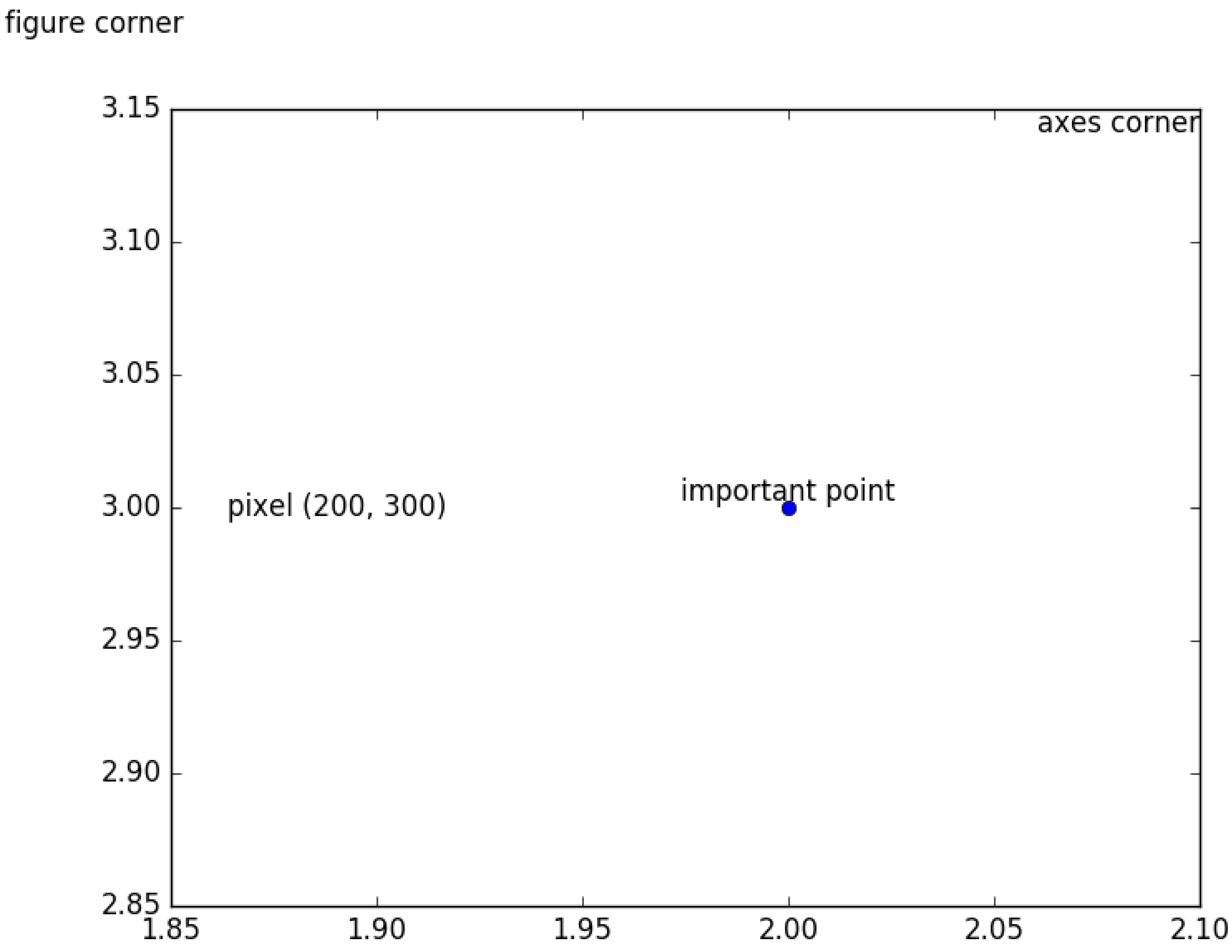
More details are available here.

Examples

Coordinate systems and text

The coordinate systems of Matplotlib come very handy when trying to annotate the plots you make. Sometimes you would like to position text relatively to your data, like when trying to label a specific point. Other times you would maybe like to add a text on top of the figure. This can easily be achieved by selecting an appropriate coordinate system by passing a transformation object to the `transform` parameter in call to `text()`.

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()

ax.plot([2.], [3.], 'bo')

plt.text(  # position text relative to data
    2., 3., 'important point',  # x, y, text,
    ha='center', va='bottom',   # text alignment,
    transform=ax.transData      # coordinate system transformation
)
plt.text(  # position text relative to Axes
    1.0, 1.0, 'axes corner',
    ha='right', va='top',
    transform=ax.transAxes
)
plt.text(  # position text relative to Figure
    0.0, 1.0, 'figure corner',
    ha='left', va='top',
    transform=fig.transFigure
)
plt.text(  # position text absolutely at specific pixel on image
    200, 300, 'pixel (200, 300)',
    ha='center', va='center',
    transform=None
)

plt.show()
```



Read Coordinates Systems online: http://www.riptutorial.com/matplotlib/topic/4566/coordinates-systems

Chapter 10: Figures and Axes Objects

Examples

Creating a figure

The figure contains all the plot elements. The main way to create a figure in `matplotlib` is to use `pyplot`.

```
import matplotlib.pyplot as plt
fig = plt.figure()
```

You can optionally supply a number, which you can use to access a previously-created figure. If a number is not supplied, the last-created figure's ID will be incremented and used instead; figures are indexed starting from 1, not 0.

```
import matplotlib.pyplot as plt
fig = plt.figure()
fig == plt.figure(1)  # True
```

Instead of a number, figures can also identified by a string. If using an interactive backend, this will also set the window title.

```
import matplotlib.pyplot as plt
fig = plt.figure('image')
```

To choose figure use

```
plt.figure(fig.number) # or
plt.figure(1)
```

Creating an axes

There are two main ways to create an axes in matplotlib: using pyplot, or using the object-oriented API.

Using pyplot:

```
import matplotlib.pyplot as plt

ax = plt.subplot(3, 2, 1)  # 3 rows, 2 columns, the first subplot
```

Using the object-oriented API:

```
import matplotlib.pyplot as plt

fig = plt.figure()
```

```
ax = fig.add_subplot(3, 2, 1)
```

The convenience function `plt.subplots()` can be used to produce a figure and collection of subplots in one command:

```
import matplotlib.pyplot as plt

fig, (ax1, ax2) = plt.subplots(ncols=2, nrows=1)  # 1 row, 2 columns
```

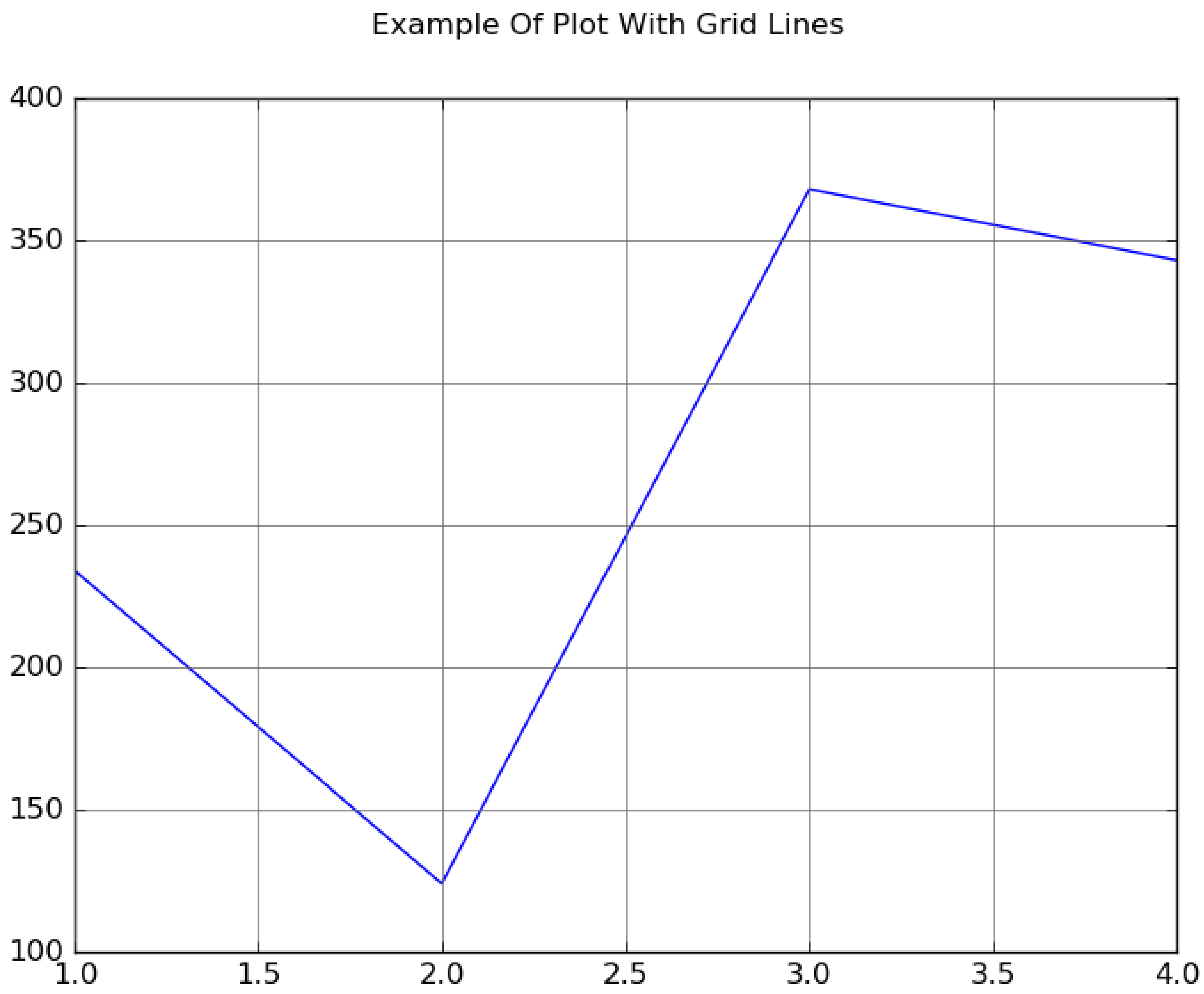
Read Figures and Axes Objects online: http://www.riptutorial.com/matplotlib/topic/2307/figures-and-axes-objects

Chapter 11: Grid Lines and Tick Marks

Examples

Plot With Gridlines

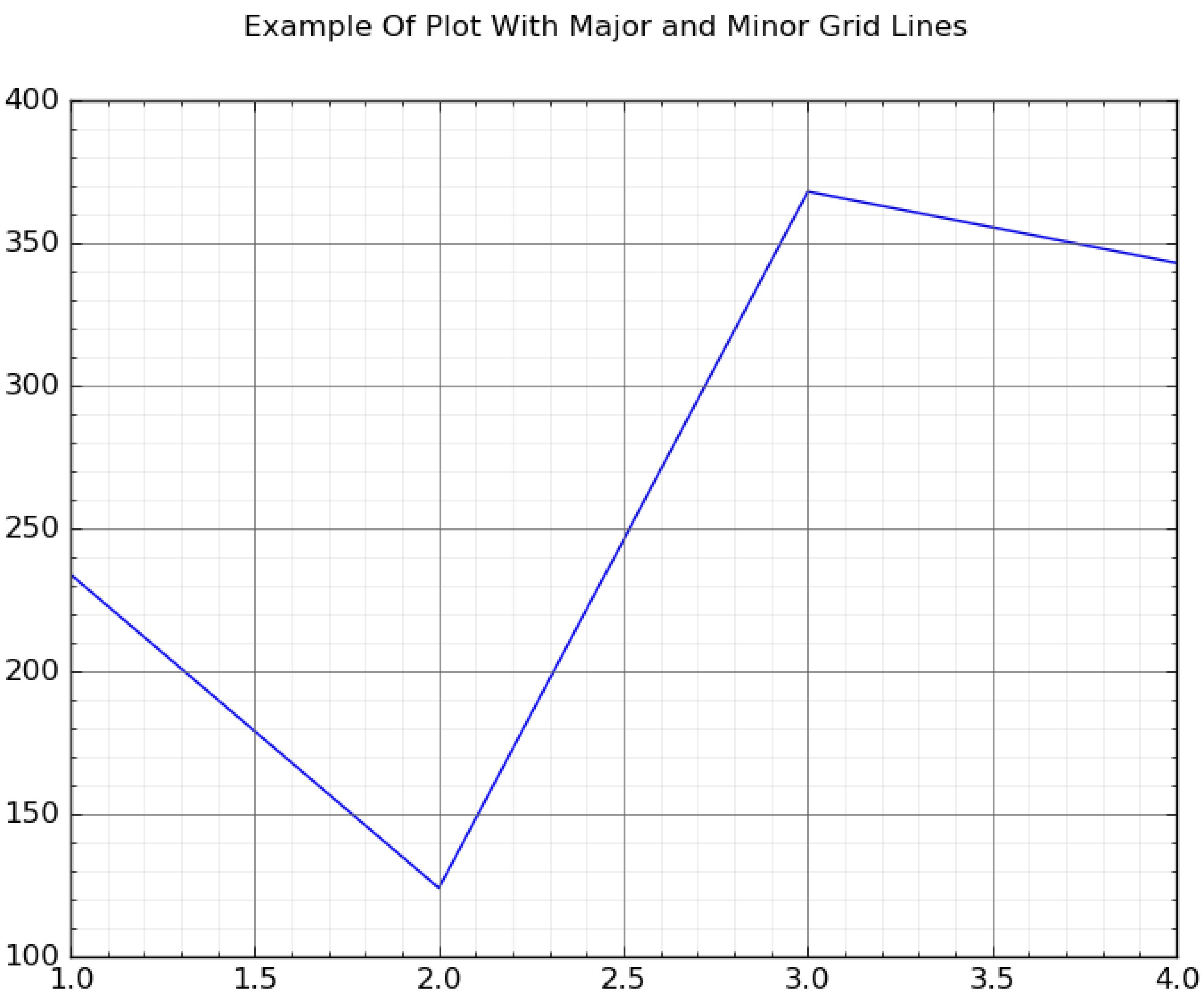
Plot With Grid Lines



```
import matplotlib.pyplot as plt

# The Data
x = [1, 2, 3, 4]
y = [234, 124,368, 343]

# Create the figure and axes objects
fig, ax = plt.subplots(1, figsize=(8, 6))
```

```
fig.suptitle('Example Of Plot With Grid Lines')

