—

MATPLOTLIB

Matplotlib Basics

Introduction

Matplotlib is the "grandfather" library of data visualization with Python. It was created by John Hunter. He created it to try to replicate MatLab's (another programming language) plotting capabilities in Python. So if you happen to be familiar with matlab, matplotlib will feel natural to you.

It is an excellent 2D and 3D graphics library for generating scientific figures.

Some of the major Pros of Matplotlib are:

- Generally easy to get started for simple plots
- Support for custom labels and texts
- Great control of every element in a figure
- High-quality output in many formats
- Very customizable in general

Matplotlib allows you to create reproducible figures programmatically. Let's learn how to use it! Before continuing this lecture, I encourage you just to explore the official Matplotlib web page: http://matplotlib.org/

Installation

If you are using our environment, its already installed for you. If you are not using our environment (not recommended), you'll need to install matplotlib first with either:

```
conda install matplotlib
```

or

```
pip install matplotlib
```

Importing

Import the `matplotlib.pyplot` module under the name `plt` (the tidy way):

```
pip install matplotlib # for insatlling first time
```

```
Note: you may need to restart the kernel to use updated packages.
```

```
ERROR: Invalid requirement: '#'
```

```
# COMMON MISTAKE!
# DON'T FORGET THE .PYPLOT part
```

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

```
%matplotlib inline
```

NOTE: If you are using an older version of jupyter, you need to run a "magic" command to see the plots inline with the notebook. Users of jupyter notebook 1.0 and above, don't need to run the cell below:

Basic Example

Let's walk through a very simple example using two numpy arrays:

Basic Array Plot

Let's walk through a very simple example using two numpy arrays. You can also use lists, but most likely you'll be passing numpy arrays or pandas columns (which essentially also behave like arrays).

The data we want to plot:

```
import numpy as np
```

```
x=np.arange(0,10)
y=2*x
x,y
```

```
(array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]),
 array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18]))
```

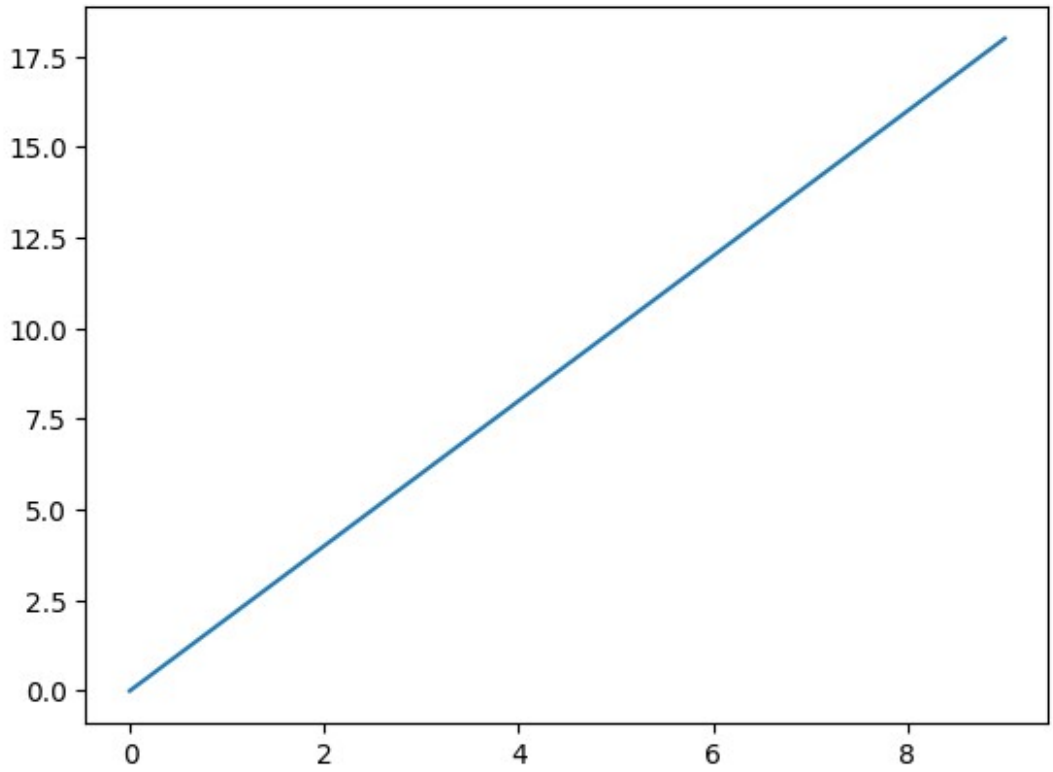
Using Matplotlib with plt.plot() function calls

Basic Matplotlib Commands

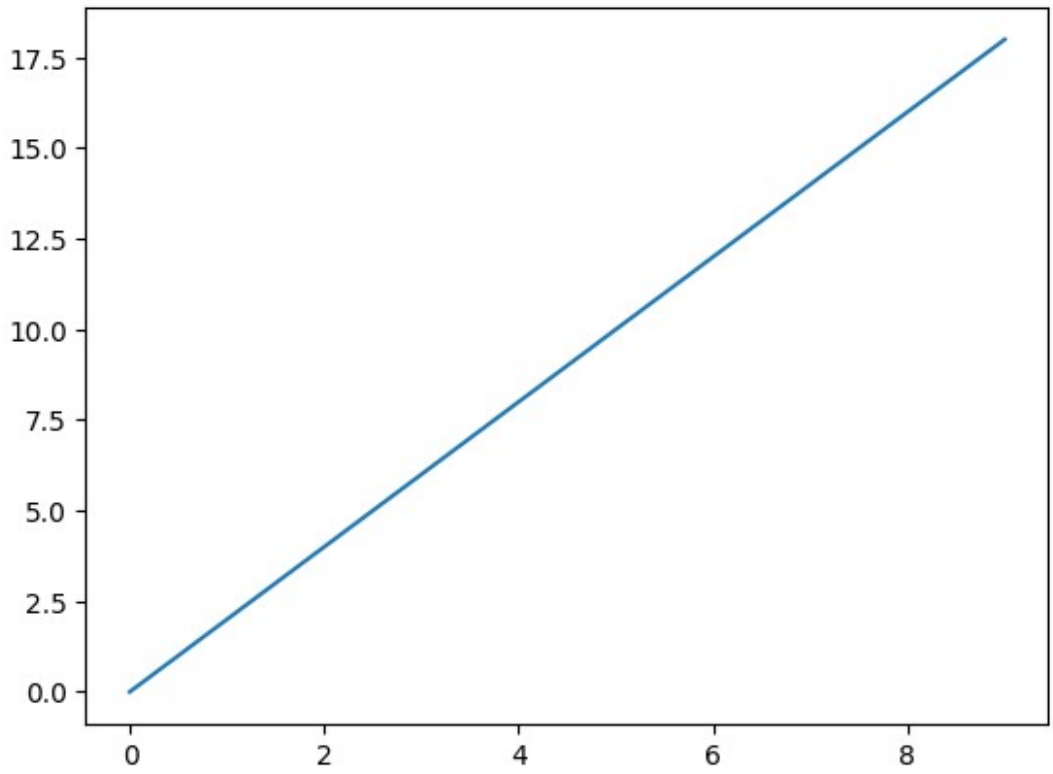
We can create a very simple line plot using the following (I encourage you to pause and use Shift+Tab along the way to check out the document strings for the functions we are using).

```
plt.plot(x,y) # Here matplotlib.lines.Line2D is type of plot and
0x2119b23fa60 is location in memory, it can be
                                                    # remove by
adding plt.show or by adding ; at last
```

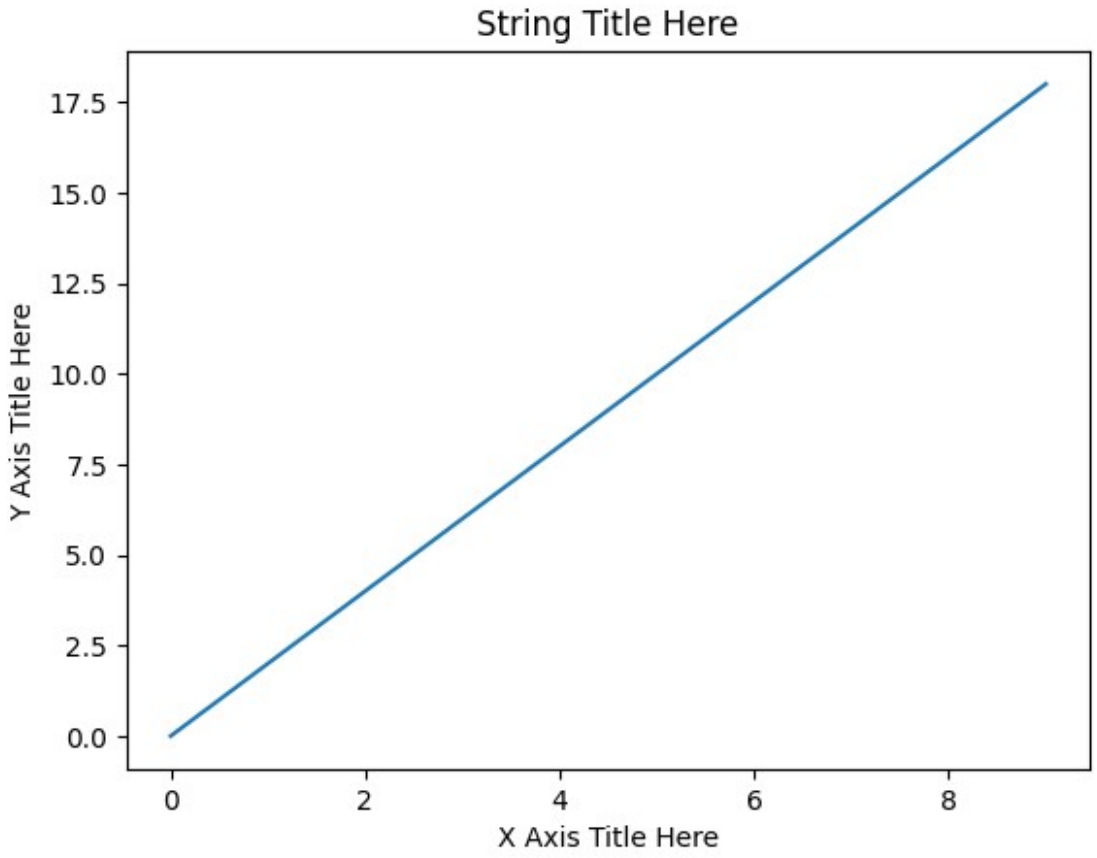
```
[<matplotlib.lines.Line2D at 0x22223b247c0>]
```


17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
0
2
4
6
8

```
plt.plot(x,y);
```