# Plot the data
ax.plot(x,y)

# Show the grid lines as dark grey lines
plt.grid(b=True, which='major', color='#666666', linestyle='-')

plt.show()
```

Plot With Major and Minor Grid Lines



```
import matplotlib.pyplot as plt

# The Data
x = [1, 2, 3, 4]
y = [234, 124,368, 343]

# Create the figure and axes objects
```

```
fig, ax = plt.subplots(1, figsize=(8, 6))
fig.suptitle('Example Of Plot With Major and Minor Grid Lines')

# Plot the data
ax.plot(x,y)

# Show the major grid lines with dark grey lines
plt.grid(b=True, which='major', color='#666666', linestyle='-')

# Show the minor grid lines with very faint and almost transparent grey lines
plt.minorticks_on()
plt.grid(b=True, which='minor', color='#999999', linestyle='-', alpha=0.2)

plt.show()
```

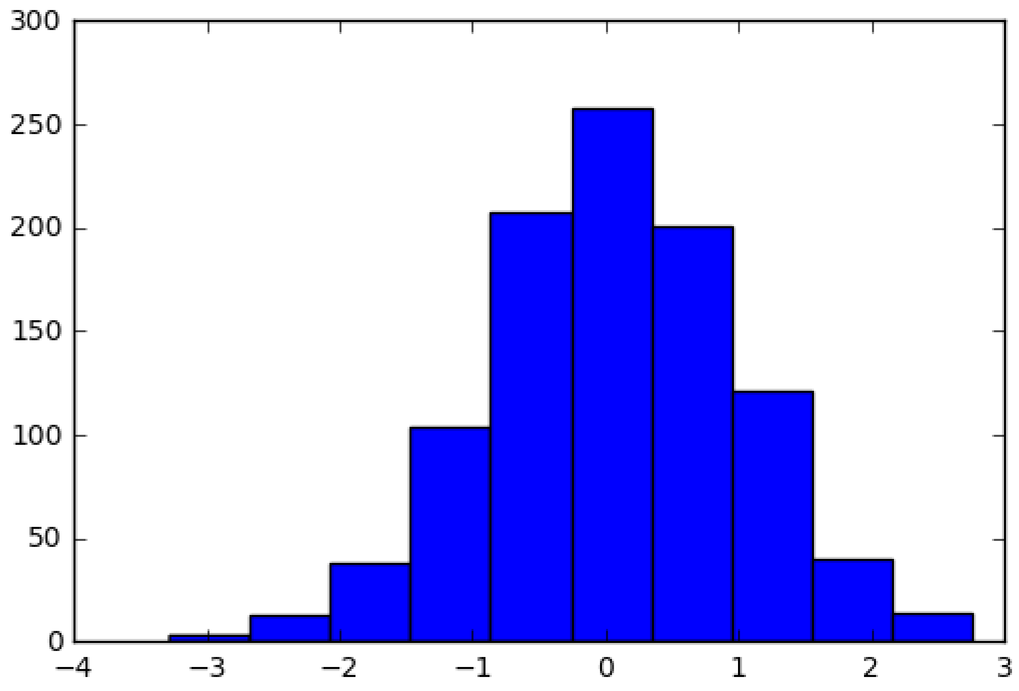
Read Grid Lines and Tick Marks online: http://www.riptutorial.com/matplotlib/topic/4029/grid-lines-and-tick-marks

Chapter 12: Histogram

Examples

Simple histogram

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

# generate 1000 data points with normal distribution
data = np.random.randn(1000)

plt.hist(data)

plt.show()
```



Read Histogram online: http://www.riptutorial.com/matplotlib/topic/7329/histogram

Chapter 13: Image manipulation

Examples

Opening images

Matplotlib includes the `image` module for image manipulation

```
import matplotlib.image as mpimg
import matplotlib.pyplot as plt
```

Images are read from file (`.png` only) with the `imread` function:

```
img = mpimg.imread('my_image.png')
```

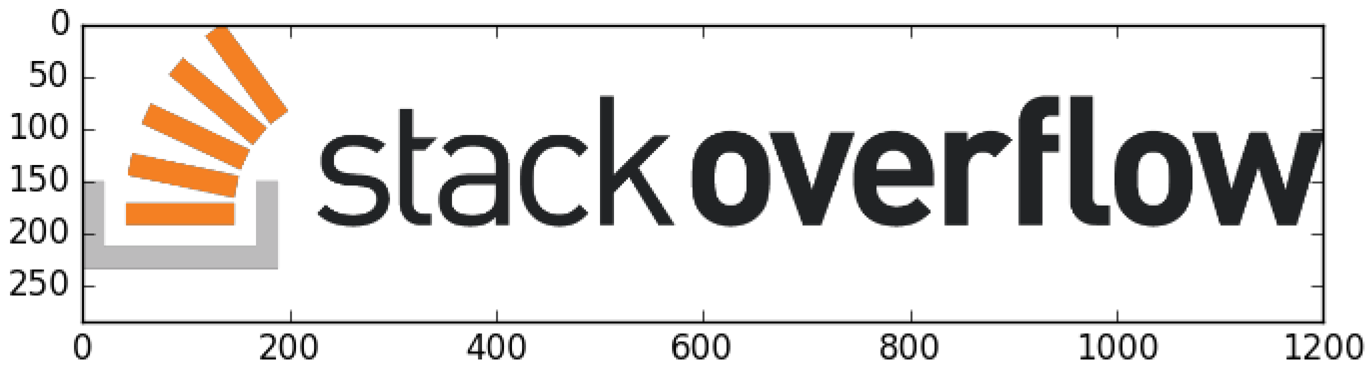
and they are rendered by the `imshow` function:

```
plt.imshow(img)
```

Let's *plot* the Stack Overflow logo:

```
import matplotlib.image as mpimg
import matplotlib.pyplot as plt
img = mpimg.imread('so-logo.png')
plt.imshow(img)
plt.show()
```

The resulting plot is



Read Image manipulation online: http://www.riptutorial.com/matplotlib/topic/4575/image-manipulation

Chapter 14: Integration with TeX/LaTeX

Remarks

- 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).
- Matplotlib’s pgf support requires a recent LaTeX installation that includes the TikZ/PGF packages (such as TeXLive), preferably with XeLaTeX or LuaLaTeX installed.

Examples

Inserting TeX formulae in plots

TeX formulae can be inserted in the plot using the `rc` function

```
import matplotlib.pyplot as plt
plt.rc(usetex = True)
```

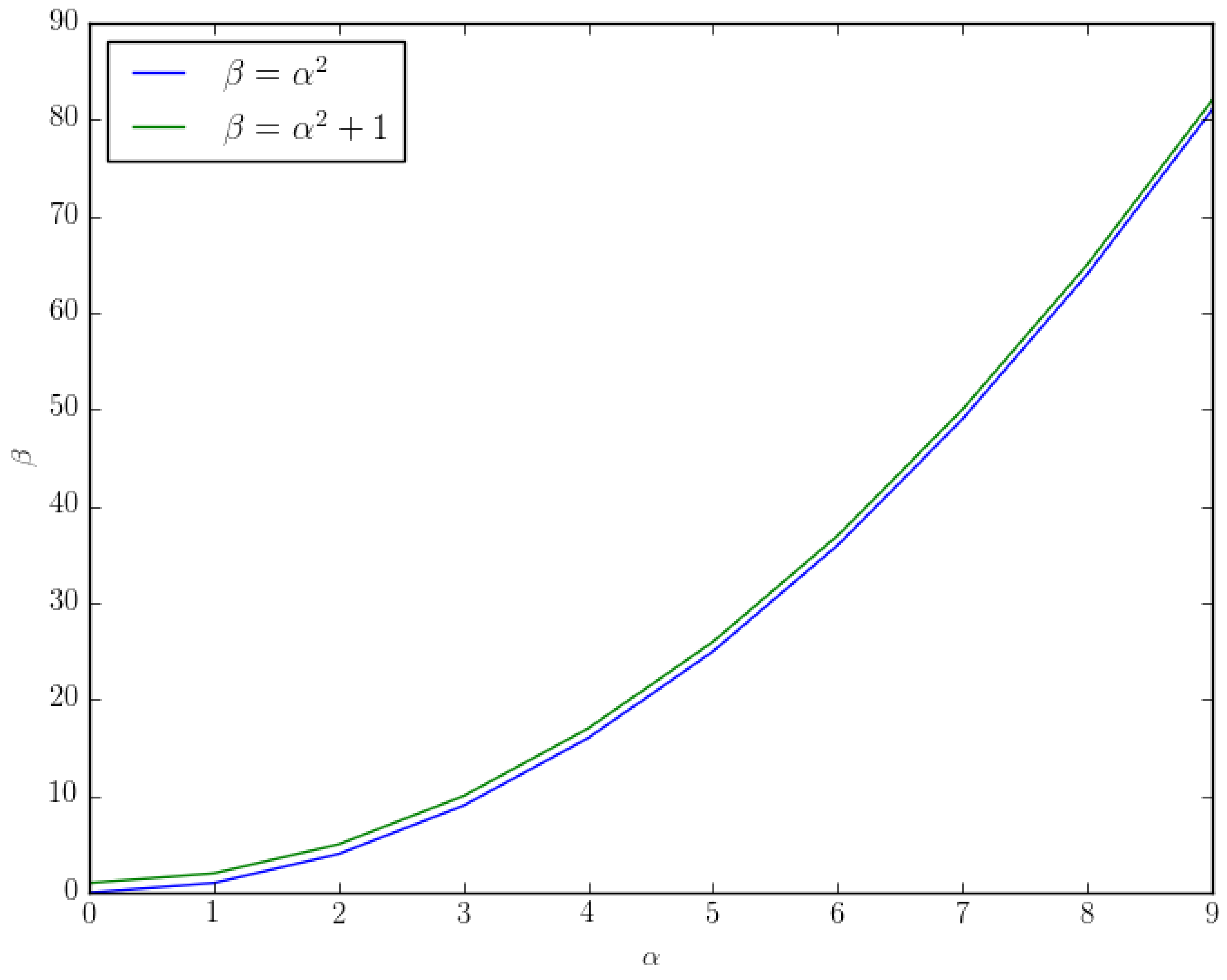
or accessing the `rcParams`:

```
import matplotlib.pyplot as plt
params = {'tex.usetex': True}
plt.rcParams.update(params)
```

TeX uses the backslash `\` for commands and symbols, which can conflict with special characters in Python strings. In order to use literal backslashes in a Python string, they must either be escaped or incorporated in a raw string:

```
plt.xlabel('\\alpha')
plt.xlabel(r'\alpha')
```

The following plot



can be produced by the code

```
import matplotlib.pyplot as plt
plt.rc(usetex = True)
x = range(0,10)
y = [t**2 for t in x]
z = [t**2+1 for t in x]
plt.plot(x, y, label = r'$\beta=\alpha^2$')
plt.plot(x, z, label = r'$\beta=\alpha^2+1$')
plt.xlabel(r'$\alpha$')
plt.ylabel(r'$\beta$')
plt.legend(loc=0)
plt.show()
```

Displayed equations (such as `$$...$$` or `\begin{equation}...\end{equation}`) are not supported. Nevertheless, displayed math style is possible with `\displaystyle`.

To load latex packages use the `tex.latex.preamble` argument:

```
params = {'text.latex.preamble' : [r'\usepackage{siunitx}', r'\usepackage{amsmath}']}
plt.rcParams.update(params)
```

Note, however, the warning in the example matplotlibrc file:

```
#text.latex.preamble : # IMPROPER USE OF THIS FEATURE WILL LEAD TO LATEX FAILURES
                       # AND IS THEREFORE UNSUPPORTED. PLEASE DO NOT ASK FOR HELP
                       # IF THIS FEATURE DOES NOT DO WHAT YOU EXPECT IT TO.
                       # preamble is a comma separated list of LaTeX statements
                       # that are included in the LaTeX document preamble.
                       # An example:
                       # text.latex.preamble : \usepackage{bm},\usepackage{euler}
                       # The following packages are always loaded with usetex, so
                       # beware of package collisions: color, geometry, graphicx,
                       # type1cm, textcomp. Adobe Postscript (PSSNFS) font packages
                       # may also be loaded, depending on your font settings
```

Saving and exporting plots that use TeX

In order to include plots created with matplotlib in TeX documents, they should be saved as `pdf` or `eps` files. In this way, any text in the plot (including TeX formulae) is rendered as text in the final document.

```
import matplotlib.pyplot as plt
plt.rc(usetex=True)
x = range(0, 10)
y = [t**2 for t in x]
z = [t**2+1 for t in x]
plt.plot(x, y, label=r'$\beta=\alpha^2$')
plt.plot(x, z, label=r'$\beta=\alpha^2+1$')
plt.xlabel(r'$\alpha$')
plt.ylabel(r'$\beta$')
plt.legend(loc=0)
plt.savefig('my_pdf_plot.pdf')  # Saving plot to pdf file
plt.savefig('my_eps_plot.eps')  # Saving plot to eps file
```

Plots in matplotlib can be exported to TeX code using the `pgf` macro package to display graphics.

```
import matplotlib.pyplot as plt
plt.rc(usetex=True)
x = range(0, 10)
y = [t**2 for t in x]
z = [t**2+1 for t in x]
plt.plot(x, y, label=r'$\beta=\alpha^2$')
plt.plot(x, z, label=r'$\beta=\alpha^2+1$')
plt.xlabel(r'$\alpha$')
plt.ylabel(r'$\beta$')
plt.legend(loc=0)
plt.savefig('my_pgf_plot.pgf')
```

Use the `rc`command to change the TeX engine used

```
plt.rc('pgf', texsystem='pdflatex')  # or luatex, xelatex...
```

To include the `.pgf` figure, write in your LaTeX document

```
\usepackage{pgf}
```

```
\input{my_pgf_plot.pgf}
```

Read Integration with TeX/LaTeX online: http://www.riptutorial.com/matplotlib/topic/2962/integration-with-tex-latex

Chapter 15: Legends

Examples

Simple Legend

Suppose you have multiple lines in the same plot, each of a different color, and you wish to make a legend to tell what each line represents. You can do this by passing on a label to each of the lines when you call `plot()`, e.g., the following line will be labelled *"My Line 1"*.

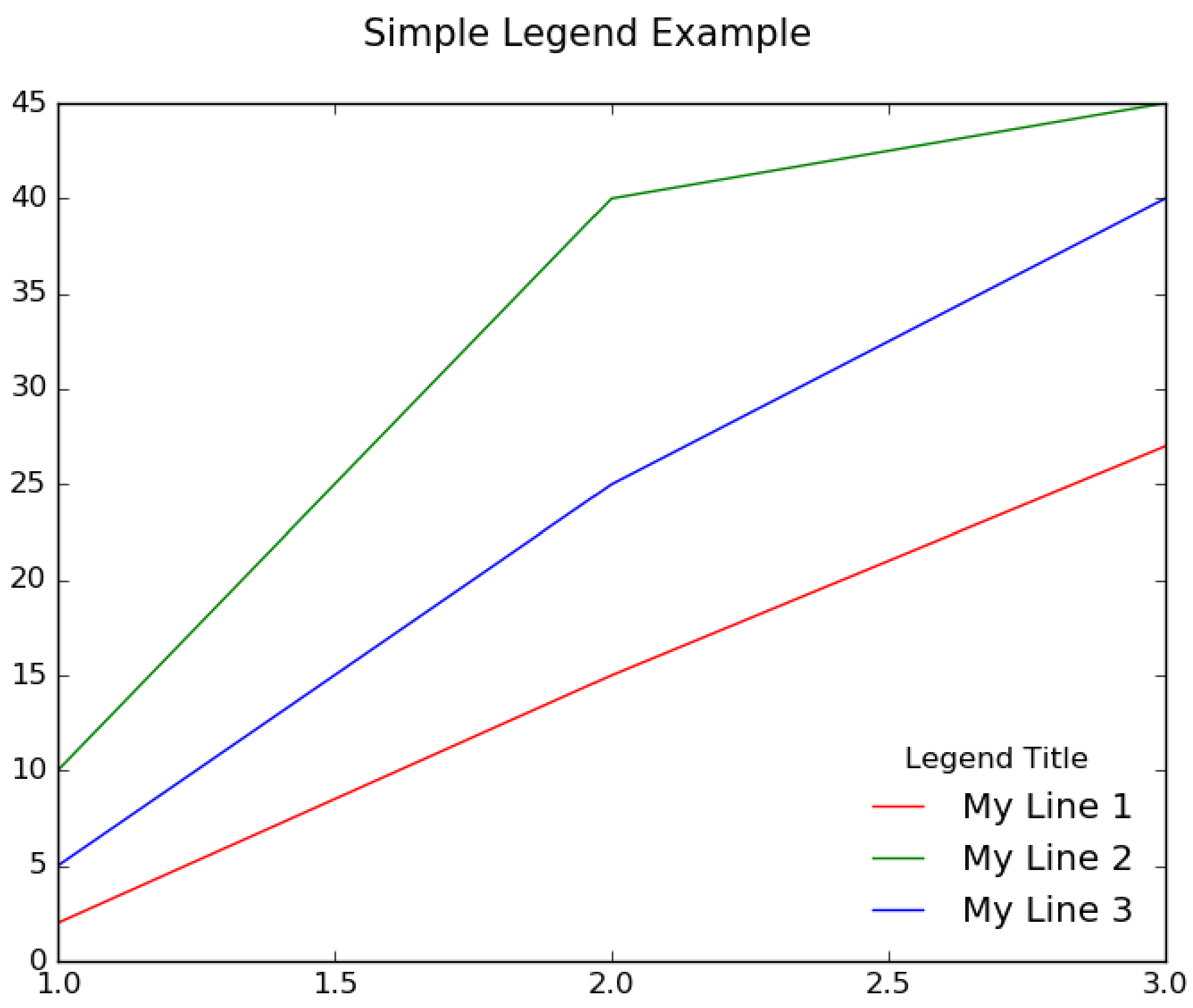
```
ax.plot(x, y1, color="red", label="My Line 1")
```

This specifies the text that will appear in the legend for that line. Now to make the actual legend visible, we can call `ax.legend()`

By default it will create a legend inside a box on the upper right hand corner of the plot. You can pass arguments to `legend()` to customize it. For instance we can position it on the lower right hand corner, with out a frame box surrounding it, and creating a title for the legend by calling the following:

```
ax.legend(loc="lower right", title="Legend Title", frameon=False)
```

Below is an example:

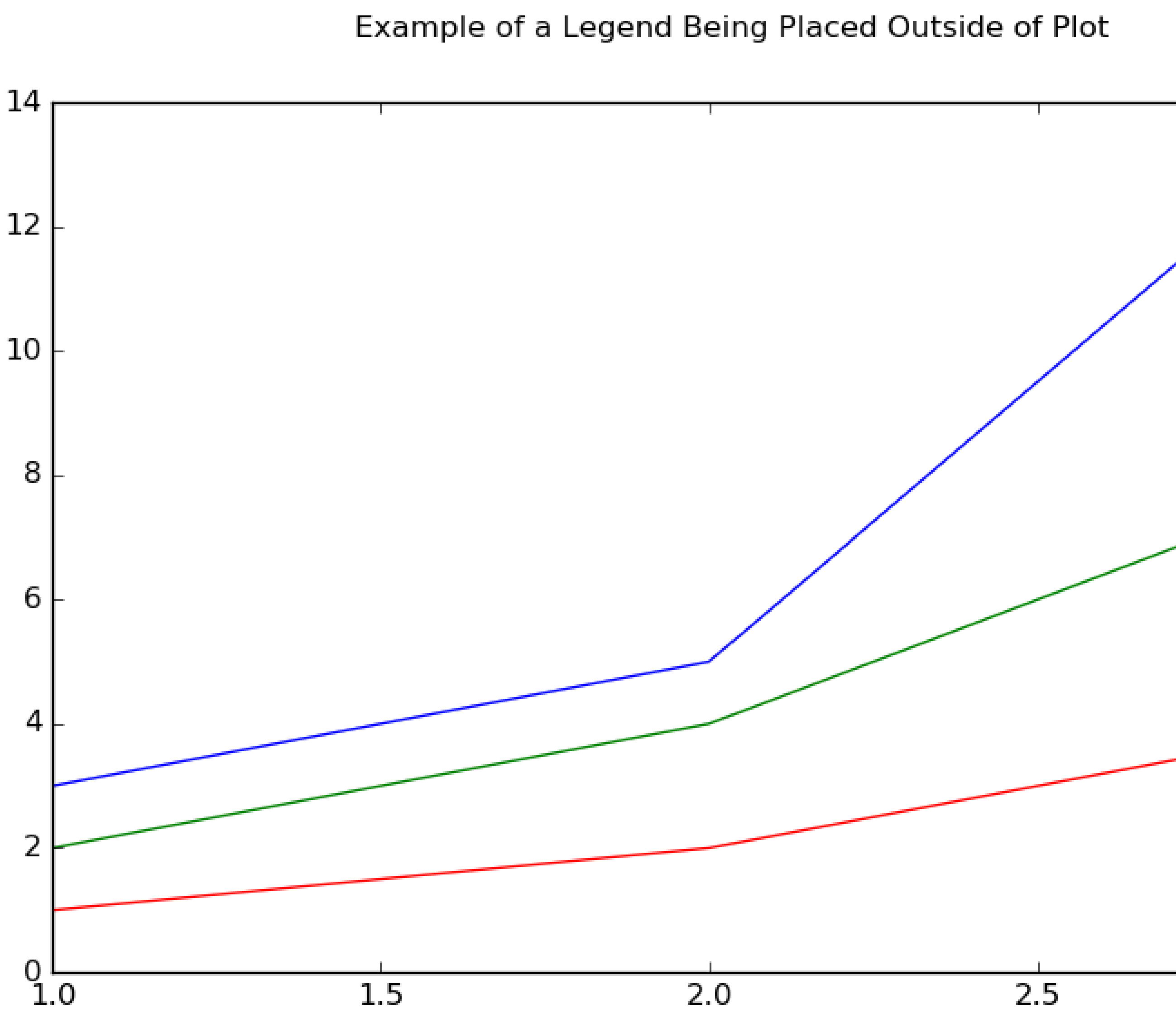


```
import matplotlib.pyplot as plt

# The data
x =  [1, 2, 3]
y1 = [2,  15, 27]
y2 = [10, 40, 45]
y3 = [5,  25, 40]

# Initialize the figure and axes
fig, ax = plt.subplots(1, figsize=(8, 6))

# Set the title for the figure
fig.suptitle('Simple Legend Example ', fontsize=15)

# Draw all the lines in the same plot, assigning a label for each one to be
# shown in the legend
ax.plot(x, y1, color="red", label="My Line 1")
ax.plot(x, y2, color="green", label="My Line 2")
ax.plot(x, y3, color="blue", label="My Line 3")

# Add a legend with title, position it on the lower right (loc) with no box framing (frameon)
ax.legend(loc="lower right", title="Legend Title", frameon=False)
```

```
# Show the plot
plt.show()
```

Legend Placed Outside of Plot

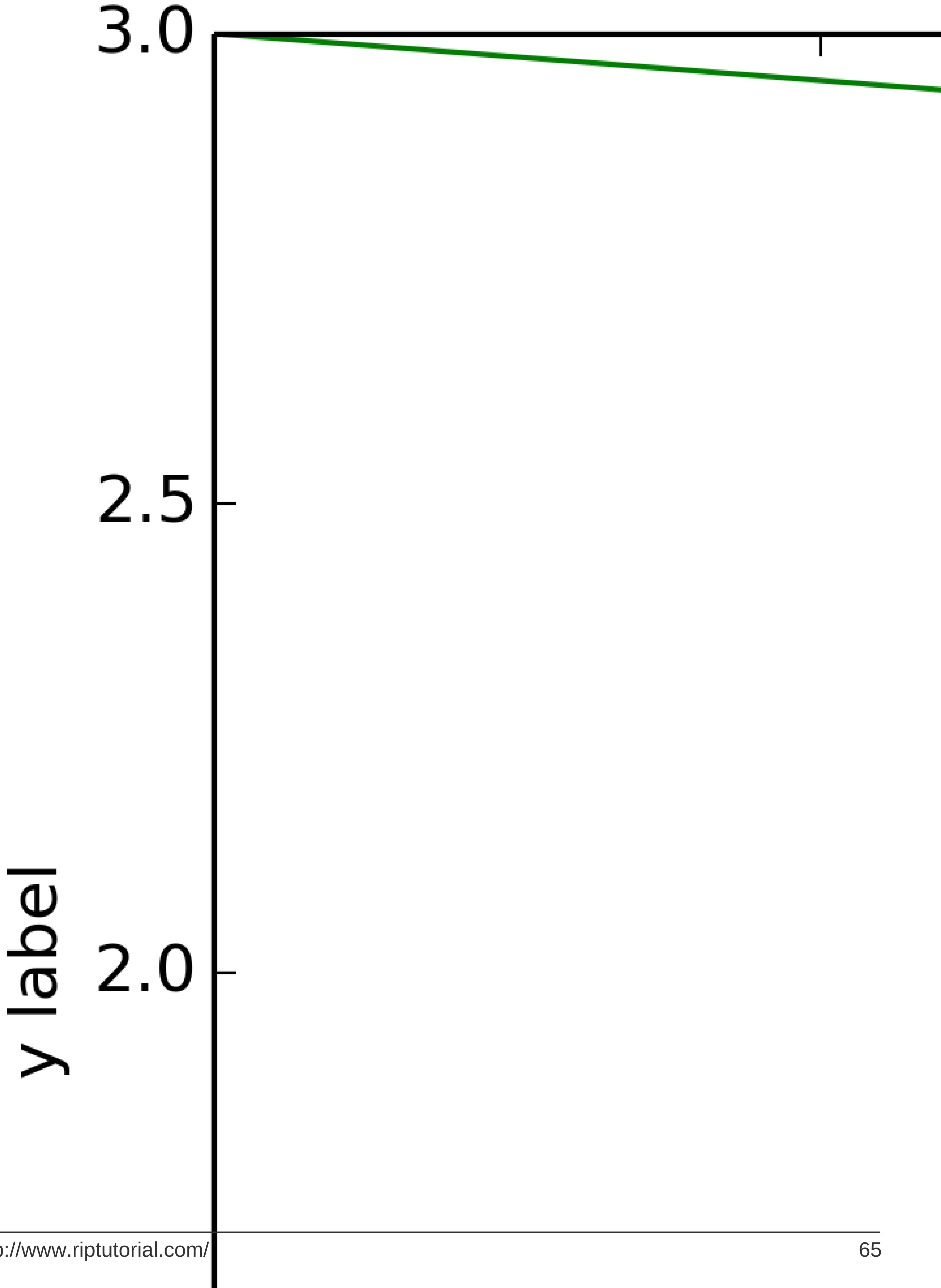
Sometimes it is necessary or desirable to place the legend outside the plot. The following code shows how to do it.



```
import matplotlib.pylab as plt
fig, ax = plt.subplots(1, 1, figsize=(10,6)) # make the figure with the size 10 x 6 inches
fig.suptitle('Example of a Legend Being Placed Outside of Plot')

# The data
x =  [1, 2, 3]
y1 = [1, 2, 4]
y2 = [2, 4, 8]
y3 = [3, 5, 14]

# Labels to use for each line
line_labels = ["Item A", "Item B", "Item C"]
```

```
# Create the lines, assigning different colors for each one.
# Also store the created line objects
l1 = ax.plot(x, y1, color="red")[0]
l2 = ax.plot(x, y2, color="green")[0]
l3 = ax.plot(x, y3, color="blue")[0]

fig.legend([l1, l2, l3],              # List of the line objects
           labels= line_labels,       # The labels for each line
           loc="center right",        # Position of the legend
           borderaxespad=0.1,         # Add little spacing around the legend box
           title="Legend Title")      # Title for the legend

# Adjust the scaling factor to fit your legend text completely outside the plot
# (smaller value results in more space being made for the legend)
plt.subplots_adjust(right=0.85)

plt.show()
```

Another way to place the legend outside the plot is to use `bbox_to_anchor` + `bbox_extra_artists` + `bbox_inches='tight'`, as shown in the example below:


3.0
2.5
2.0
y label

instead of creating a legend at the *axes* level (which will create a separate legend for each subplot). This is achieved by calling `fig.legend()` as can be seen in the code for the following code.

```
fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(10,4))
fig.suptitle('Example of a Single Legend Shared Across Multiple Subplots')

# The data
x =  [1, 2, 3]
y1 = [1, 2, 3]
y2 = [3, 1, 3]
y3 = [1, 3, 1]
y4 = [2, 2, 3]

# Labels to use in the legend for each line
line_labels = ["Line A", "Line B", "Line C", "Line D"]

# Create the sub-plots, assigning a different color for each line.
# Also store the line objects created
l1 = ax1.plot(x, y1, color="red")[0]
l2 = ax2.plot(x, y2, color="green")[0]
l3 = ax3.plot(x, y3, color="blue")[0]
l4 = ax3.plot(x, y4, color="orange")[0] # A second line in the third subplot

# Create the legend
fig.legend([l1, l2, l3, l4],     # The line objects
           labels=line_labels,   # The labels for each line
           loc="center right",   # Position of legend
           borderaxespad=0.1,    # Small spacing around legend box
           title="Legend Title"  # Title for the legend
           )

# Adjust the scaling factor to fit your legend text completely outside the plot
# (smaller value results in more space being made for the legend)
plt.subplots_adjust(right=0.85)

plt.show()
```

Something to note about the above example is the following:

```
l1 = ax1.plot(x, y1, color="red")[0]
```

When `plot()` is called, it returns a list of **line2D** objects. In this case it just returns a list with one single *line2D* object, which is extracted with the `[0]` indexing, and stored in `l1`.