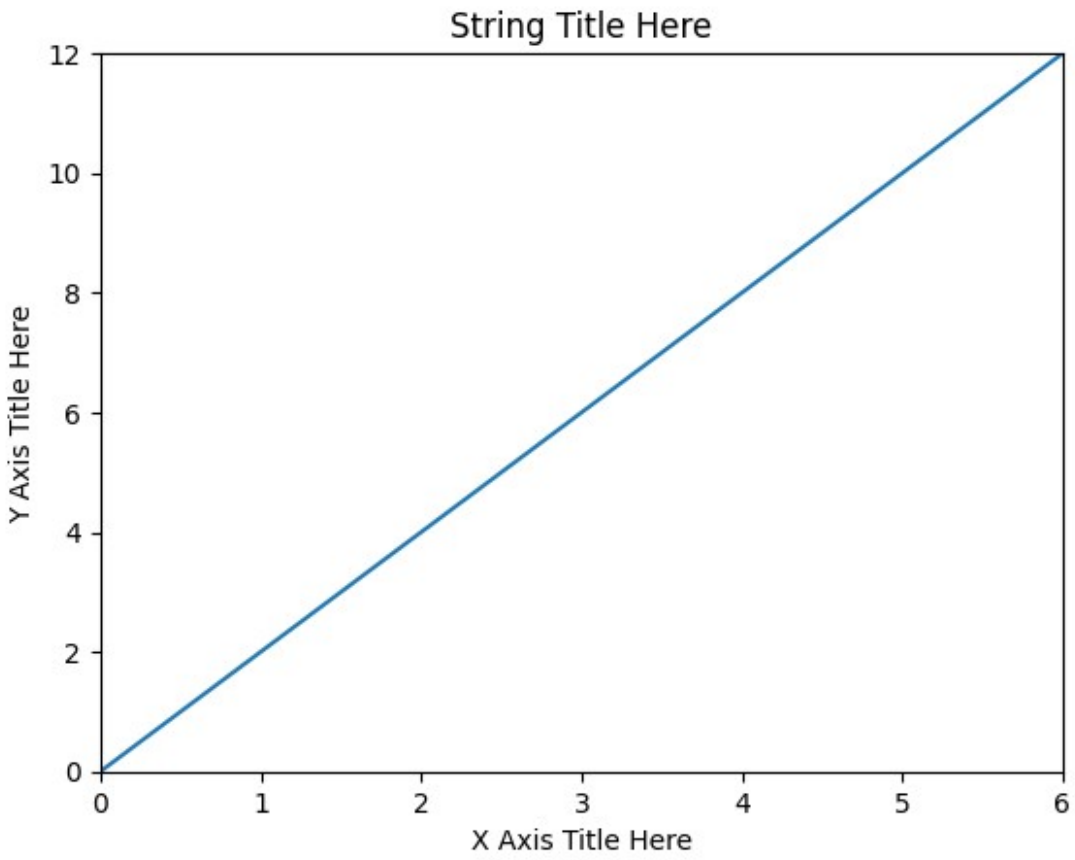

17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
0
2
4
6
8

```
# Adding Titles
plt.plot(x, y)
plt.xlabel('X Axis Title Here')
plt.ylabel('Y Axis Title Here')
plt.title('String Title Here')
plt.show() # Required for non-jupyter users , but also removes Out[]
info
```



Editing more figure parameters

```
plt.plot(x, y)
plt.xlabel('X Axis Title Here')
plt.ylabel('Y Axis Title Here')
plt.title('String Title Here')
plt.xlim(0,6) # Lower Limit, Upper Limit
plt.ylim(0,12) # Lower Limit, Upper Limit
plt.show() # Required for non-jupyter users , but also removes Out[]
infoaa
```

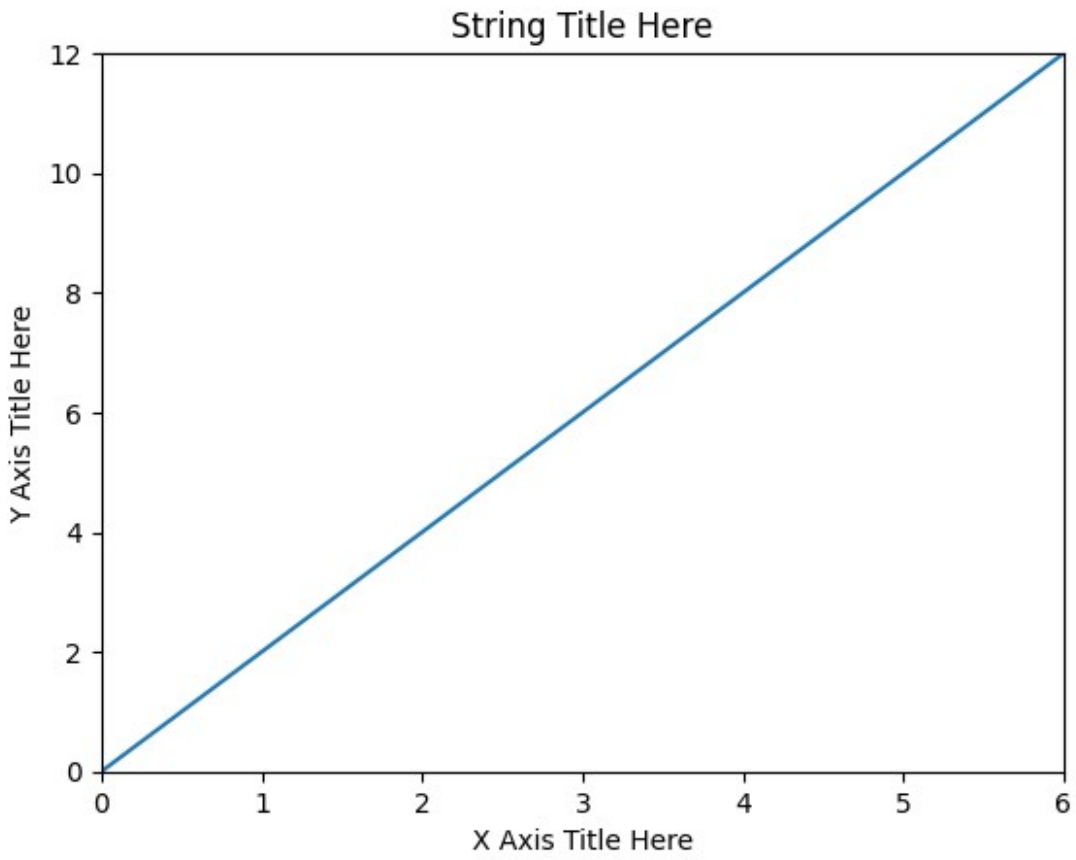


Exporting a plot

```
help(plt.savefig)
```

Saving graphs

```
plt.plot(x, y)
plt.xlabel('X Axis Title Here')
plt.ylabel('Y Axis Title Here')
plt.title('String Title Here')
plt.xlim(0,6)
plt.ylim(0,12)
plt.savefig('D:\\Study\\example.png')
```



Matplotlib Figure Object

```
axes=fig.add_axes([.5,.5,1,1])
axes.plot(x,y);
```

Matplotlib Object Oriented Method

Now that we've seen the basics, let's break it all down with a more formal introduction of Matplotlib's Object Oriented API. This means we will instantiate figure objects and then call methods or attributes from that object.

The Data

```
a=np.linspace(0,10,11)
```

```
b=a**4
```

```
a
```

```
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10.])
```

```
b
```

```
array([0.000e+00, 1.000e+00, 1.600e+01, 8.100e+01, 2.560e+02,
6.250e+02,
       1.296e+03, 2.401e+03, 4.096e+03, 6.561e+03, 1.000e+04])
```

```
x=np.arange(0,10)
x
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
 y
```

```
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])
```

Creating a Figure

The main idea in using the more formal Object Oriented method is to create figure objects and then just call methods or attributes off of that object. This approach is nicer when dealing with a canvas that has multiple plots on it.

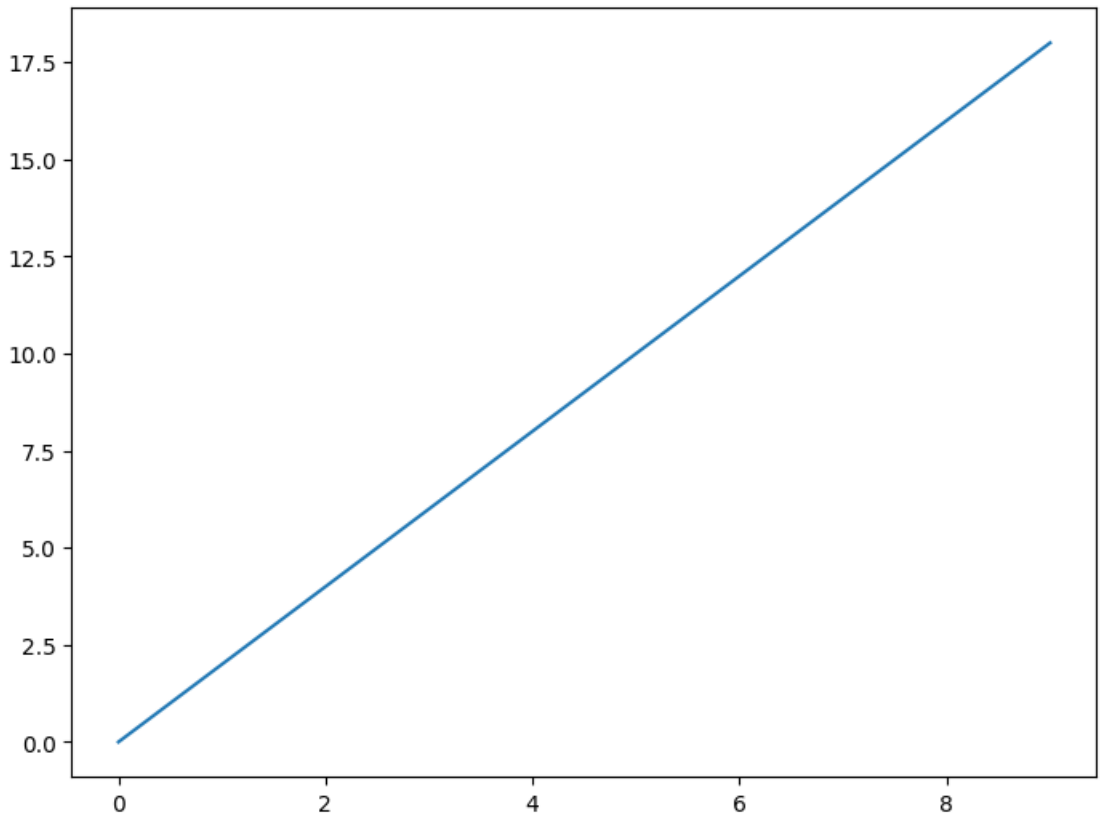
```
# Creates blank canvas
fig = plt.figure()
```

```
<Figure size 640x480 with 0 Axes>
```

NOTE: ALL THE COMMANDS NEED TO GO IN THE SAME CELL!

To begin we create a figure instance. Then we can add axes to that figure:

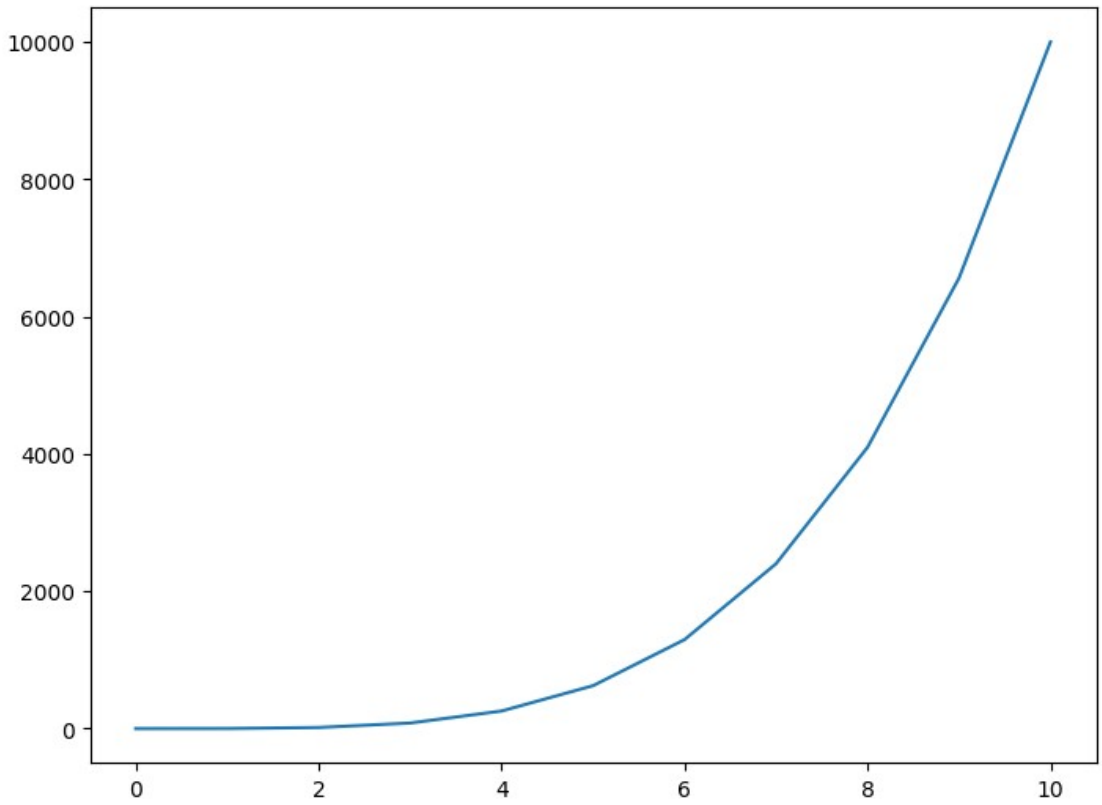
```
# Create Figure (empty canvas)
fig = plt.figure()

# Add set of axes to figure
axes = fig.add_axes([0, 0, 1, 1]) # left, bottom, width, height (range
0 to 1)

# Plot on that set of axes
axes.plot(x, y)

plt.show()
```



we have fig = plot.figure() axes =fig.add_axes([0,0,1,1]} Here first 0,0 is where we want origin of graph , 0,0 is bottom left corner , 1,1 will be upper right corner , .5,.5 will be mid of the plot

and next 1,1 is how long x(width) and y(height) of axis we want see examples below

```
# Create Figure (empty canvas)
fig = plt.figure()

# Add set of axes to figure
axes = fig.add_axes([0, 0, 1, 1]) # left, bottom, width, height (range
0 to 1)

# Plot on that set of axes
axes.plot(a, b)

plt.show()
```



Adding another set of axes to the Figure

So far we've only seen one set of axes on this figure object, but we can keep adding new axes on to it at any location and size we want. We can then plot on that new set of axes.

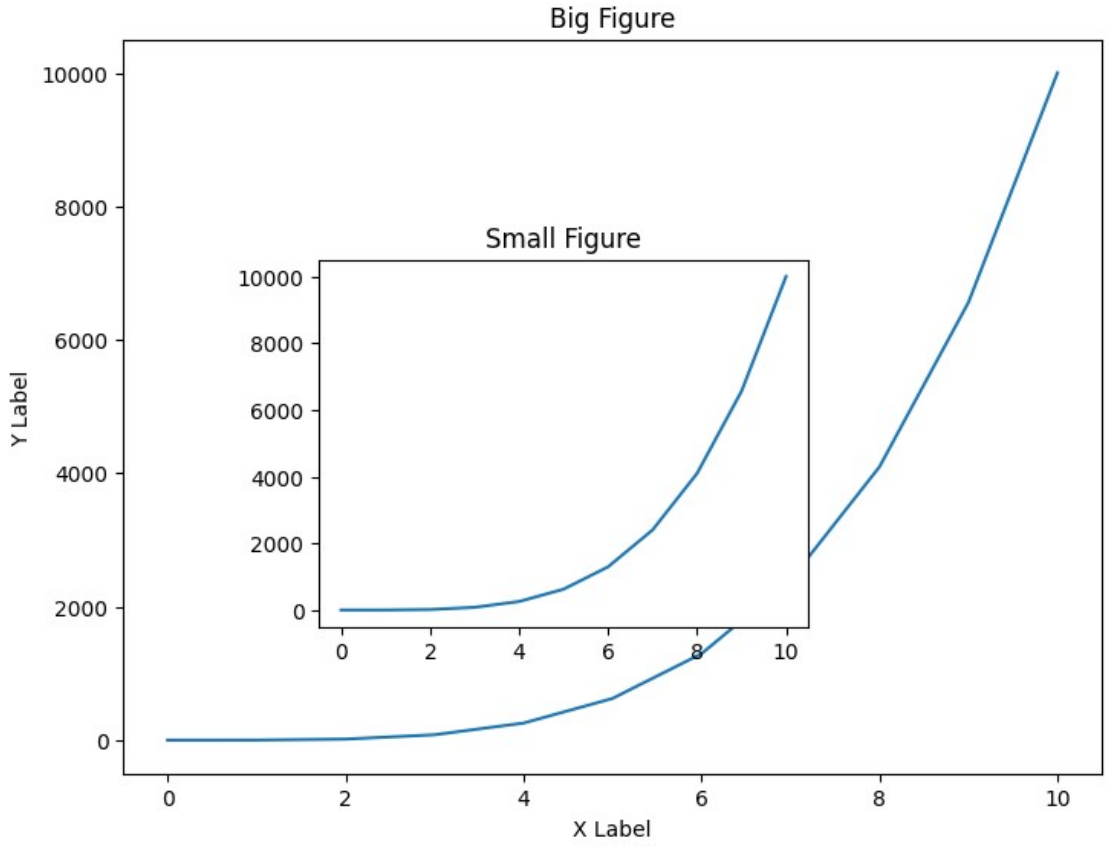
```
type(fig)
```

```
matplotlib.figure.Figure
```

Code is a little more complicated, but the advantage is that we now have full control of where the plot axes are placed, and we can easily add more than one axis to the figure. Note how we're plotting a,b twice here

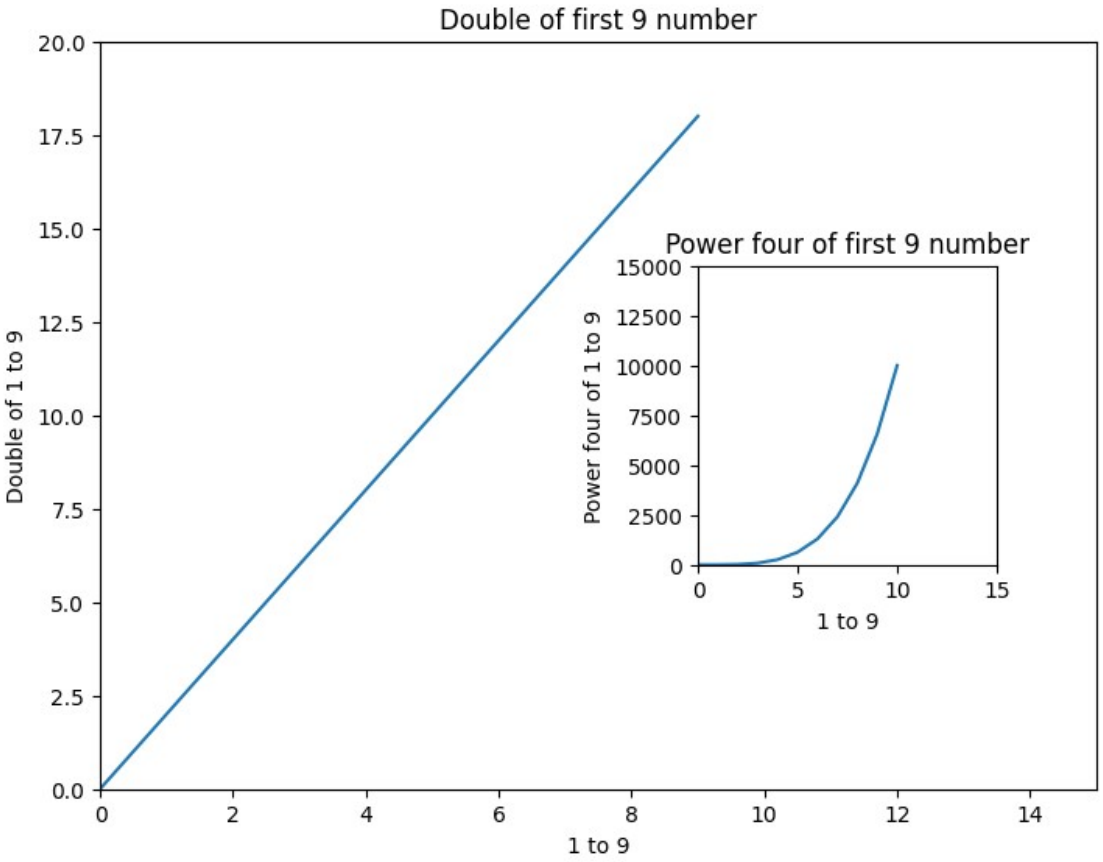
```
# Creates blank canvas
9fig = plt.figure()

axes1 = fig.add_axes([0, 0, 1, 1]) # Large figure
axes2 = fig.add_axes([0.2, 0.2, 0.5, 0.5]) # Smaller figure

# Larger Figure Axes 1
axes1.plot(a, b)

# Use set_ to add to the axes figure
axes1.set_xlabel('X Label')
axes1.set_ylabel('Y Label')
axes1.set_title('Big Figure')
```

```
# Insert Figure Axes 2
axes2.plot(a,b)
axes2.set_title('Small Figure');
```




```
# Another Example
fig= plt.figure()

axes1= fig.add_axes([0,0,1,1])
axes2= fig.add_axes([.6,.3,.3,.4])


axes1.set_xlim(0,15)
axes1.set_ylim(0,20)
axes1.set_xlabel('1 to 9')
axes1.set_ylabel('Double of 1 to 9')
axes1.set_title('Double of first 9 number')


axes2.set_xlim(0,15)
axes2.set_ylim(0,15000)
axes2.set_xlabel('1 to 9')
axes2.set_ylabel('Power four of 1 to 9')
axes2.set_title('Power four of first 9 number')
```

```
axes1.plot(x,y);
axes2.plot(a,b);
```



You can add as many axes on to the same figure as you want, even outside of the main figure if the length and width correspond to this.

```
# Creates blank canvas
fig = plt.figure()

axes1 = fig.add_axes([0, 0, 1, 1]) # Full figure
axes2 = fig.add_axes([0.2, 0.5, 0.25, 0.25]) # Smaller figure
axes3 = fig.add_axes([1, 1, 0.25, 0.25]) # Starts at top right corner!

# Larger Figure Axes 1
axes1.plot(a, b);

# Use set_ to add to the axes figure
axes1.set_xlabel('X Label')
axes1.set_ylabel('Y Label')
axes1.set_title('Big Figure')
```

```
# Insert Figure Axes 2
axes2.plot(a,b);
axes2.set_xlim(8,10)
axes2.set_ylim(4000,10000)
axes2.set_xlabel('X')
axes2.set_ylabel('Y')
axes2.set_title('Zoomed In');

# Insert Figure Axes 3
axes3.plot(a,b);
```