A list of all the *line2D* objects that we are interested in including in the legend need to be passed on as the first argument to `fig.legend()`. The second argument to `fig.legend()` is also necessary. It is supposed to be a list of strings to use as the labels for each line in the legend.

The other arguments passed on to `fig.legend()` are purely optional, and just help with fine-tuning the aesthetics of the legend.

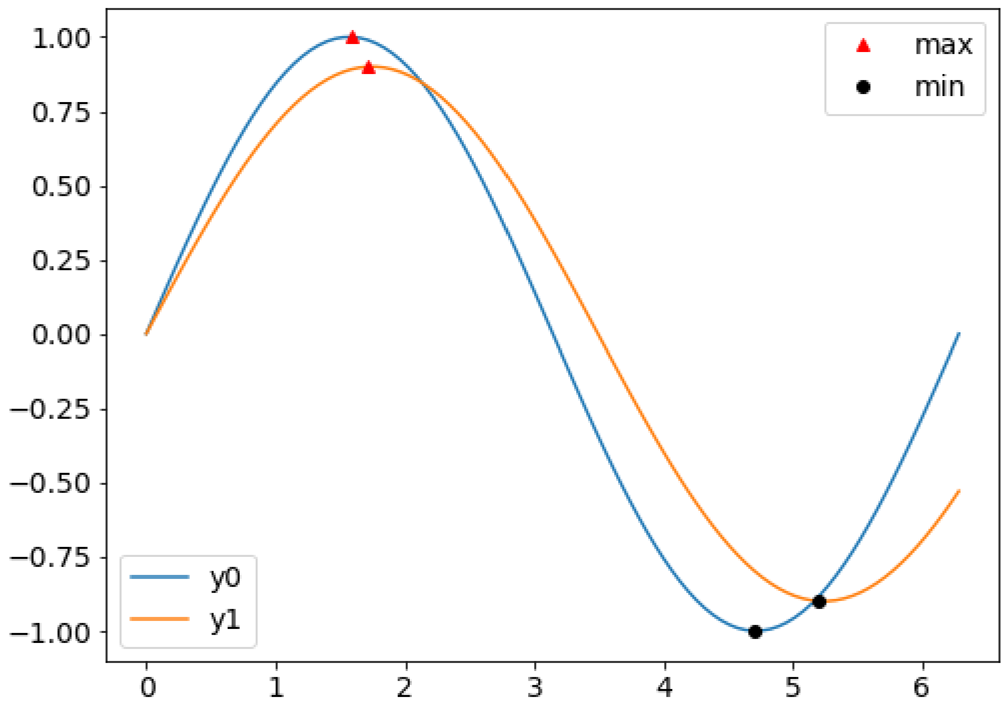
Multiple Legends on the Same Axes

If you call `plt.legend()` or `ax.legend()` more than once, the first legend is removed and a new one is drawn. According the official documentation:

> This has been done so that it is possible to call legend() repeatedly to update the legend to the latest handles on the Axes

Fear not, though: It is still quite simple to add a second legend (or third, or fourth...) to an axes. In the example here, we plot two lines, then plot markers on their respective maxima and minima. One legend is for the lines, and the other is for the markers.

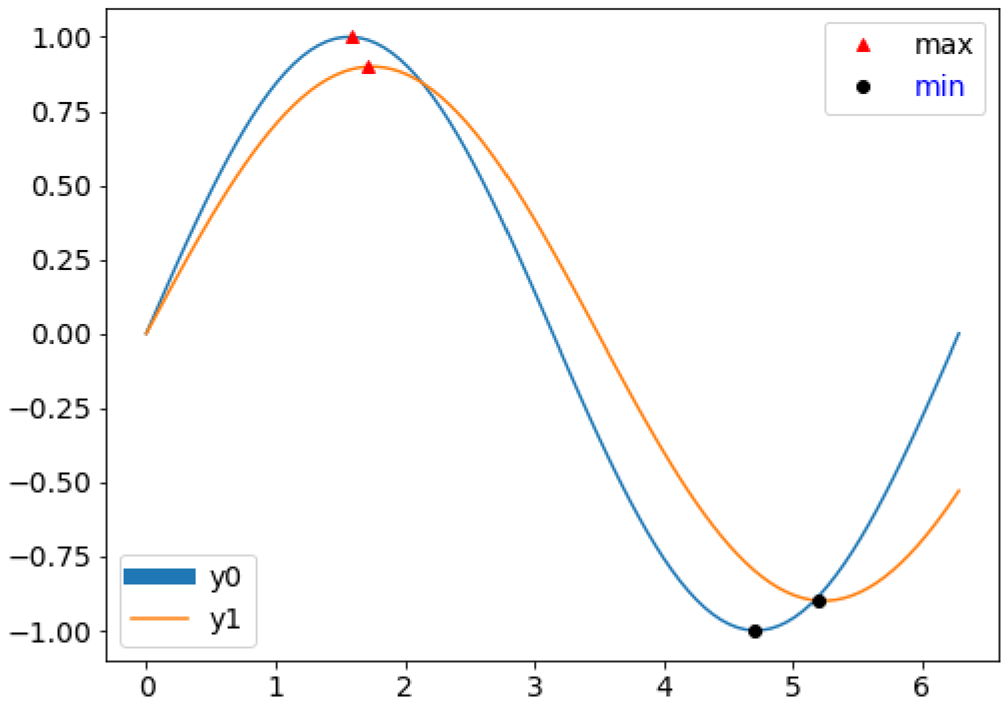
```
import matplotlib.pyplot as plt
import numpy as np

# Generate data for plotting:
x = np.linspace(0,2*np.pi,100)
y0 = np.sin(x)
y1 = .9*np.sin(.9*x)
# Find their maxima and minima and store
maxes = np.empty((2,2))
mins = np.empty((2,2))
for k,y in enumerate([y0,y1]):
    maxloc = y.argmax()
    maxes[k] = x[maxloc], y[maxloc]
    minloc = y.argmin()
    mins[k] = x[minloc], y[minloc]

# Instantiate figure and plot
fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(x,y0, label='y0')
ax.plot(x,y1, label='y1')
# Plot maxima and minima, and keep references to the lines
maxline, = ax.plot(maxes[:,0], maxes[:,1], 'r^')
minline, = ax.plot(mins[:,0], mins[:,1], 'ko')

# Add first legend:  only labeled data is included
leg1 = ax.legend(loc='lower left')
# Add second legend for the maxes and mins.
# leg1 will be removed from figure
leg2 = ax.legend([maxline,minline],['max','min'], loc='upper right')
# Manually add the first legend back
ax.add_artist(leg1)
```



The key is to make sure you have references to the legend objects. The first one you instantiate (`leg1`) is removed from the figure when you add the second one, but the `leg1` object still exists and can be added back with `ax.add_artist`.

The really great thing is that you can can still manipulate *both* legends. For example, add the following to the bottom of the above code:

```
leg1.get_lines()[0].set_lw(8)
leg2.get_texts()[1].set_color('b')
```



Finally, it's worth mentioning that in the example only the lines were given labels when plotted, meaning that `ax.legend()` adds only those lines to the `leg1`. The legend for the markers (`leg2`) therefore required the lines and labels as arguments when it was instantiated. We could have, alternatively, given labels to the markers when they were plotted too. But then *both* calls to `ax.legend` would have required some extra arguments so that each legend contained only the items we wanted.

Read Legends online: http://www.riptutorial.com/matplotlib/topic/2840/legends

Chapter 16: LogLog Graphing

Introduction

LogLog graphing is a possibility to illustrate an exponential function in a linear way.

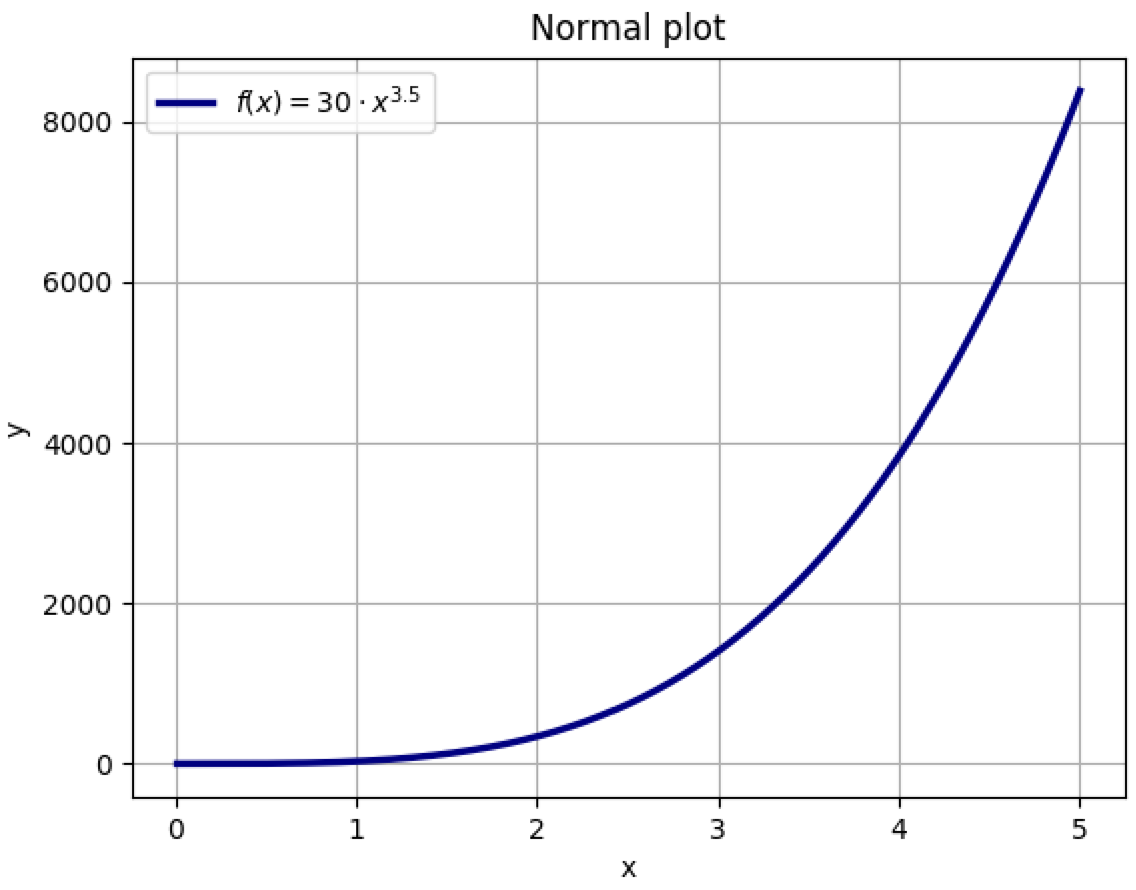
Examples

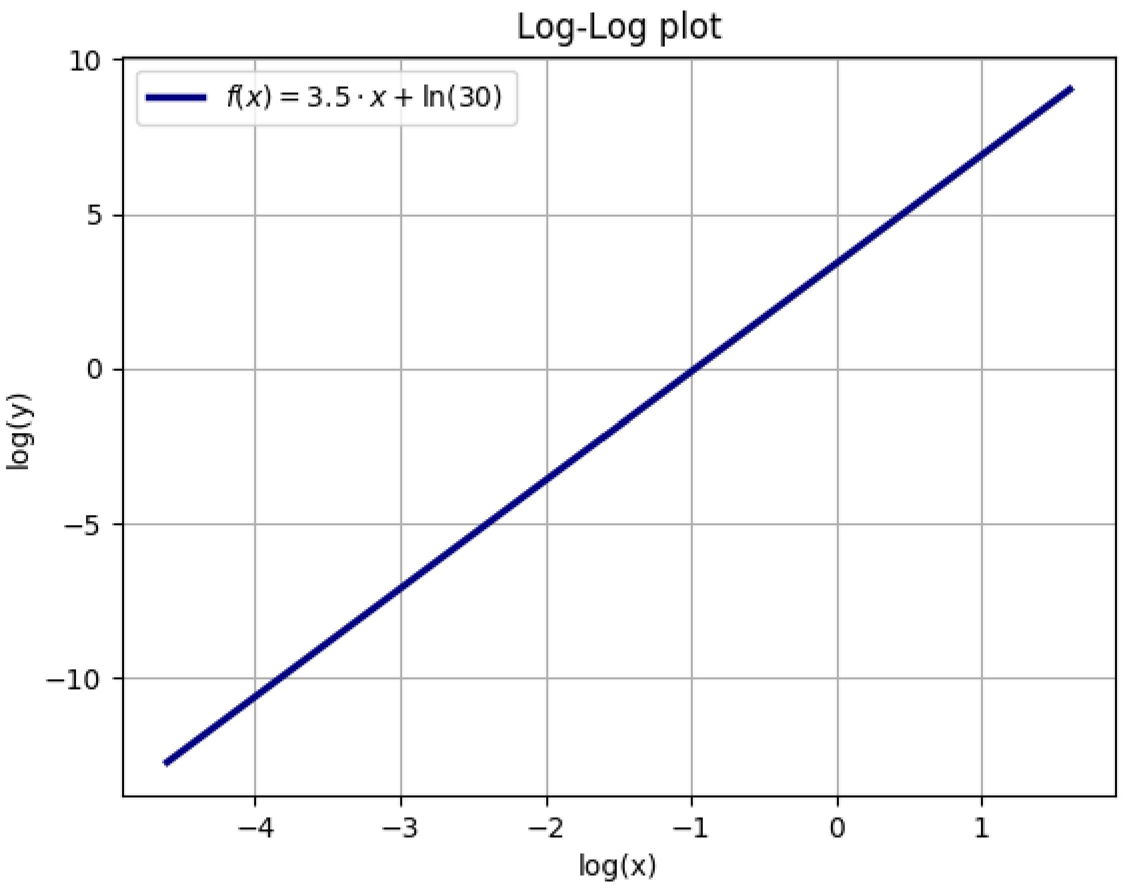
LogLog graphing

Let y(x) = A * x^a, for example A=30 and a=3.5. Taking the natural logarithm (ln) of both sides yields (using the common rules for logarithms): ln(y) = ln(A * x^a) = ln(A) + ln(x^a) = ln(A) + a * ln(x). Thus, a plot with logarithmic axes for both x and y will be a linear curve. The slope of this curve is the exponent a of y(x), while the y-intercept y(0) is the natural logarithm of A, ln(A) = ln(30) = 3.401.