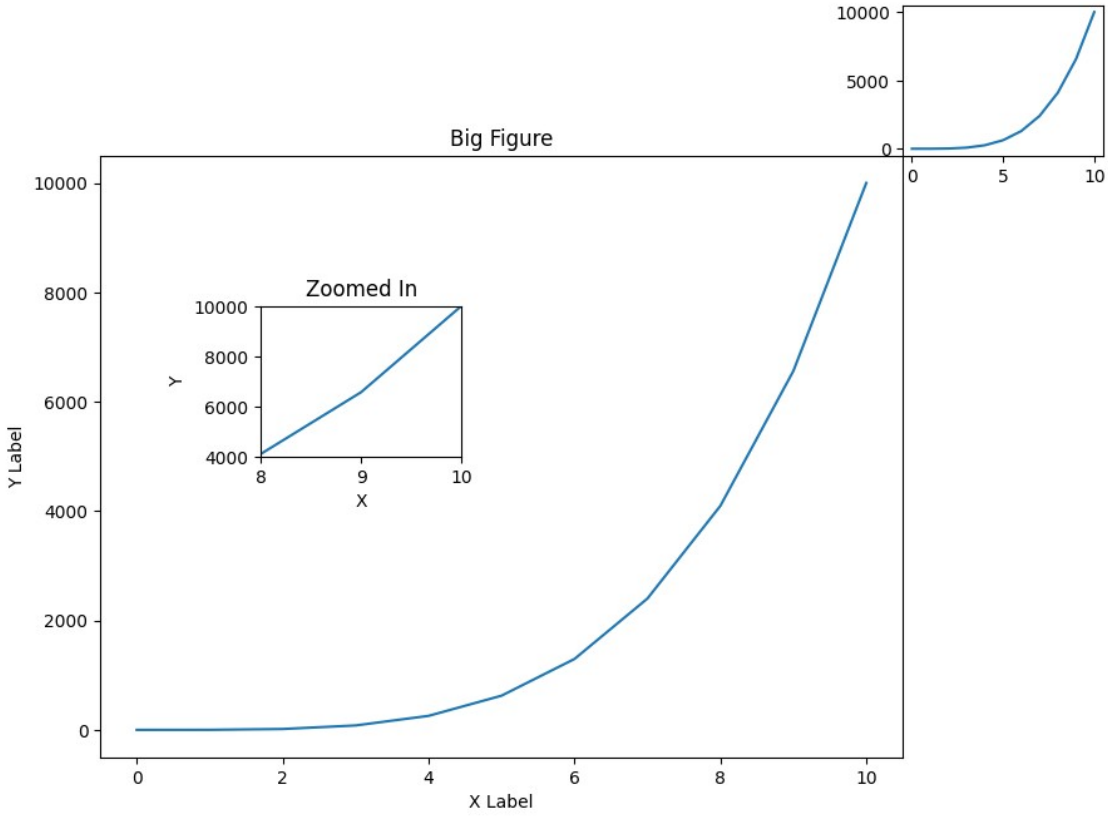
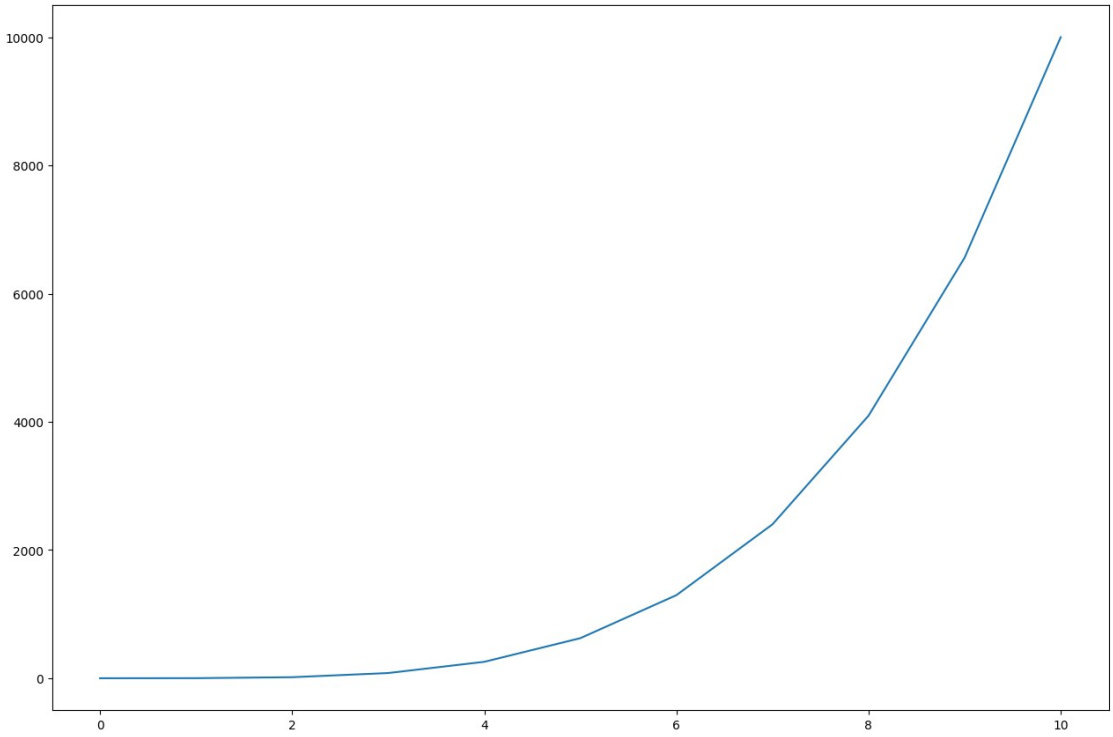


Figure Parameters

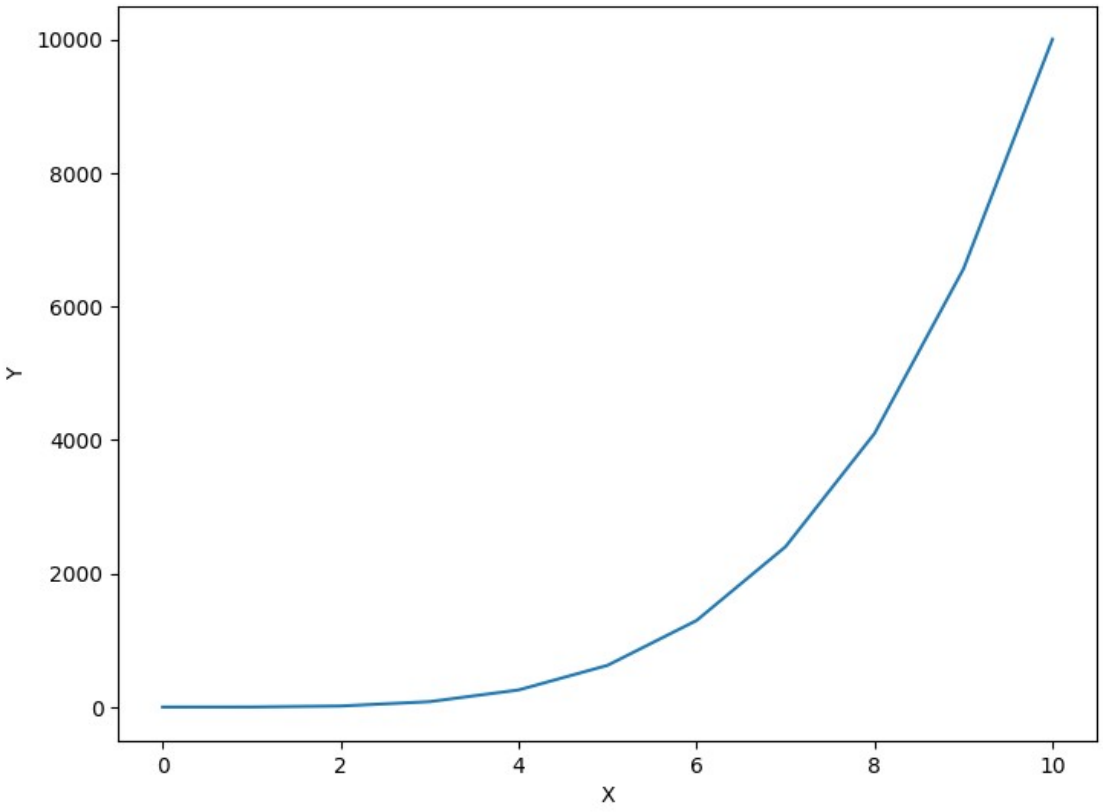
```
# Creates blank canvas
fig = plt.figure(figsize=(12,8),dpi=100)

axes1 = fig.add_axes([0, 0, 1, 1])

axes1.plot(a,b)
```

```
[<matplotlib.lines.Line2D at 0x211a31eb070>]
```

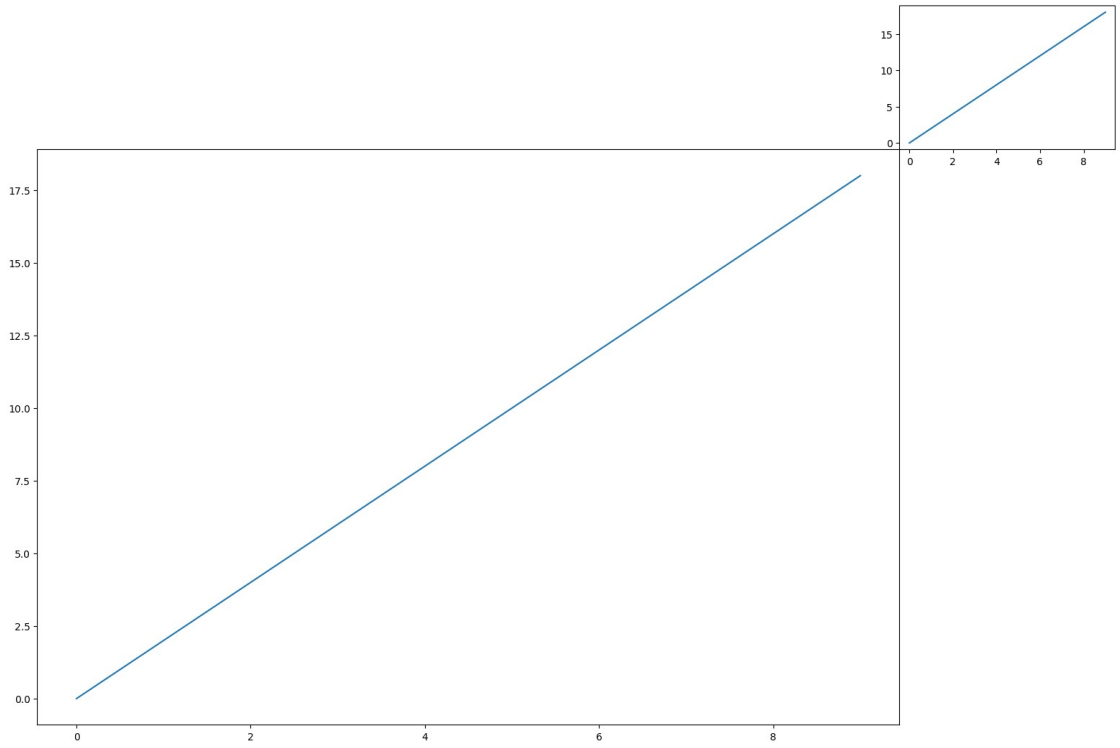


Exporting a Figure

```
fig = plt.figure()

axes1 = fig.add_axes([0, 0, 1, 1])

axes1.plot(a,b)
axes1.set_xlabel('X')
axes1.set_ylabel('Y')

# bbox_inches ='tight' automatically makes sure the bounding box is
correct, and show graph marking in image
fig.savefig('figure.png',bbox_inches='tight')
```




```
# Creates blank canvas
fig = plt.figure(figsize=(12,8))

axes1 = fig.add_axes([0, 0, 1, 1]) # Full figure
axes2 = fig.add_axes([1, 1, 0.25, 0.25]) # Starts at top right corner!

# Larger Figure Axes 1
axes1.plot(x,y)

# Insert Figure Axes 2
axes2.plot(x,y)

fig.savefig('test.png',bbox_inches='tight')
```



Matplotlib Sub Plots

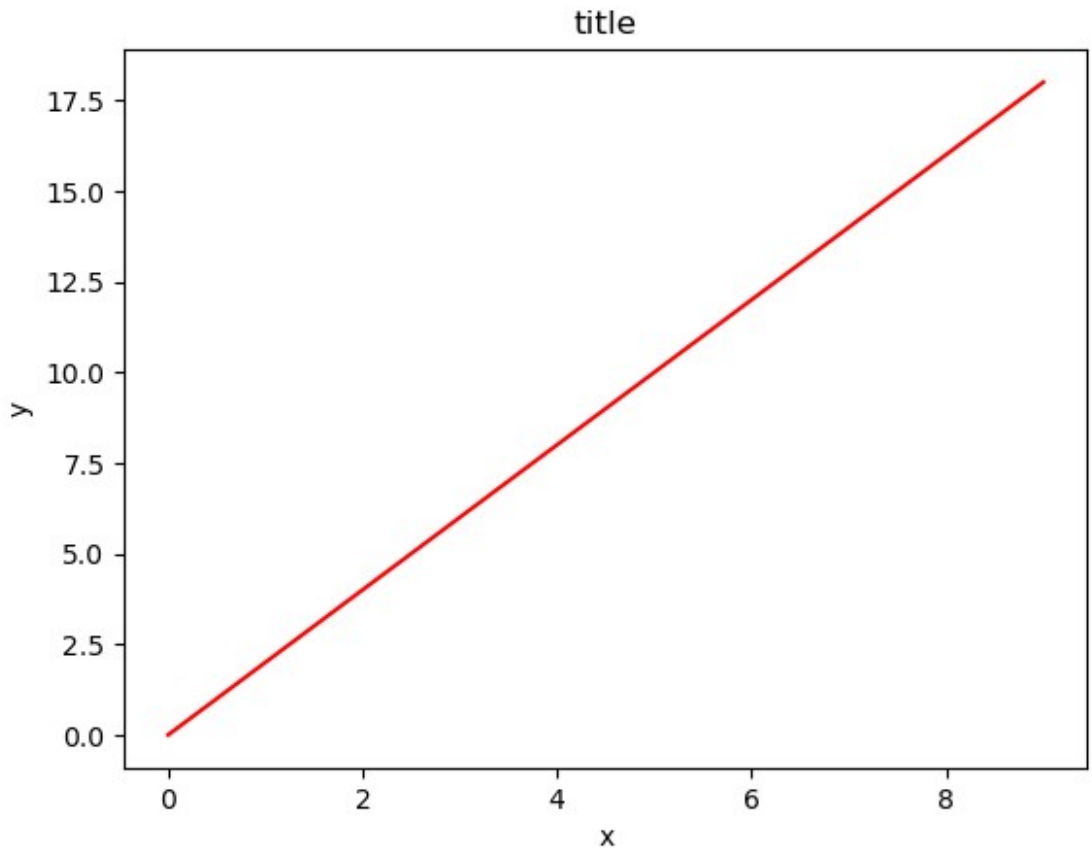
plt.subplots()

NOTE: Make sure you put the commands all together in the same cell as we do in this notebook and video!

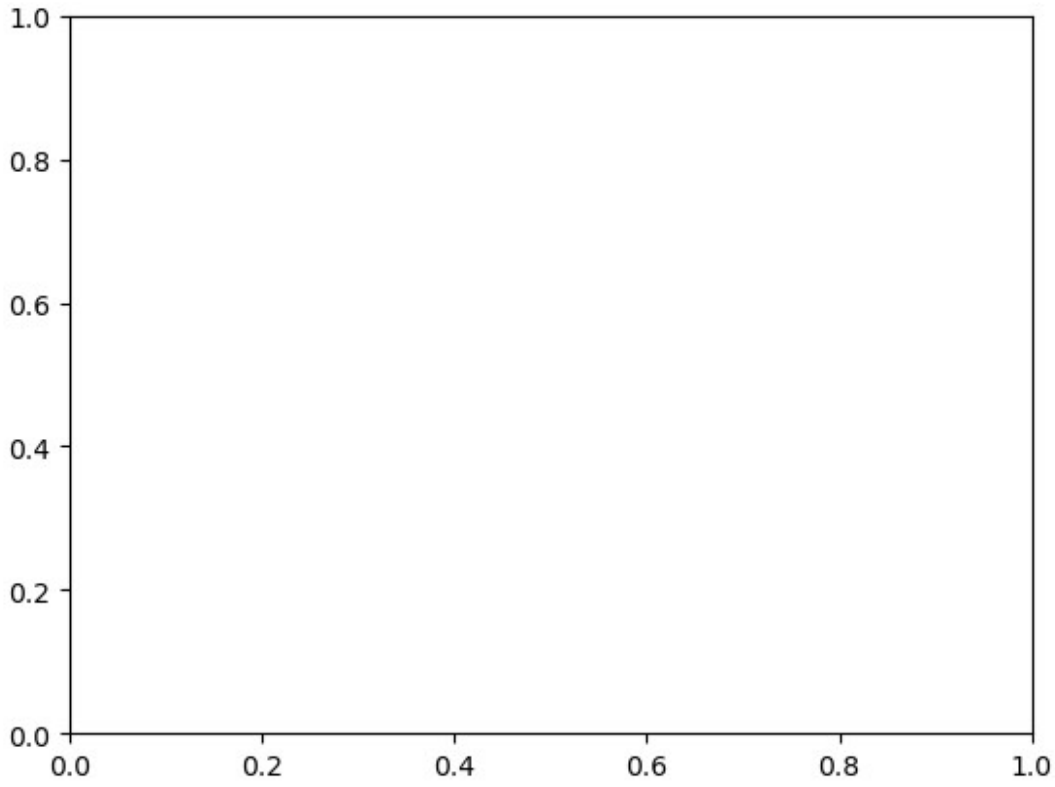
The plt.subplots() object will act as a more automatic axis manager. This makes it much easier to show multiple plots side by side.

Note how we use tuple unpacking to grba both the Figure object and a numpy array of axes:

```
 # Use similar to plt.figure() except use tuple unpacking to grab fig and axes
fig, axes = plt.subplots()

# Now use the axes object to add stuff to plot
axes.plot(x, y, 'r') # Here 'r' is for red color
axes.set_xlabel('x')
axes.set_ylabel('y')
axes.set_title('title'); #; hides Out[]
```



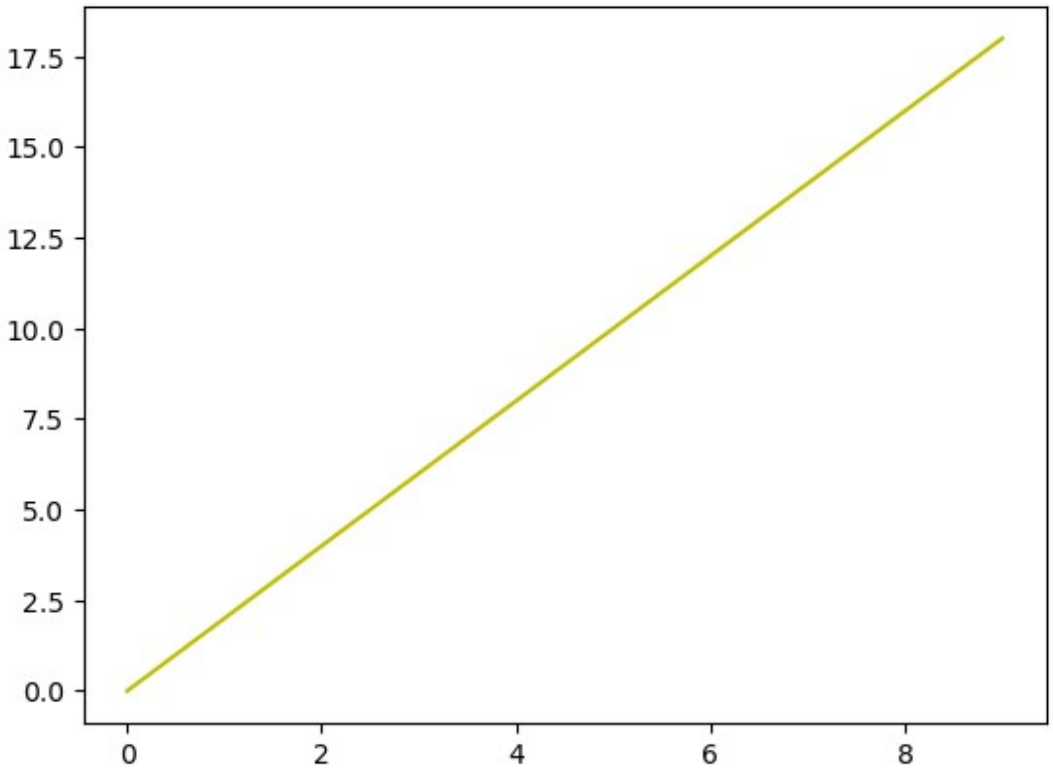

```
fix,axes = plt.subplots()
```




```
axes
```

```
<AxesSubplot:>
```

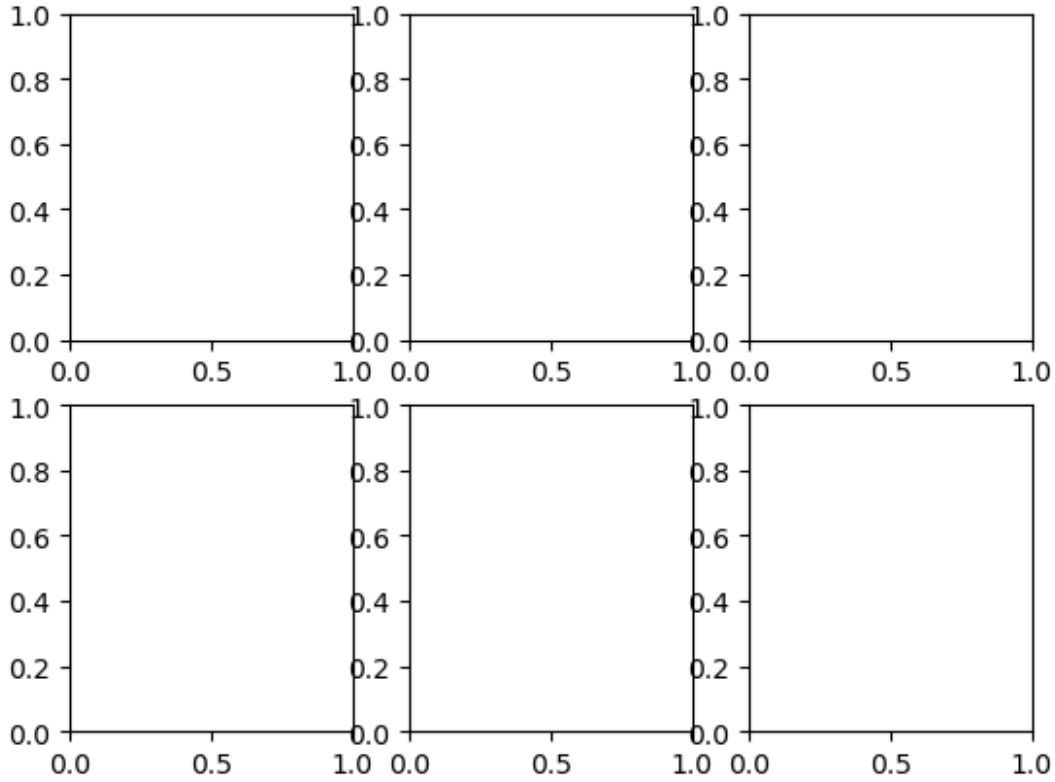
```
fix,axes = plt.subplots()
axes.plot(x,y,'y');
```



Adding rows and columns

Then you can specify the number of rows and columns when creating the subplots() object:

```
# Empty canvas of 1 by 2 subplots
fix,axes = plt.subplots(nrows=2, ncols=3) # There is 2 rows and 3
columns
```



```
# Axes is an array of axes to plot on
axes
```

```
array([[<AxesSubplot:>, <AxesSubplot:>, <AxesSubplot:>],
       [<AxesSubplot:>, <AxesSubplot:>, <AxesSubplot:>]],
dtype=object)
```

```
axes.shape
```

```
(2, 3)
```

```
type(axes) # It is array
```

```
numpy.ndarray
```

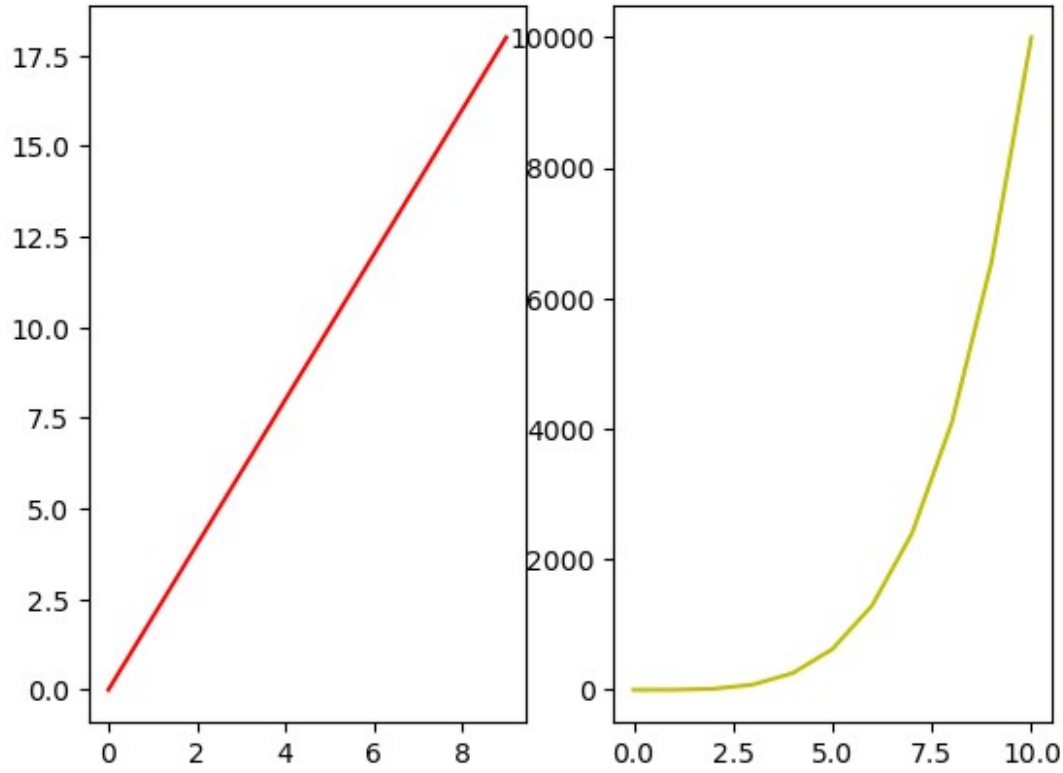
Plotting on axes objects

Just as before, we simple .plot() on the axes objects, and we can also use the .set_ methods on each axes.