The following example illustrates the relation between an exponential function and the linear loglog plot (the function is y = A * x^a with A=30 and a=3.5):

```
import numpy as np
import matplotlib.pyplot as plt
A = 30
a = 3.5
x = np.linspace(0.01, 5, 10000)
y = A * x**a

ax = plt.gca()
plt.plot(x, y, linewidth=2.5, color='navy', label=r'$f(x) = 30 \cdot x^{3.5}$')
plt.legend(loc='upper left')
plt.xlabel(r'x')
plt.ylabel(r'y')
ax.grid(True)
plt.title(r'Normal plot')
plt.show()
plt.clf()

xlog = np.log(x)
ylog = np.log(y)
ax = plt.gca()
plt.plot(xlog, ylog, linewidth=2.5, color='navy', label=r'$f(x) = 3.5\cdot x + \ln(30)$')
plt.legend(loc='best')
plt.xlabel(r'log(x)')
plt.ylabel(r'log(y)')
ax.grid(True)
plt.title(r'Log-Log plot')
plt.show()
plt.clf()
```


Normal plot
$f(x) = 30 \cdot x^{3.5}$
8000
6000
4000
2000
0
y
0
1
2
3
4
5
x


Log-Log plot
$f(x) = 3.5 \cdot x + \ln(30)$
10
5
0
−5
−10
log(y)
−4
−3
−2
−1
0
1
log(x)

Read LogLog Graphing online: http://www.riptutorial.com/matplotlib/topic/10145/loglog-graphing

Chapter 17: Multiple Plots

Syntax

- List item

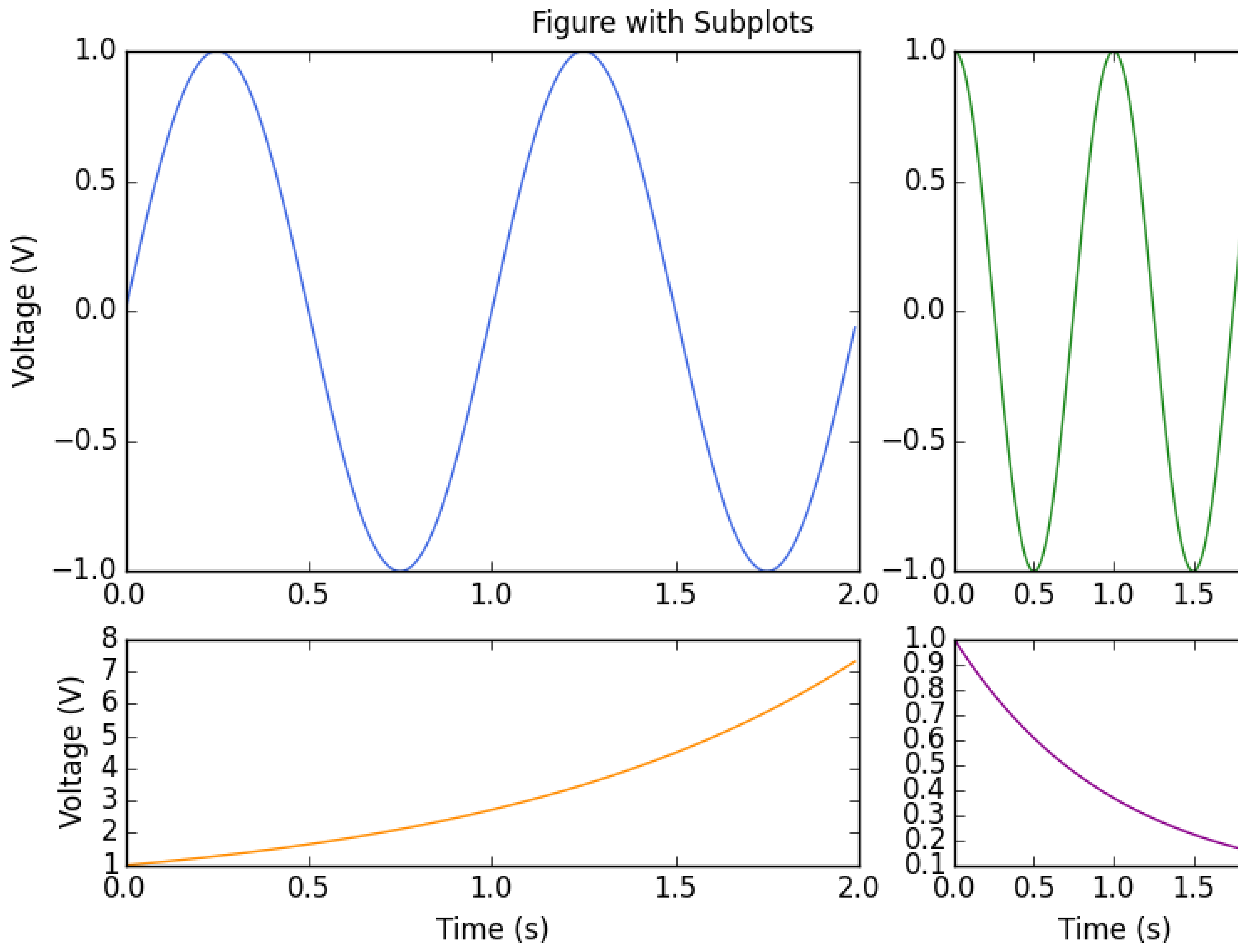
Examples

Multiple Plots with gridspec

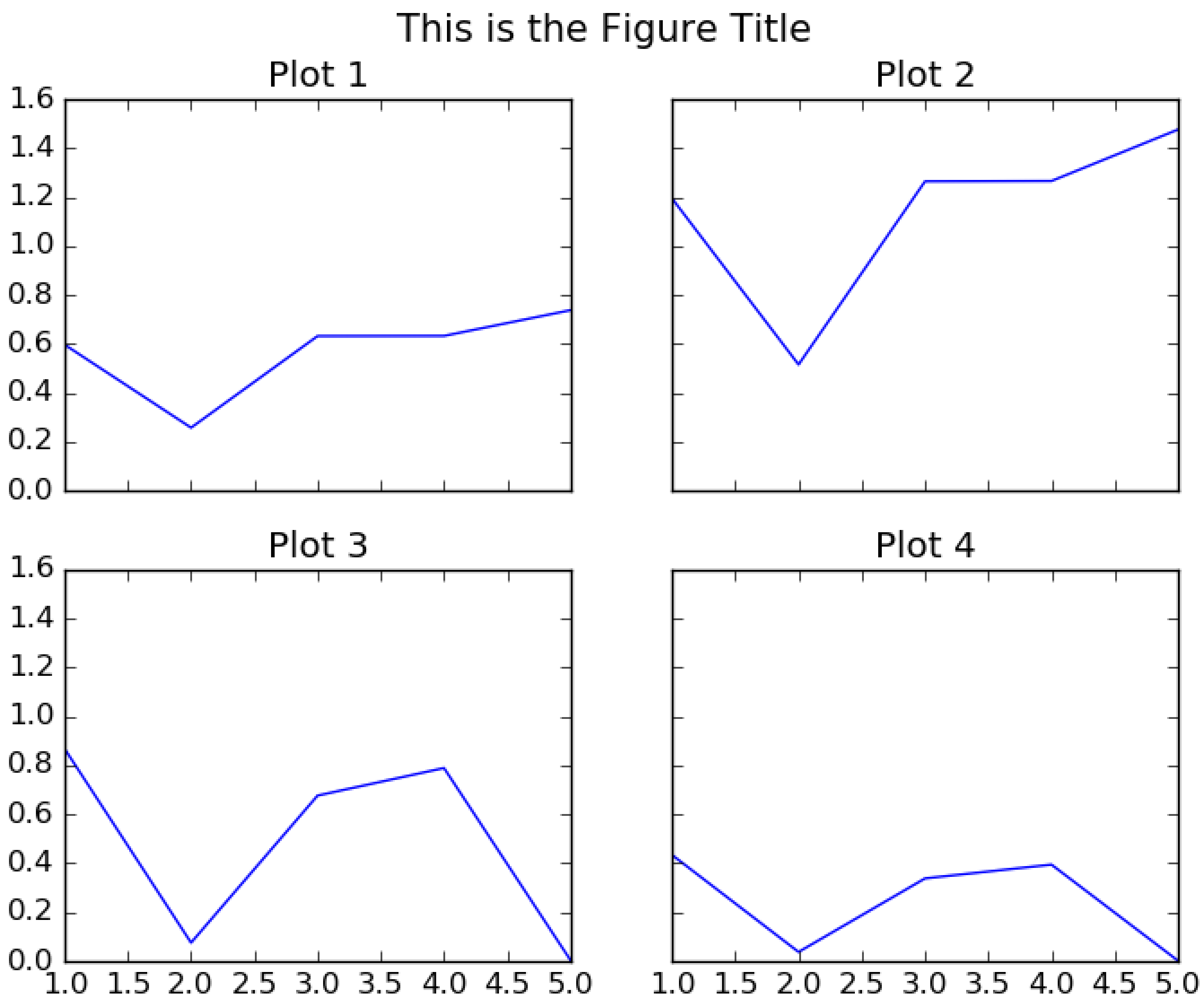
The `gridspec` package allows more control over the placement of subplots. It makes it much easier to control the margins of the plots and the spacing between the individual subplots. In addition, it allows for different sized axes on the same figure by defining axes which take up multiple grid locations.

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.gridspec import  GridSpec

# Make some data
t = np.arange(0, 2, 0.01)
y1 = np.sin(2*np.pi * t)
y2 = np.cos(2*np.pi * t)
y3 = np.exp(t)
y4 = np.exp(-t)

# Initialize the grid with 3 rows and 3 columns
ncols = 3
nrows = 3
grid = GridSpec(nrows, ncols,
                left=0.1, bottom=0.15, right=0.94, top=0.94, wspace=0.3, hspace=0.3)

fig = plt.figure(0)
fig.clf()

# Add axes which can span multiple grid boxes
ax1 = fig.add_subplot(grid[0:2, 0:2])
ax2 = fig.add_subplot(grid[0:2, 2])
ax3 = fig.add_subplot(grid[2, 0:2])
ax4 = fig.add_subplot(grid[2, 2])

ax1.plot(t, y1, color='royalblue')
ax2.plot(t, y2, color='forestgreen')
ax3.plot(t, y3, color='darkorange')
ax4.plot(t, y4, color='darkmagenta')

# Add labels and titles
fig.suptitle('Figure with Subplots')
ax1.set_ylabel('Voltage (V)')
ax3.set_ylabel('Voltage (V)')
ax3.set_xlabel('Time (s)')
ax4.set_xlabel('Time (s)')
```

This code produces the plot shown below.



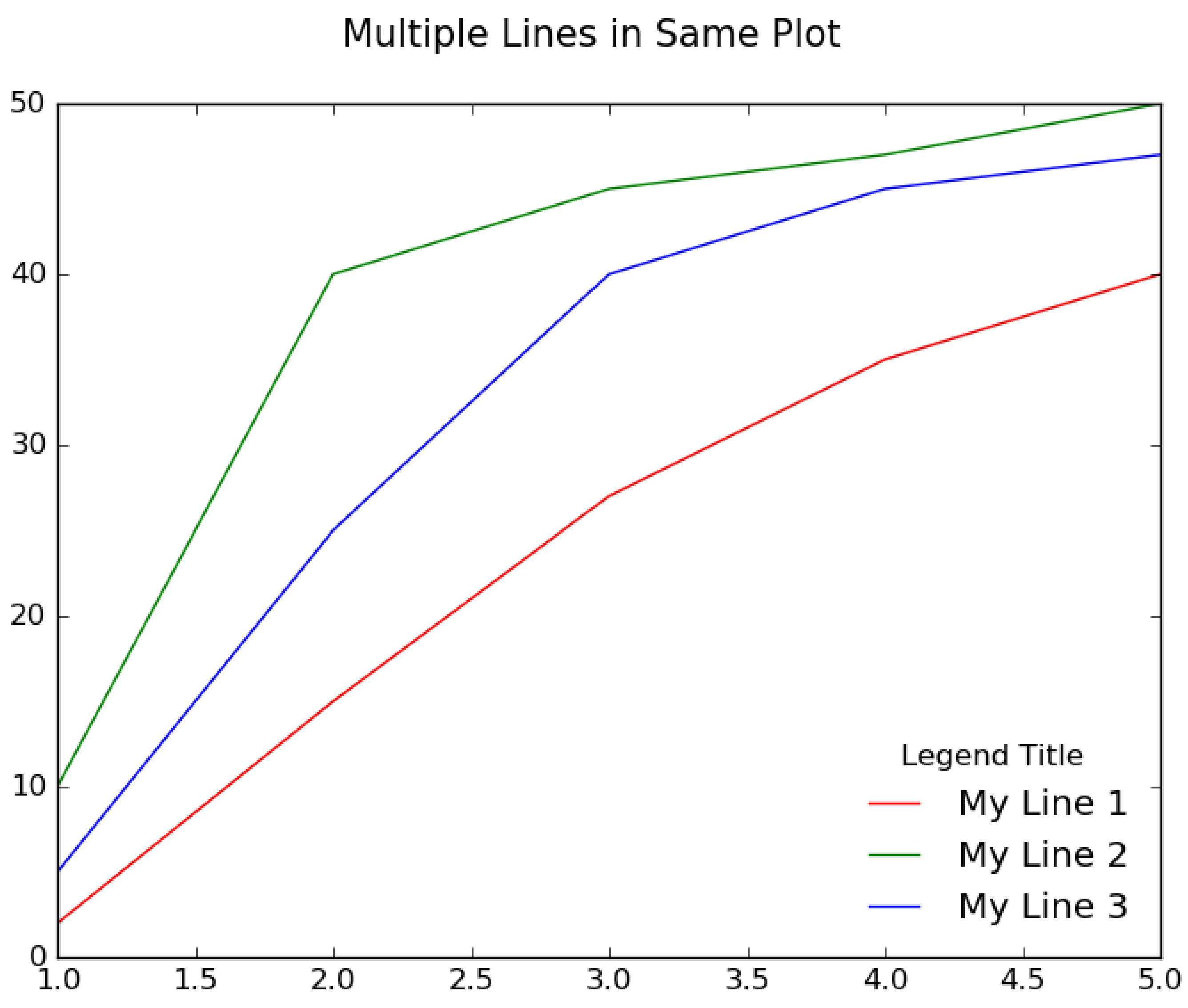
Grid of Subplots using subplot



```
"""
================================================================================
CREATE A 2 BY 2 GRID OF SUB-PLOTS WITHIN THE SAME FIGURE.
================================================================================
"""
import matplotlib.pyplot as plt

# The data
x = [1,2,3,4,5]
y1 = [0.59705847, 0.25786401, 0.63213726, 0.63287317, 0.73791151]
y2 = [1.19411694, 0.51572803, 1.26427451, 1.26574635, 1.47582302]
y3 = [0.86793828, 0.07563408, 0.67670068, 0.78932712, 0.0043694]
# 5 more random values
y4 = [0.43396914, 0.03781704, 0.33835034, 0.39466356, 0.0021847]

# Initialise the figure and a subplot axes. Each subplot sharing (showing) the
# same range of values for the x and y axis in the plots.
fig, axes = plt.subplots(2, 2, figsize=(8, 6), sharex=True, sharey=True)

# Set the title for the figure
fig.suptitle('This is the Figure Title', fontsize=15)

# Top Left Subplot
```

```
axes[0,0].plot(x, y1)
axes[0,0].set_title("Plot 1")

# Top Right Subplot
axes[0,1].plot(x, y2)
axes[0,1].set_title("Plot 2")

# Bottom Left Subplot
axes[1,0].plot(x, y3)
axes[1,0].set_title("Plot 3")

# Bottom Right Subplot
axes[1,1].plot(x, y4)
axes[1,1].set_title("Plot 4")

plt.show()
```

Multiple Lines/Curves in the Same Plot



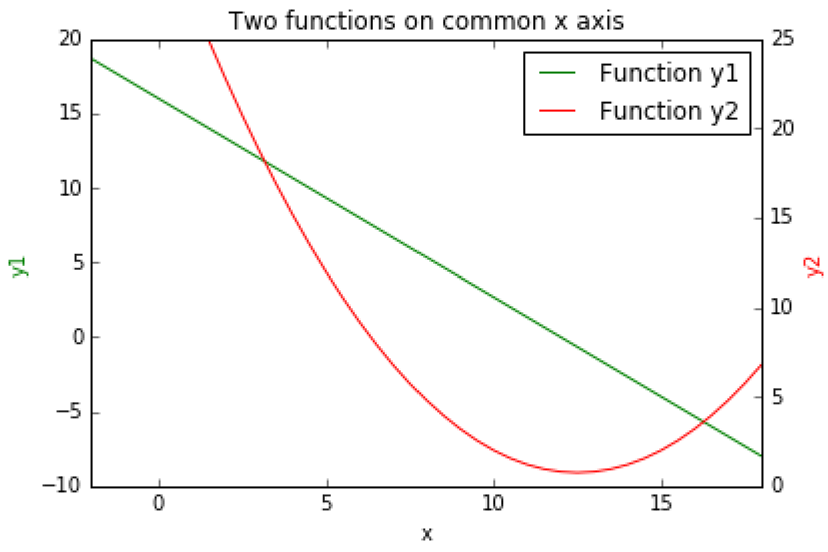
```
"""
================================================================================
                     DRAW MULTIPLE LINES IN THE SAME PLOT
================================================================================
"""
import matplotlib.pyplot as plt

# The data
x =  [1, 2, 3, 4, 5]
y1 = [2,  15, 27, 35, 40]
y2 = [10, 40, 45, 47, 50]
y3 = [5,  25, 40, 45, 47]

# Initialise the figure and axes.
fig, ax = plt.subplots(1, figsize=(8, 6))

# Set the title for the figure
fig.suptitle('Multiple Lines in Same Plot', fontsize=15)

# Draw all the lines in the same plot, assigning a label for each one to be
# shown in the legend.
ax.plot(x, y1, color="red", label="My Line 1")
ax.plot(x, y2, color="green", label="My Line 2")
ax.plot(x, y3, color="blue", label="My Line 3")

# Add a legend, and position it on the lower right (with no box)
plt.legend(loc="lower right", title="Legend Title", frameon=False)

plt.show()
```

A plot of 2 functions on shared x-axis.

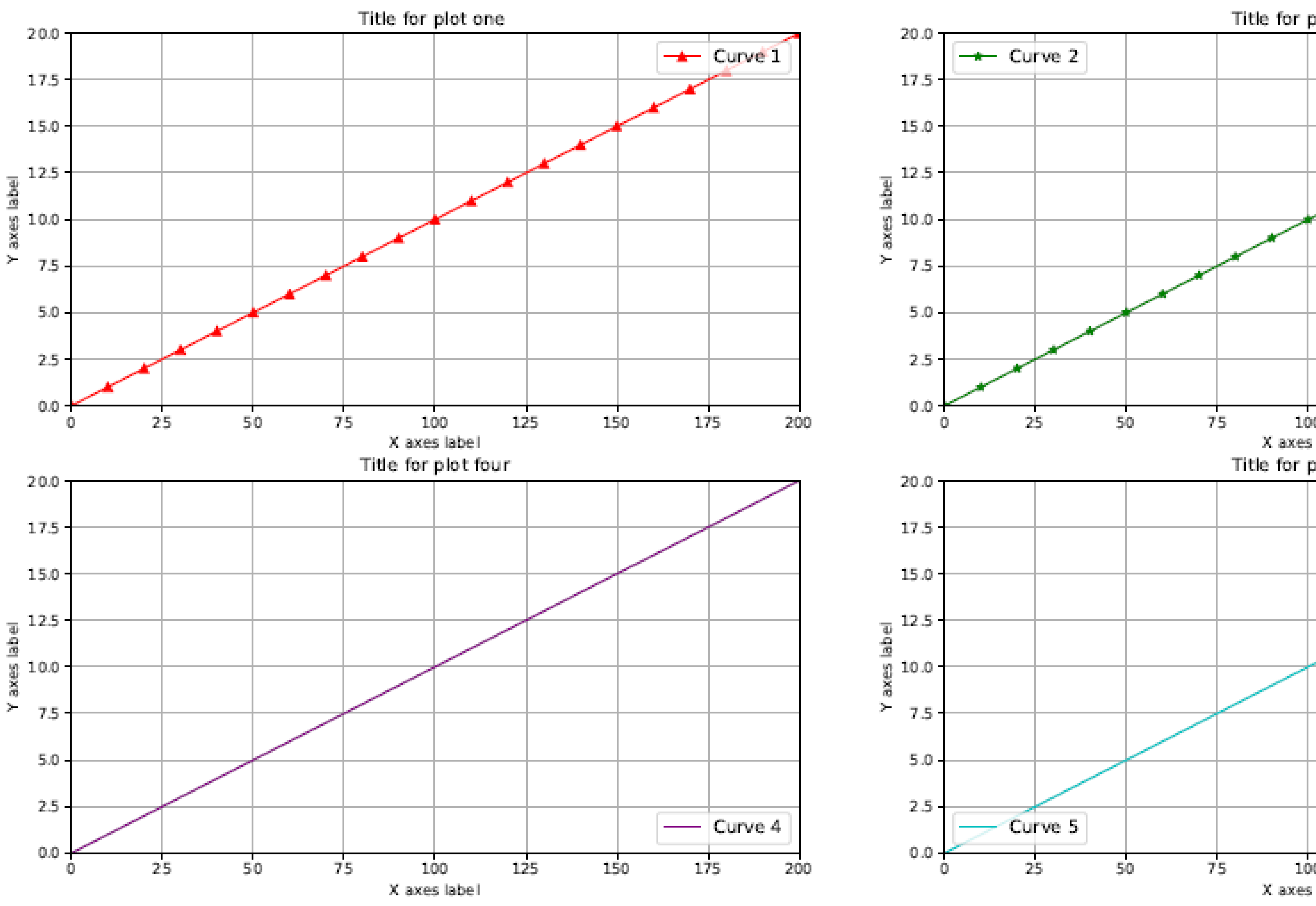
```
import numpy as np
import matplotlib.pyplot as plt

# create some data
x = np.arange(-2, 20, 0.5)                 # values of x
y1 = map(lambda x: -4.0/3.0*x + 16, x)     # values of y1(x)
y2 = map(lambda x: 0.2*x**2 -5*x + 32, x)  # svalues of y2(x)

fig = plt.figure()
ax1 = fig.add_subplot(111)

# create line plot of y1(x)
line1, = ax1.plot(x, y1, 'g', label="Function y1")
ax1.set_xlabel('x')
ax1.set_ylabel('y1', color='g')

# create shared axis for y2(x)
ax2 = ax1.twinx()

# create line plot of y2(x)
line2, = ax2.plot(x, y2, 'r', label="Function y2")
ax2.set_ylabel('y2', color='r')

# set title, plot limits, etc
plt.title('Two functions on common x axis')
plt.xlim(-2, 18)
plt.ylim(0, 25)
```

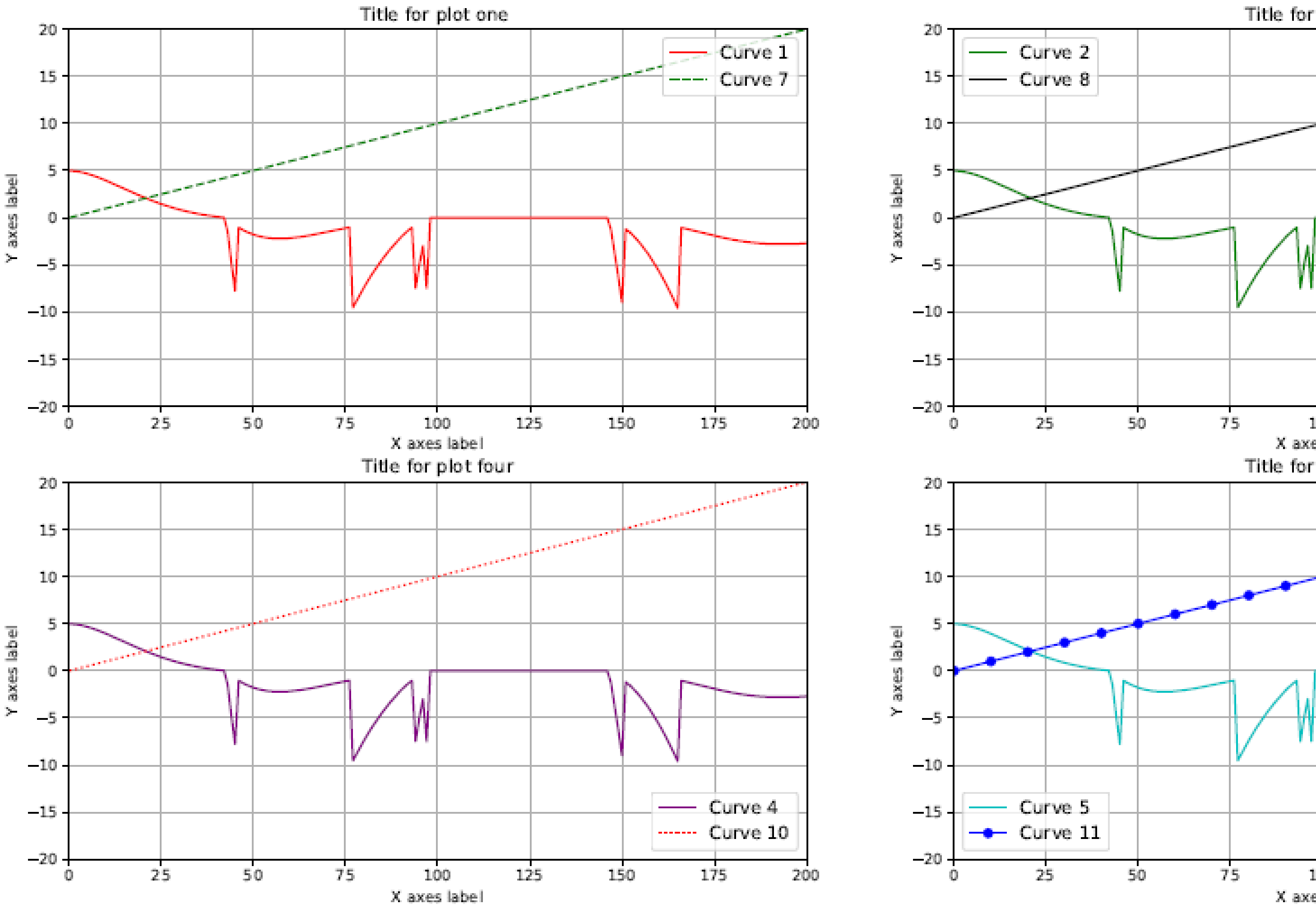
```
# add a legend, and position it on the upper right
plt.legend((line1, line2), ('Function y1', 'Function y2'))

plt.show()
```

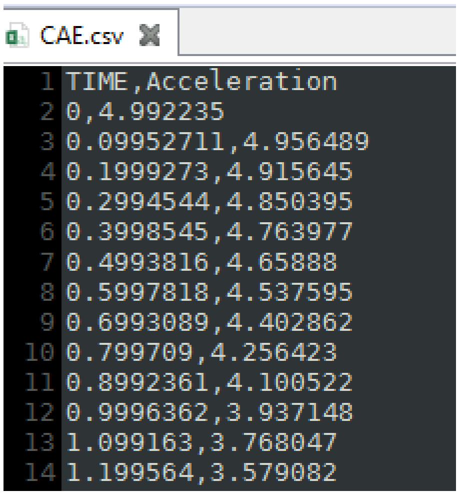
This code produces the plot shown below.



Multiple Plots and Multiple Plot Features


Title for plot one
Curve 1
Title for p
Curve 2
Title for plot four
Curve 4
Title for p
Curve 5
Y axes label
X axes label