Let's explore this, make sure this is all in the same cell:

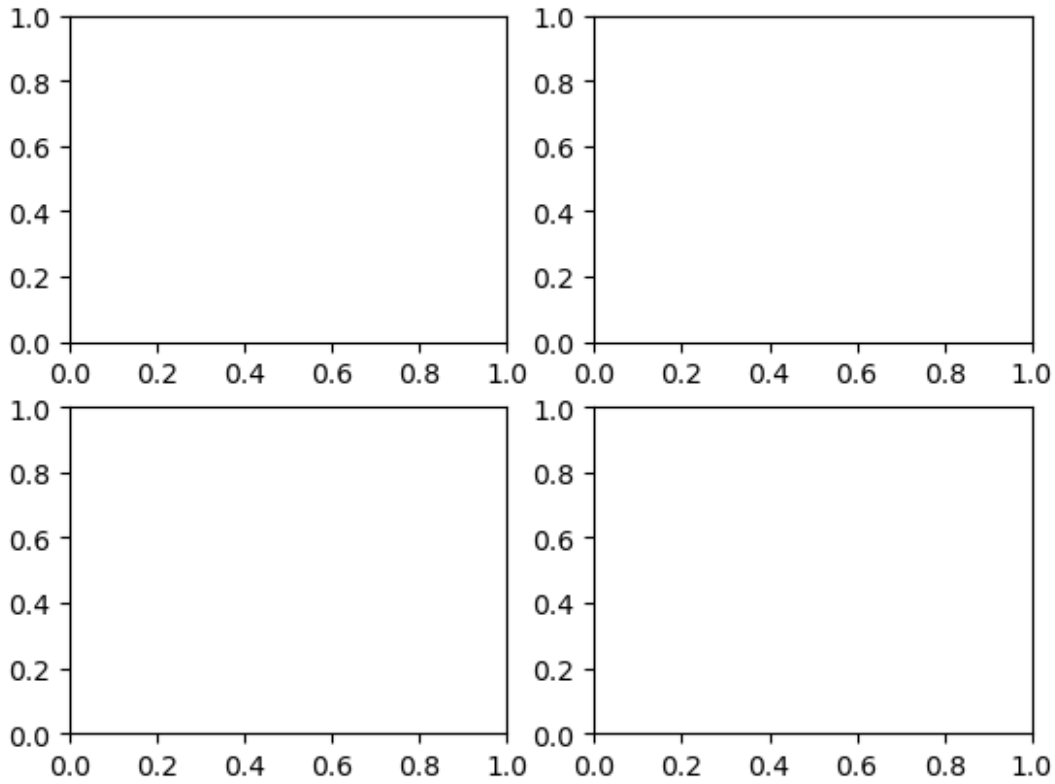
```
fig,axes = plt.subplots(nrows=1,ncols=2)

axes[0].plot(x,y,'r')
axes[1].plot(a,b,'y')
```

```
[<matplotlib.lines.Line2D at 0x1913dc64130>]
```

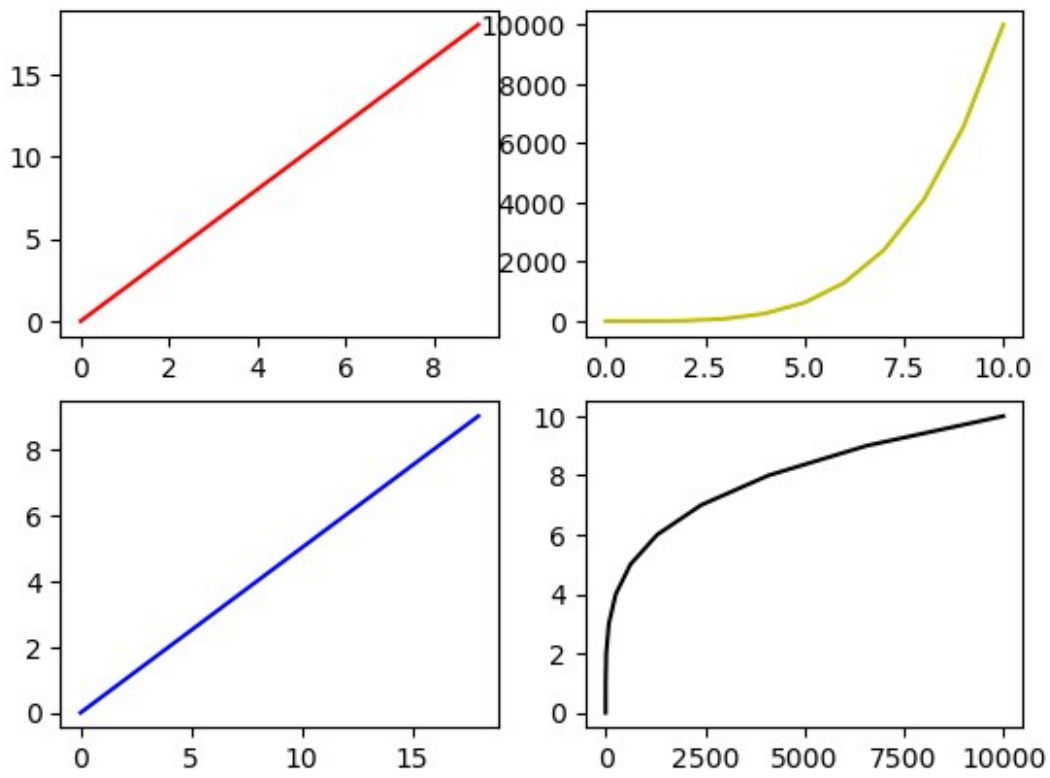



```
# NOTE! This returns 2 dimensional array
fix,axes=plt.subplots(nrows=2,ncols=2)

axes[0].plot(x,y,'r')
axes[1].plot(x,y,'g')
axes[2].plot(x,y,'y')
axes[3].plot(x,y,'b')
# Here because now it become 2 dimensional
```

```
---------------------------------------------------------------------------
-----
AttributeError                            Traceback (most recent call
last)
~\AppData\Local\Temp\ipykernel_12840\1695712468.py in <module>
      2 fix,axes=plt.subplots(nrows=2,ncols=2)
      3
----> 4 axes[0].plot(x,y,'r')
      5 axes[1].plot(x,y,'g')
      6 axes[2].plot(x,y,'y')

AttributeError: 'numpy.ndarray' object has no attribute 'plot'
```



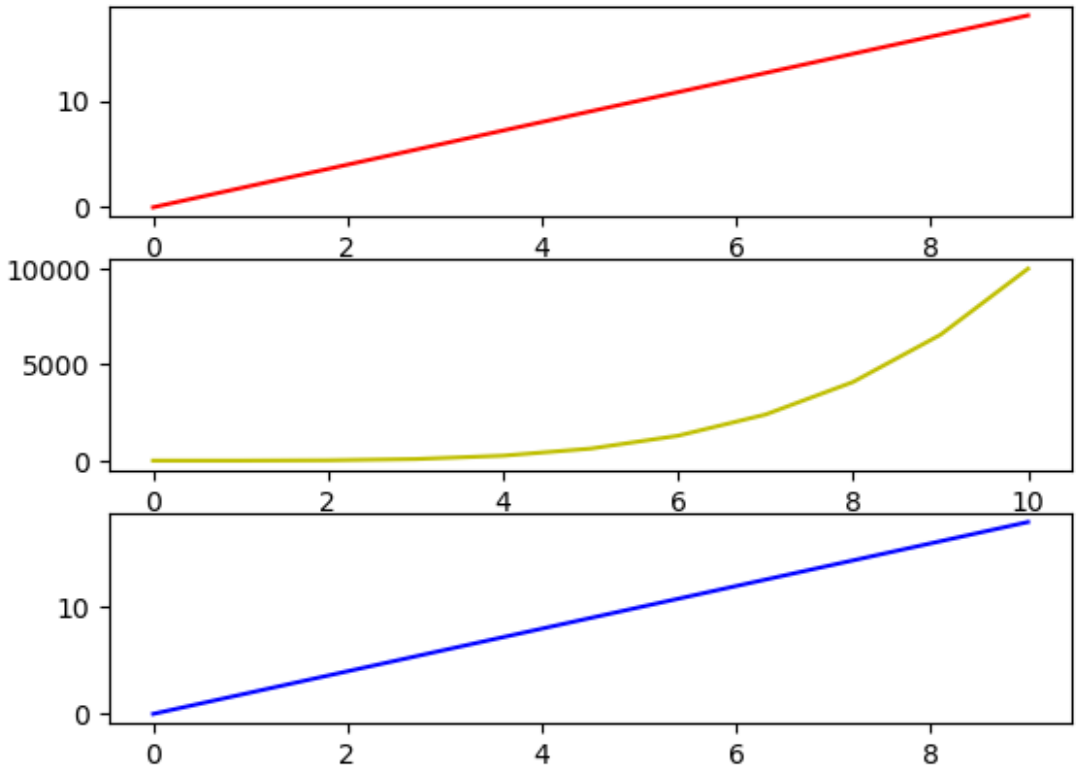

```
# That we put plot address in axes
fig,axes=plt.subplots(nrows=2,ncols=2)

axes[0][0].plot(x,y,'r');
axes[0][1].plot(a,b,'y');
axes[1][0].plot(y,x,'b');
axes[1][1].plot(b,a,'black');
```