```
TIME,Acceleration
0,4.992235
0.09952711,4.956489
0.1999273,4.915645
0.2994544,4.850395
0.3998545,4.763977
0.4993816,4.65888
0.5997818,4.537595
0.6993089,4.402862
0.799709,4.256423
0.8992361,4.100522
0.9996362,3.937148
1.099163,3.768047
1.199564,3.579082
```

```
import matplotlib
matplotlib.use("TKAgg")

# module to save pdf files
from matplotlib.backends.backend_pdf import PdfPages

import matplotlib.pyplot as plt   # module to plot

import pandas as pd  # module to read csv file
```

```
# module to allow user to select csv file
from tkinter.filedialog import askopenfilename

# module to allow user to select save directory
from tkinter.filedialog import askdirectory


#==============================================================================
#  User chosen Data for plots
#==============================================================================

# User choose csv file then read csv file
filename = askopenfilename() # user selected file
data = pd.read_csv(filename, delimiter=',')

# check to see if data is reading correctly
#print(data)

#==============================================================================
#  Plots on two different Figures and sets the size of the figures
#==============================================================================

# figure size = (width,height)
f1 = plt.figure(figsize=(30,10))
f2 = plt.figure(figsize=(30,10))

#------------------------------------------------------------------------------
#  Figure 1 with 6 plots
#------------------------------------------------------------------------------

# plot one
# Plot column labeled TIME from csv file and color it red
# subplot(2 Rows, 3 Columns, First subplot,)
ax1 = f1.add_subplot(2,3,1)
ax1.plot(data[["TIME"]], label = 'Curve 1', color = "r", marker = '^', markevery = 10)
# added line marker triangle


# plot two
# plot column labeled TIME from csv file and color it green
# subplot(2 Rows, 3 Columns, Second subplot)
ax2 = f1.add_subplot(2,3,2)
ax2.plot(data[["TIME"]], label = 'Curve 2', color = "g", marker = '*', markevery = 10)
# added line marker star


# plot three
# plot column labeled TIME from csv file and color it blue
# subplot(2 Rows, 3 Columns, Third subplot)
ax3 = f1.add_subplot(2,3,3)
ax3.plot(data[["TIME"]], label = 'Curve 3', color = "b", marker = 'D', markevery = 10)
# added line marker diamond


# plot four
# plot column labeled TIME from csv file and color it purple
# subplot(2 Rows, 3 Columns, Fourth subplot)
ax4 = f1.add_subplot(2,3,4)
ax4.plot(data[["TIME"]], label = 'Curve 4', color = "#800080")
```

```
# plot five
# plot column labeled TIME from csv file and color it cyan
# subplot(2 Rows, 3 Columns, Fifth subplot)
ax5 = f1.add_subplot(2,3,5)
ax5.plot(data[["TIME"]], label = 'Curve 5', color = "c")


# plot six
# plot column labeled TIME from csv file and color it black
# subplot(2 Rows, 3 Columns, Sixth subplot)
ax6 = f1.add_subplot(2,3,6)
ax6.plot(data[["TIME"]], label = 'Curve 6', color = "k")

#-------------------------------------------------------------------------------
# Figure 2 with 6 plots
#-------------------------------------------------------------------------------

# plot one
# Curve 1: plot column labeled Acceleration from csv file and color it red
# Curve 2: plot column labeled     TIME     from csv file and color it green
# subplot(2 Rows, 3 Columns, First subplot)
ax10 = f2.add_subplot(2,3,1)
ax10.plot(data[["Acceleration"]], label = 'Curve 1', color = "r")
ax10.plot(data[["TIME"]], label = 'Curve 7', color="g", linestyle ='--')
# dashed line


# plot two
# Curve 1: plot column labeled Acceleration from csv file and color it green
# Curve 2: plot column labeled     TIME     from csv file and color it black
# subplot(2 Rows, 3 Columns, Second subplot)
ax20 = f2.add_subplot(2,3,2)
ax20.plot(data[["Acceleration"]], label = 'Curve 2', color = "g")
ax20.plot(data[["TIME"]], label = 'Curve 8', color = "k", linestyle ='-')
# solid line (default)


# plot three
# Curve 1: plot column labeled Acceleration from csv file and color it blue
# Curve 2: plot column labeled     TIME     from csv file and color it purple
# subplot(2 Rows, 3 Columns, Third subplot)
ax30 = f2.add_subplot(2,3,3)
ax30.plot(data[["Acceleration"]], label = 'Curve 3', color = "b")
ax30.plot(data[["TIME"]], label = 'Curve 9', color = "#800080", linestyle ='-.')
# dash_dot line


# plot four
# Curve 1: plot column labeled Acceleration from csv file and color it purple
# Curve 2: plot column labeled     TIME     from csv file and color it red
# subplot(2 Rows, 3 Columns, Fourth subplot)
ax40 = f2.add_subplot(2,3,4)
ax40.plot(data[["Acceleration"]], label = 'Curve 4', color = "#800080")
ax40.plot(data[["TIME"]], label = 'Curve 10', color = "r", linestyle =':')
# dotted line


# plot five
# Curve 1: plot column labeled Acceleration from csv file and color it cyan
# Curve 2: plot column labeled     TIME     from csv file and color it blue
# subplot(2 Rows, 3 Columns, Fifth subplot)
```

```
ax50 = f2.add_subplot(2,3,5)
ax50.plot(data[["Acceleration"]], label = 'Curve 5', color = "c")
ax50.plot(data[["TIME"]], label = 'Curve 11', color = "b", marker = 'o', markevery = 10)
# added line marker circle


# plot six
# Curve 1: plot column labeled Acceleration from csv file and color it black
# Curve 2: plot column labeled     TIME     from csv file and color it cyan
# subplot(2 Rows, 3 Columns, Sixth subplot)
ax60 = f2.add_subplot(2,3,6)
ax60.plot(data[["Acceleration"]], label = 'Curve 6', color = "k")
ax60.plot(data[["TIME"]], label = 'Curve 12', color = "c", marker = 's', markevery = 10)
# added line marker square


#================================================================================
#  Figure Plot options
#================================================================================

#--------------------------------------------------------------------------------
#  Figure 1 options
#--------------------------------------------------------------------------------

#switch to figure one for editing
plt.figure(1)

# Plot one options
ax1.legend(loc='upper right', fontsize='large')
ax1.set_title('Title for plot one ')
ax1.set_xlabel('X axes label')
ax1.set_ylabel('Y axes label')
ax1.grid(True)
ax1.set_xlim([0,200])
ax1.set_ylim([0,20])

# Plot two options
ax2.legend(loc='upper left', fontsize='large')
ax2.set_title('Title for plot two ')
ax2.set_xlabel('X axes label')
ax2.set_ylabel('Y axes label')
ax2.grid(True)
ax2.set_xlim([0,200])
ax2.set_ylim([0,20])

# Plot three options
ax3.legend(loc='upper center', fontsize='large')
ax3.set_title('Title for plot three ')
ax3.set_xlabel('X axes label')
ax3.set_ylabel('Y axes label')
ax3.grid(True)
ax3.set_xlim([0,200])
ax3.set_ylim([0,20])

# Plot four options
ax4.legend(loc='lower right', fontsize='large')
ax4.set_title('Title for plot four')
ax4.set_xlabel('X axes label')
ax4.set_ylabel('Y axes label')
ax4.grid(True)
ax4.set_xlim([0,200])
```

```
ax4.set_ylim([0,20])

# Plot five options
ax5.legend(loc='lower left', fontsize='large')
ax5.set_title('Title for plot five ')
ax5.set_xlabel('X axes label')
ax5.set_ylabel('Y axes label')
ax5.grid(True)
ax5.set_xlim([0,200])
ax5.set_ylim([0,20])

# Plot six options
ax6.legend(loc='lower center', fontsize='large')
ax6.set_title('Title for plot six')
ax6.set_xlabel('X axes label')
ax6.set_ylabel('Y axes label')
ax6.grid(True)
ax6.set_xlim([0,200])
ax6.set_ylim([0,20])

#-----------------------------------------------------------------------------
#  Figure 2 options
#-----------------------------------------------------------------------------

#switch to figure two for editing
plt.figure(2)

# Plot one options
ax10.legend(loc='upper right', fontsize='large')
ax10.set_title('Title for plot one ')
ax10.set_xlabel('X axes label')
ax10.set_ylabel('Y axes label')
ax10.grid(True)
ax10.set_xlim([0,200])
ax10.set_ylim([-20,20])

# Plot two options
ax20.legend(loc='upper left', fontsize='large')
ax20.set_title('Title for plot two ')
ax20.set_xlabel('X axes label')
ax20.set_ylabel('Y axes label')
ax20.grid(True)
ax20.set_xlim([0,200])
ax20.set_ylim([-20,20])

# Plot three options
ax30.legend(loc='upper center', fontsize='large')
ax30.set_title('Title for plot three ')
ax30.set_xlabel('X axes label')
ax30.set_ylabel('Y axes label')
ax30.grid(True)
ax30.set_xlim([0,200])
ax30.set_ylim([-20,20])

# Plot four options
ax40.legend(loc='lower right', fontsize='large')
ax40.set_title('Title for plot four')
ax40.set_xlabel('X axes label')
ax40.set_ylabel('Y axes label')
ax40.grid(True)
ax40.set_xlim([0,200])
```

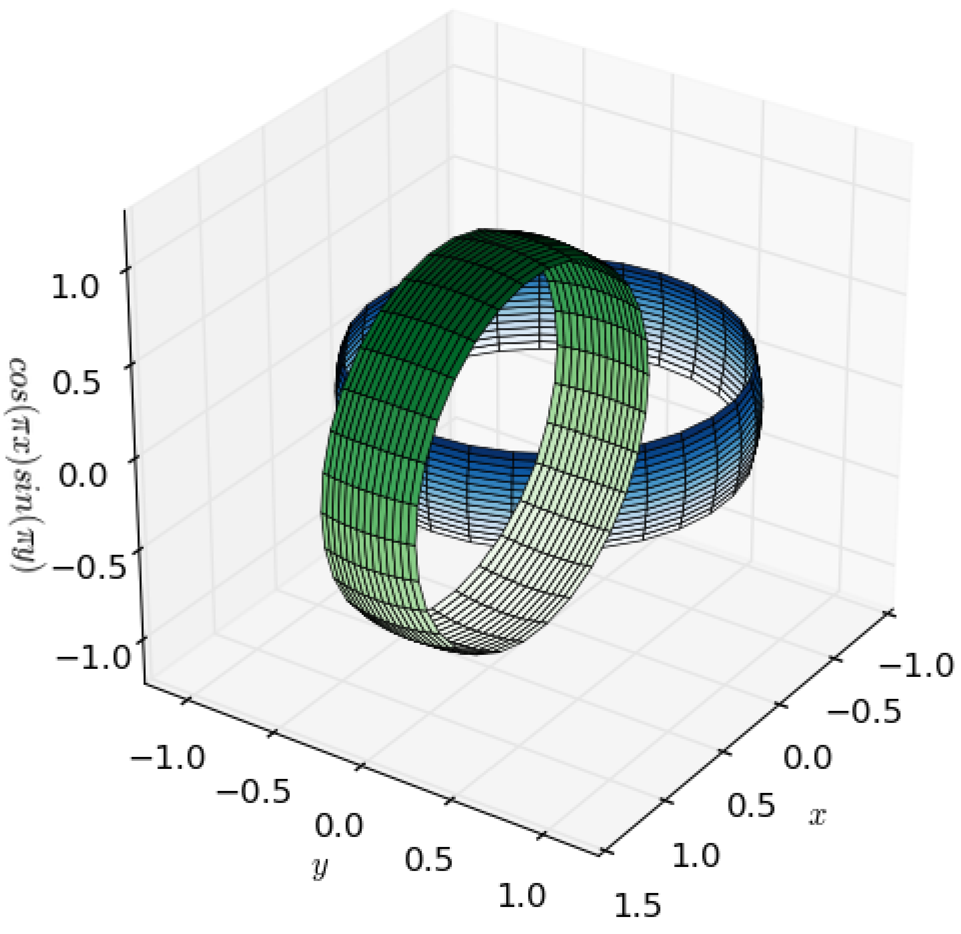
```
ax40.set_ylim([-20,20])

# Plot five options
ax50.legend(loc='lower left', fontsize='large')
ax50.set_title('Title for plot five ')
ax50.set_xlabel('X axes label')
ax50.set_ylabel('Y axes label')
ax50.grid(True)
ax50.set_xlim([0,200])
ax50.set_ylim([-20,20])

# Plot six options
ax60.legend(loc='lower center', fontsize='large')
ax60.set_title('Title for plot six')
ax60.set_xlabel('X axes label')
ax60.set_ylabel('Y axes label')
ax60.grid(True)
ax60.set_xlim([0,200])
ax60.set_ylim([-20,20])


#==============================================================================
#  User chosen file location Save PDF
#==============================================================================

savefilename = askdirectory()# user selected file path
pdf = PdfPages(f'{savefilename}/longplot.pdf')
# using formatted string literals ("f-strings")to place the variable into the string

# save both figures into one pdf file
pdf.savefig(1)
pdf.savefig(2)

pdf.close()


#==============================================================================
#  Show plot
#==============================================================================

# manually set the subplot spacing when there are multiple plots
#plt.subplots_adjust(left=None, bottom=None, right=None, top=None, wspace =None, hspace=None )

# Automaticlly adds space between plots
plt.tight_layout()

plt.show()
```

Read Multiple Plots online: http://www.riptutorial.com/matplotlib/topic/3279/multiple-plots

Chapter 18: Three-dimensional plots

Remarks

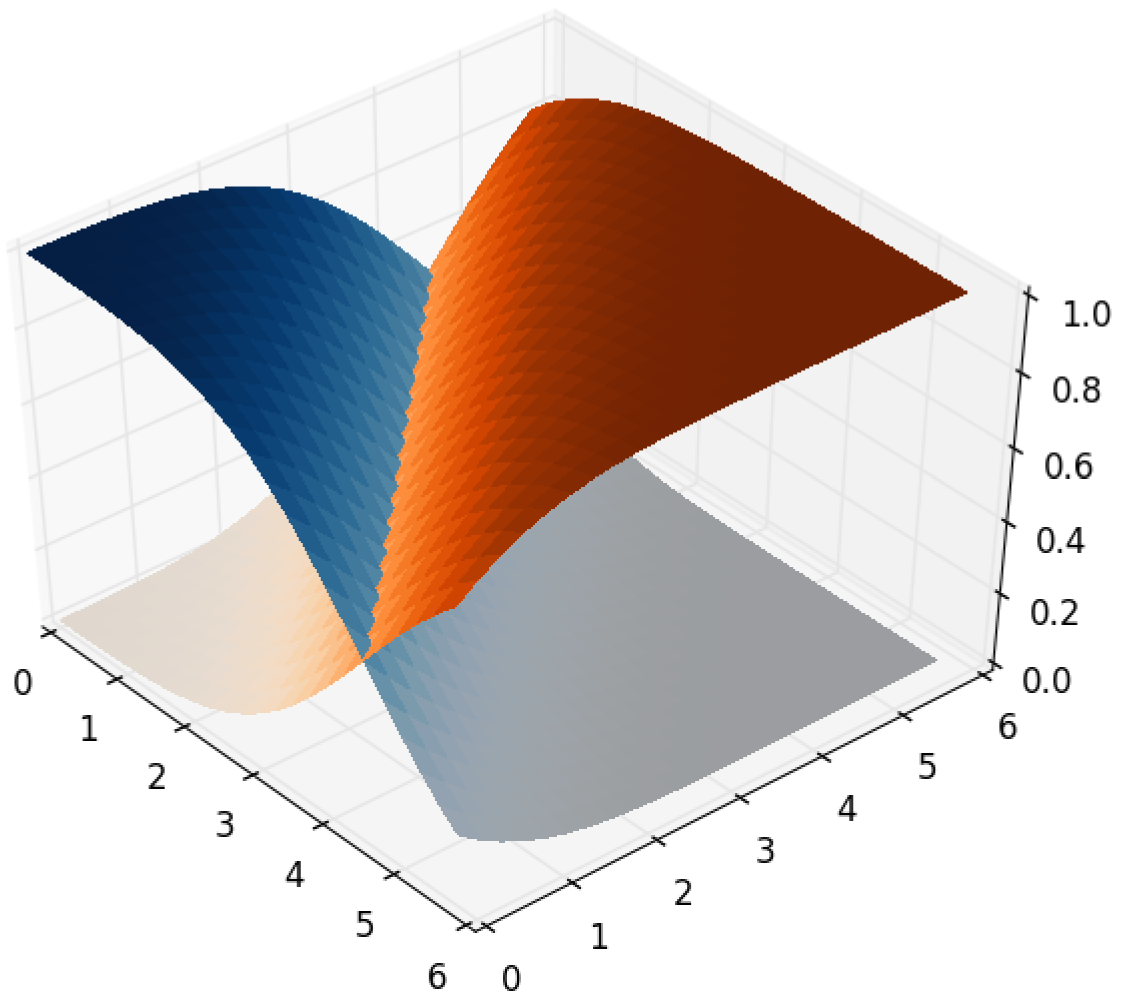
Three-dimensional plotting in matplotlib has historically been a bit of a kludge, as the rendering engine is inherently 2d. The fact that 3d setups are rendered by plotting one 2d chunk after the other implies that there are often rendering issues related to the apparent depth of objects. The core of the problem is that two non-connected objects can either be fully behind, or fully in front of one another, which leads to artifacts as shown in the below figure of two interlocked rings (click for animated gif):

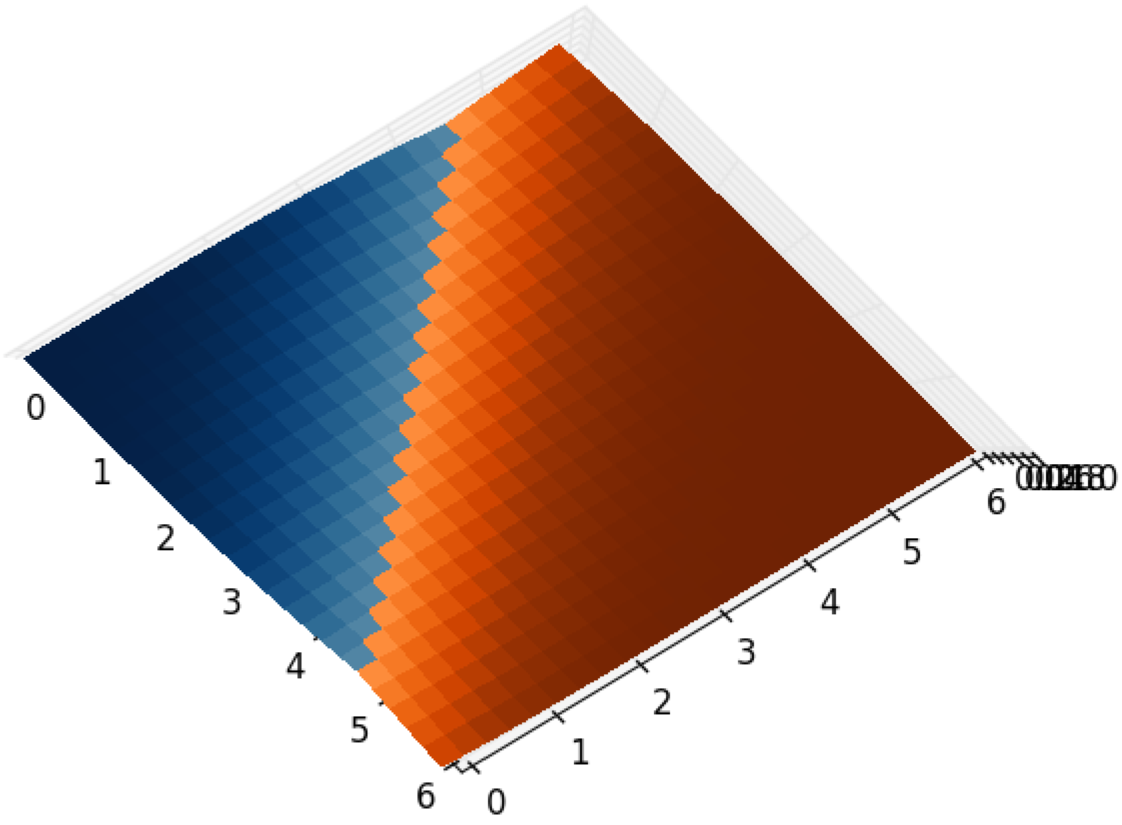


This can however be fixed. This artefact only exists when plotting multiple surfaces on the same plot - as each is rendered as a flat 2D shape, with a single parameter determining the view distance. You will notice that a single complicated surface does not suffer the same problem.

The way to remedy this is to join the plot objects together using transparent bridges:

```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np
from scipy.special import erf

fig = plt.figure()
ax = fig.gca(projection='3d')

X = np.arange(0, 6, 0.25)
Y = np.arange(0, 6, 0.25)
X, Y = np.meshgrid(X, Y)

Z1 = np.empty_like(X)
Z2 = np.empty_like(X)
C1 = np.empty_like(X, dtype=object)
C2 = np.empty_like(X, dtype=object)

for i in range(len(X)):
  for j in range(len(X[0])):
    z1 = 0.5*(erf((X[i,j]+Y[i,j]-4.5)*0.5)+1)
    z2 = 0.5*(erf((-X[i,j]-Y[i,j]+4.5)*0.5)+1)
    Z1[i,j] = z1
    Z2[i,j] = z2

    # If you want to grab a colour from a matplotlib cmap function,
    # you need to give it a number between 0 and 1. z1 and z2 are
    # already in this range, so it just works as is.
    C1[i,j] = plt.get_cmap("Oranges")(z1)
    C2[i,j] = plt.get_cmap("Blues")(z2)


# Create a transparent bridge region
X_bridge = np.vstack([X[-1,:],X[-1,:]])
Y_bridge = np.vstack([Y[-1,:],Y[-1,:]])
Z_bridge = np.vstack([Z1[-1,:],Z2[-1,:]])
color_bridge = np.empty_like(Z_bridge, dtype=object)

color_bridge.fill((1,1,1,0)) # RGBA colour, onlt the last component matters - it represents
the alpha / opacity.

# Join the two surfaces flipping one of them (using also the bridge)
X_full = np.vstack([X, X_bridge, np.flipud(X)])
Y_full = np.vstack([Y, Y_bridge, np.flipud(Y)])
Z_full = np.vstack([Z1, Z_bridge, np.flipud(Z2)])
color_full = np.vstack([C1, color_bridge, np.flipud(C2)])

surf_full = ax.plot_surface(X_full, Y_full, Z_full, rstride=1, cstride=1,
                            facecolors=color_full, linewidth=0,
                            antialiased=False)

plt.show()
```


0
1
2
3
4
5
6
0
1
2
3
4
5
6
0.0
0.2
0.4
0.6
0.8
1.0



Examples

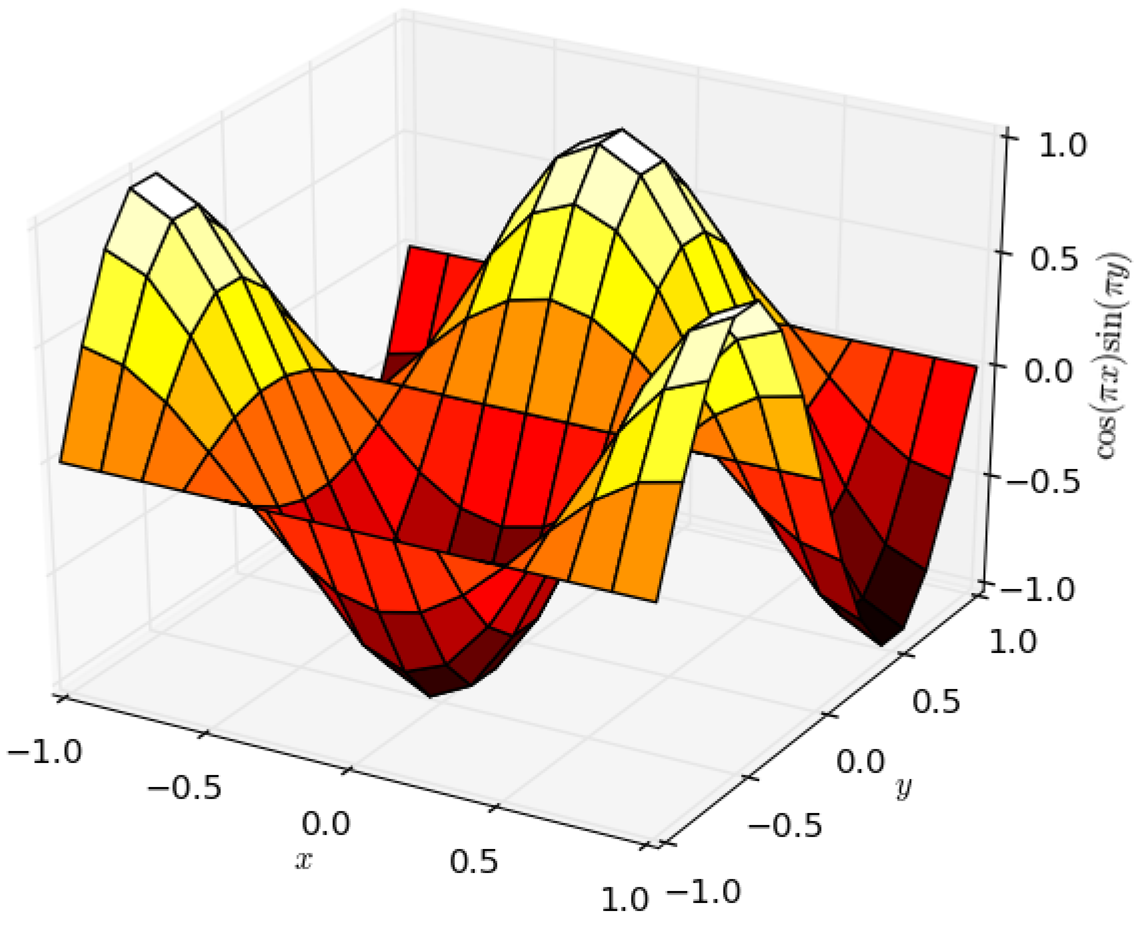
Creating three-dimensional axes

Matplotlib axes are two-dimensional by default. In order to create three-dimensional plots, we need to import the `Axes3D` class from the mplot3d toolkit, that will enable a new kind of projection for an axes, namely `'3d'`:

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
```

Beside the straightforward generalizations of two-dimensional plots (such as line plots, scatter plots, bar plots, contour plots), several surface plotting methods are available, for instance `ax.plot_surface`:

```
# generate example data
import numpy as np
x,y = np.meshgrid(np.linspace(-1,1,15),np.linspace(-1,1,15))
z = np.cos(x*np.pi)*np.sin(y*np.pi)

# actual plotting example
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
# rstride and cstride are row and column stride (step size)
ax.plot_surface(x,y,z,rstride=1,cstride=1,cmap='hot')
ax.set_xlabel(r'$x$')
ax.set_ylabel(r'$y$')
ax.set_zlabel(r'$\cos(\pi x) \sin(\pi y)$')
plt.show()
```



Read Three-dimensional plots online: http://www.riptutorial.com/matplotlib/topic/1880/three-dimensional-plots

Credits

S. No	Chapters	Contributors
1	Getting started with matplotlib	Amitay Stern, ChaoticTwist, Chr, Chris Mueller, Community, dermen, evtoh, farenorth, Josh, jrjc, pmos, Serenity, tacaswell
2	Animations and interactive plotting	FiN, smurfendrek123, user2314737
3	Basic Plots	Franck Dernoncourt, Josh, ml4294, ronrest, Scimonster, Serenity, user2314737
4	Boxplots	Luis
5	Closing a figure window	Brian, David Zwicker
6	Colormaps	Andras Deak, Xevaquor
7	Contour Maps	Eugene Loy, Serenity
8	Coordinates Systems	jure
9	Figures and Axes Objects	David Zwicker, Josh, Serenity, tom
10	Grid Lines and Tick Marks	ronrest
11	Histogram	Yegor Kishilov
12	Image manipulation	Bosoneando
13	Integration with TeX/LaTeX	Andras Deak, Bosoneando, Chris Mueller, Næreen, Serenity
14	Legends	Andras Deak, Franck Dernoncourt, ronrest, saintsfan342000, Serenity
15	LogLog Graphing	ml4294
16	Multiple Plots	Chris Mueller, Robert Branam, ronrest, swatchai
17	Three-dimensional plots	Andras Deak, Serenity, will