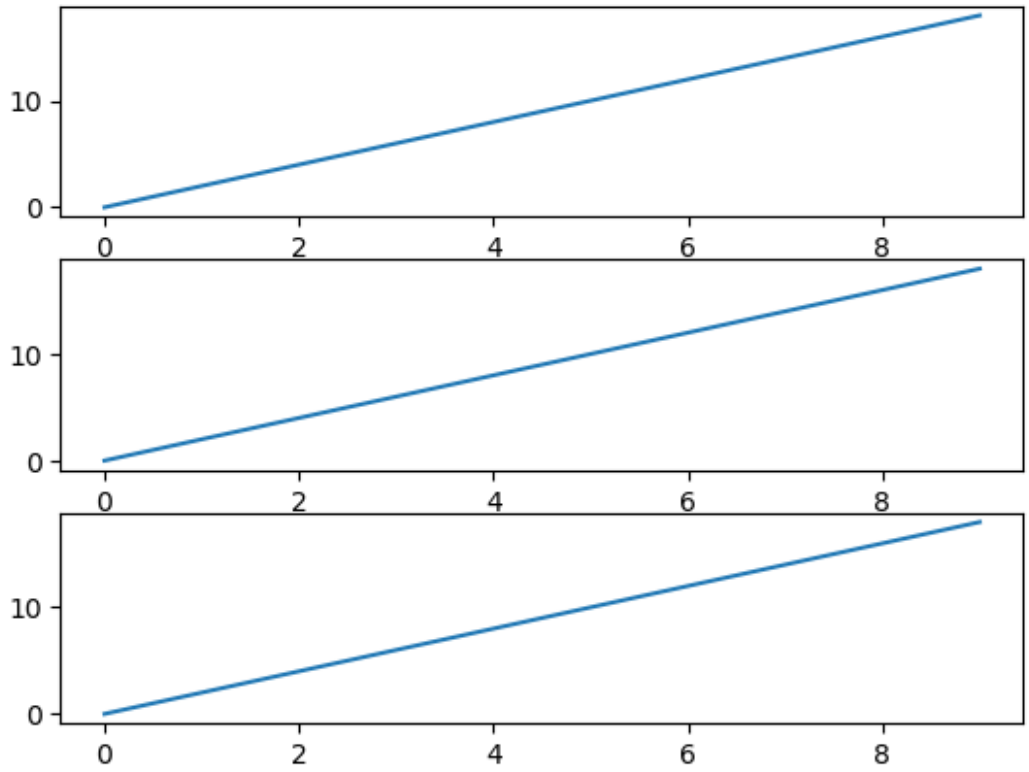
A common issue with matplolib is overlapping subplots or figures. We ca use **fig.tight_layout()** or **plt.tight_layout()** method, which automatically adjusts the positions of the axes on the figure canvas so that there is no overlapping content:

```
fig,axes= plt.subplots(nrows=3,ncols=1)

# Here it one dimensional
axes[0].plot(x,y,'r')
axes[1].plot(a,b,'y')
axes[2].plot(x,y,'b')
```

```
[<matplotlib.lines.Line2D at 0x1913e25cb80>]
```




```
# If want to put same data in every plot we can use for loop

fig,axes=plt.subplots(nrows=3,ncols=1)

for ax in axes:
    ax.plot(x,y)
```

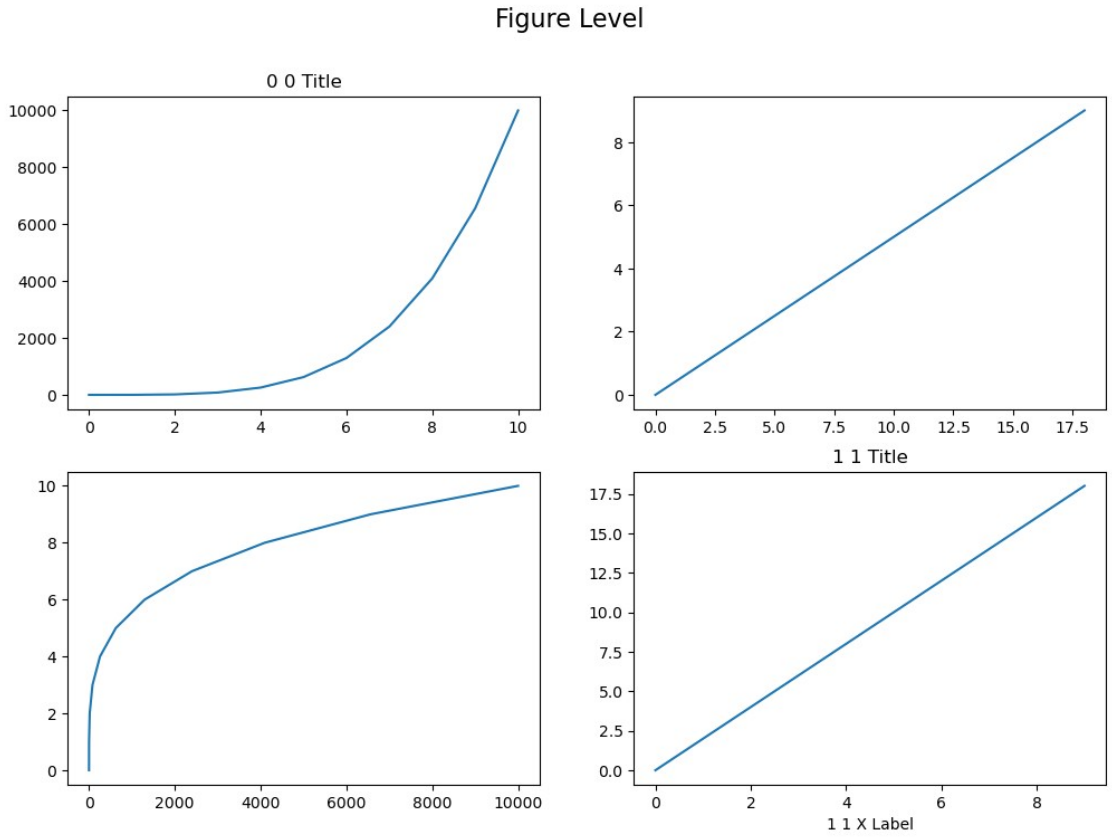


Parameters on subplots()

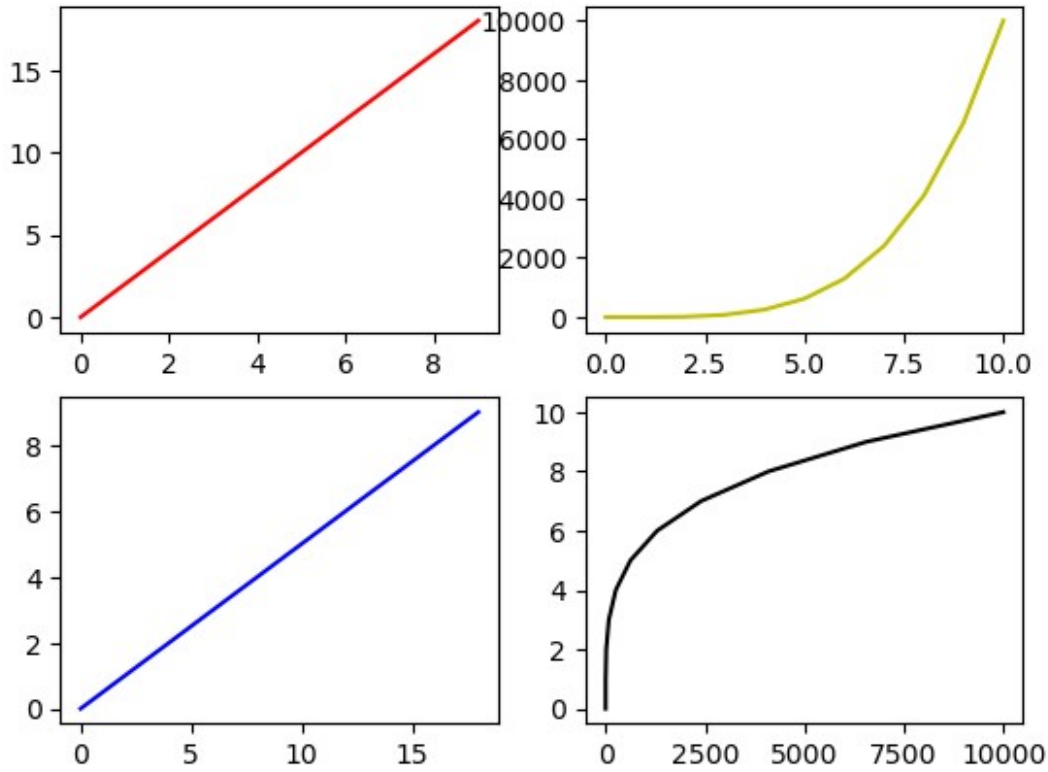
Recall we have both the Figure object and the axes. Meaning we can edit properties at both levels.

```
fig,axes = plt.subplots(nrows=2,ncols=2,figsize=(12,8))

# SET YOUR AXES PARAMETERS FIRST

# Parameters at the axes level
axes[0][0].plot(a,b)
axes[0][0].set_title('0 0 Title')


axes[1][1].plot(x,y)
axes[1][1].set_title('1 1 Title')
axes[1][1].set_xlabel('1 1 X Label')

axes[0][1].plot(y,x)
axes[1][0].plot(b,a)

# THEN SET OVERALL FIGURE PARAMETERS

# Parameters at the Figure level
fig.suptitle("Figure Level",fontsize=16)
```

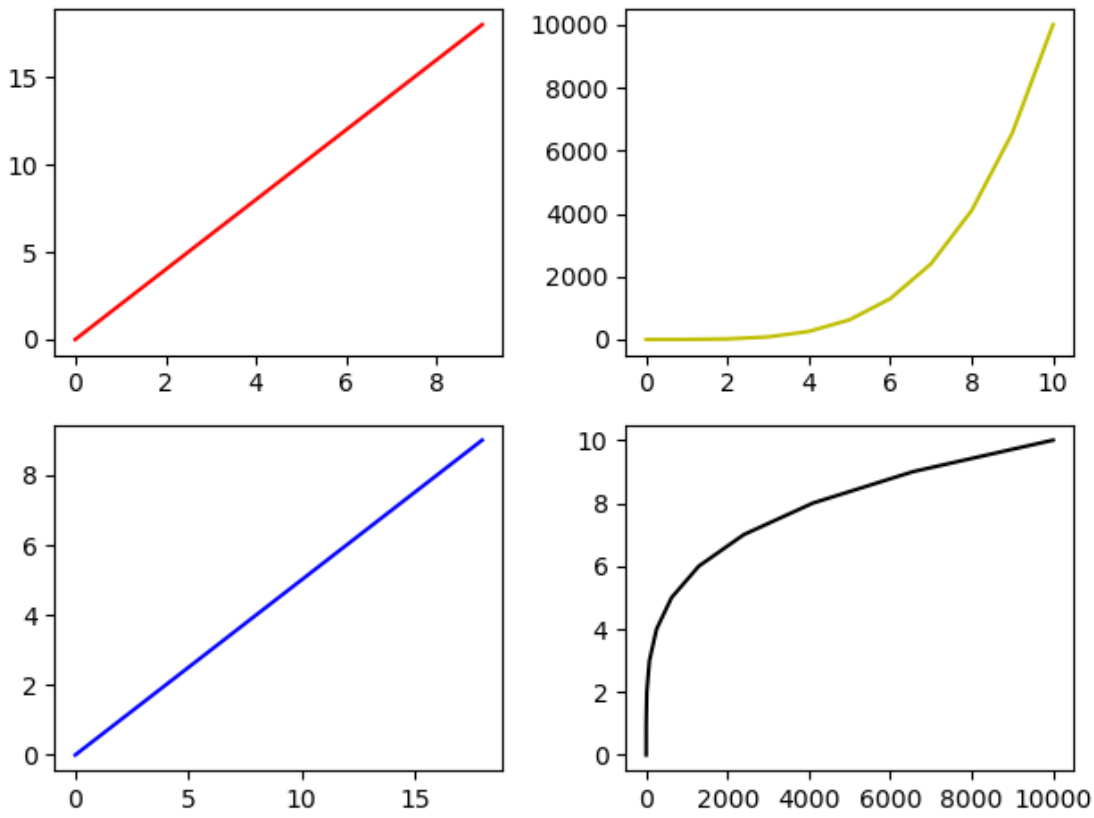
```
plt.show()
```



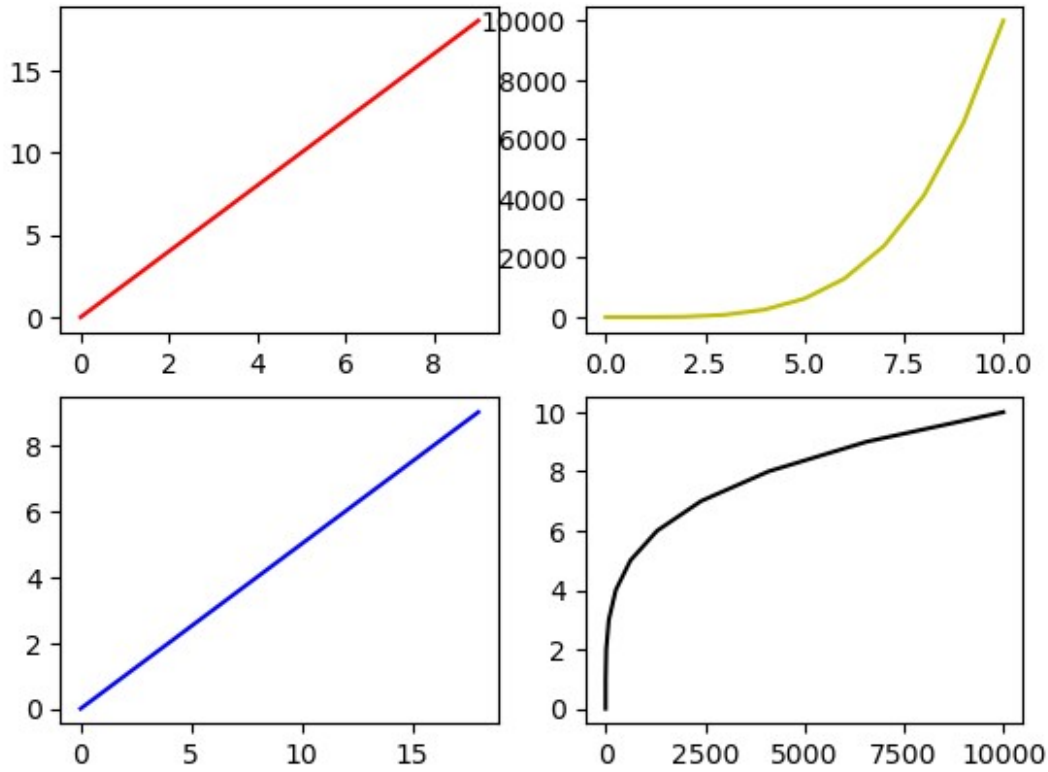

```
# That we put plot address in axes
fig,axes=plt.subplots(nrows=2,ncols=2)

axes[0][0].plot(x,y,'r');
axes[0][1].plot(a,b,'y');
axes[1][0].plot(y,x,'b');
axes[1][1].plot(b,a,'black');
```



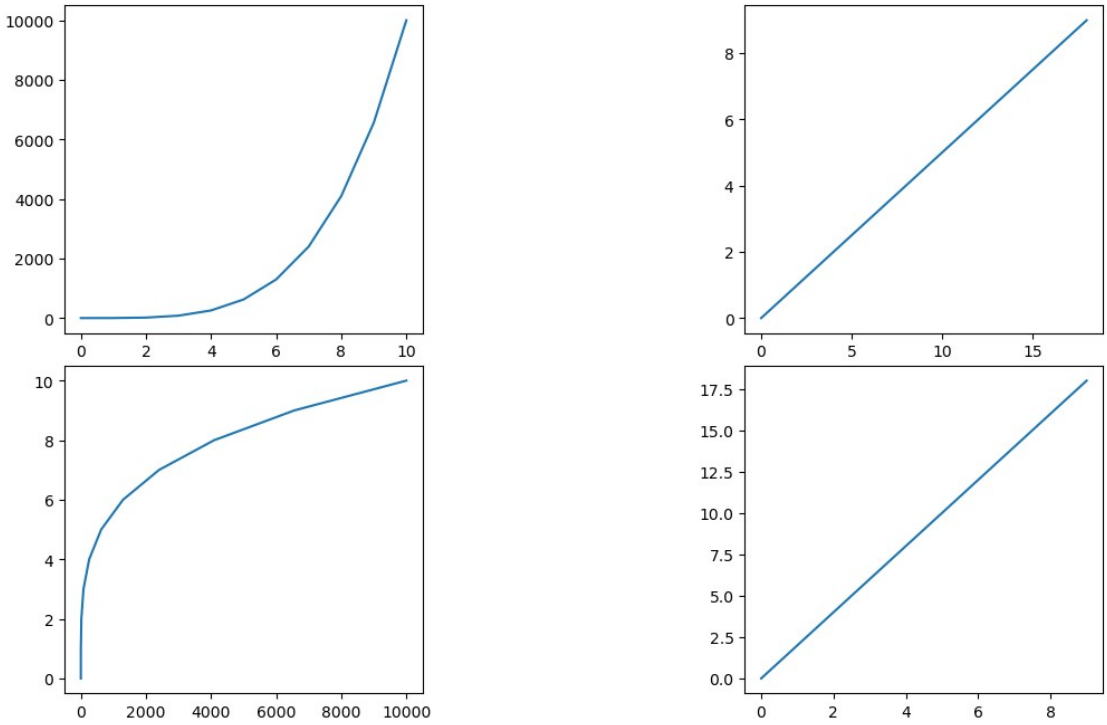

```
# its too close so we are going to use plt.tight_layout()
fig,axes=plt.subplots(nrows=2,ncols=2)

axes[0][0].plot(x,y,'r');
axes[0][1].plot(a,b,'y');
axes[1][0].plot(y,x,'b');
axes[1][1].plot(b,a,'black');

plt.tight_layout()
```




```
# But still we want more spacing
# That we put plot address in axes
fig,axes=plt.subplots(nrows=2,ncols=2)

axes[0][0].plot(x,y,'r');
axes[0][1].plot(a,b,'y');
axes[1][0].plot(y,x,'b');
axes[1][1].plot(b,a,'black');

fig.subplots_adjust()  # Press Shift + tab to know more
```

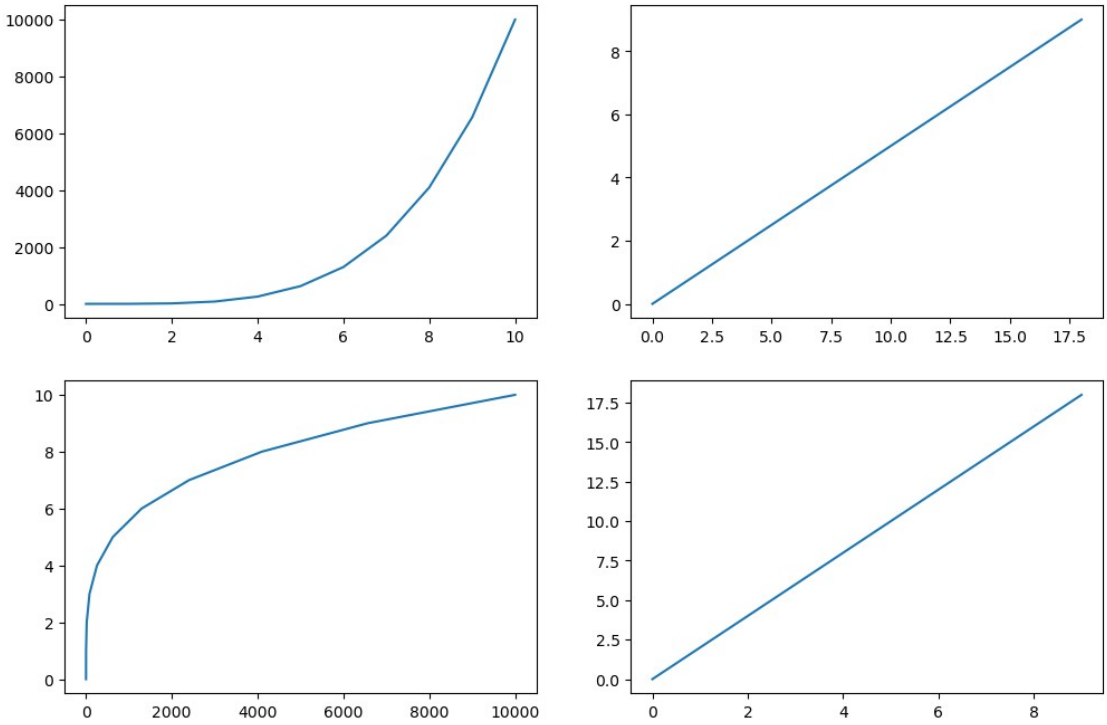



```
fig,axes = plt.subplots(nrows=2,ncols=2,figsize=(12,8))

# Parameters at the axes level
axes[0][0].plot(a,b)
axes[1][1].plot(x,y)
axes[0][1].plot(y,x)
axes[1][0].plot(b,a)

# Use left,right,top, bottom to stretch subplots
# Use wspace,hspace to add spacing between subplots
fig.subplots_adjust(left=None,
    bottom=None,
    right=None,
    top=None,
    wspace=0.9,
    hspace=0.1,)

plt.show()
```



Exporting plt.subplots()

```
# NOTE! This returns 2 dimensional array
fig,axes = plt.subplots(nrows=2,ncols=2,figsize=(12,8))

axes[0][0].plot(a,b)
axes[1][1].plot(x,y)
axes[0][1].plot(y,x)
axes[1][0].plot(b,a)

fig.savefig('subplots.png',bbox_inches='tight')

plt.show()
```



Matplotlib Styling

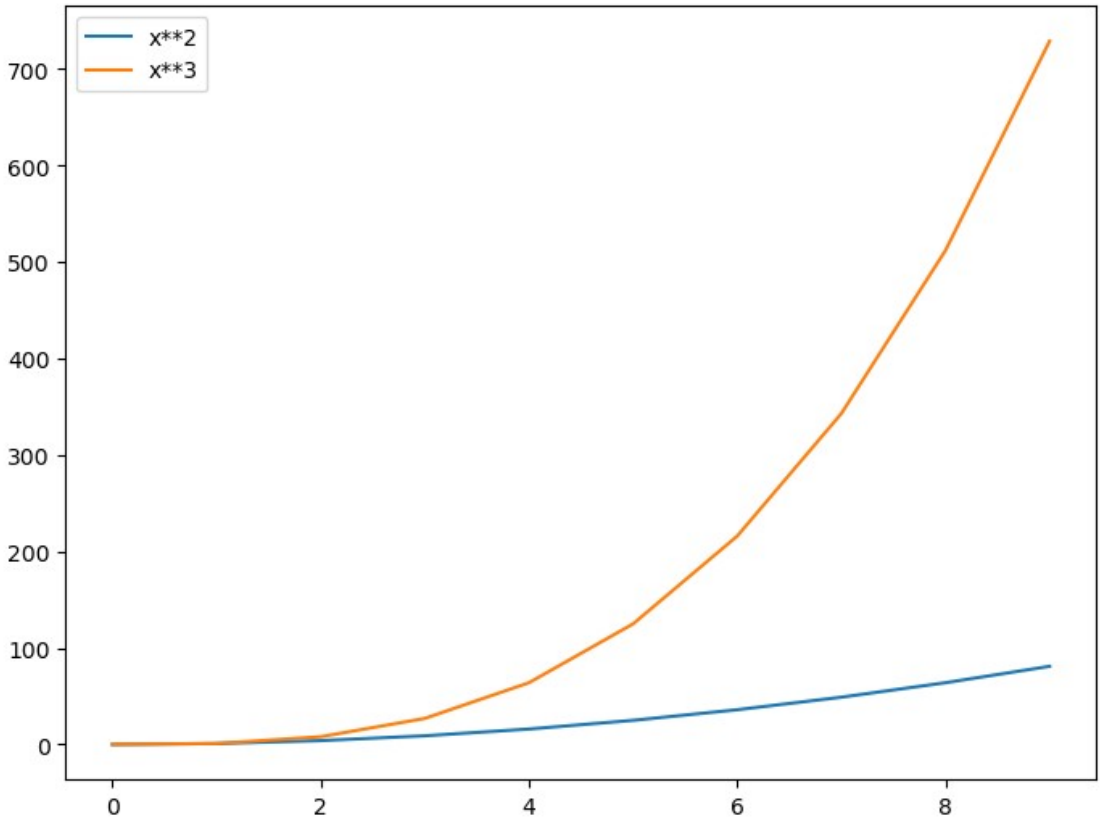
Import the `matplotlib.pyplot` module under the name `plt` (the tidy way):

Legends

You can use the **label="label text"** keyword argument when plots or other objects are added to the figure, and then using the **legend** method without arguments to add the legend to the figure:

```
fig = plt.figure()

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

ax.plot(x, x**2, label="x**2")
ax.plot(x, x**3, label="x**3")
ax.legend()
```

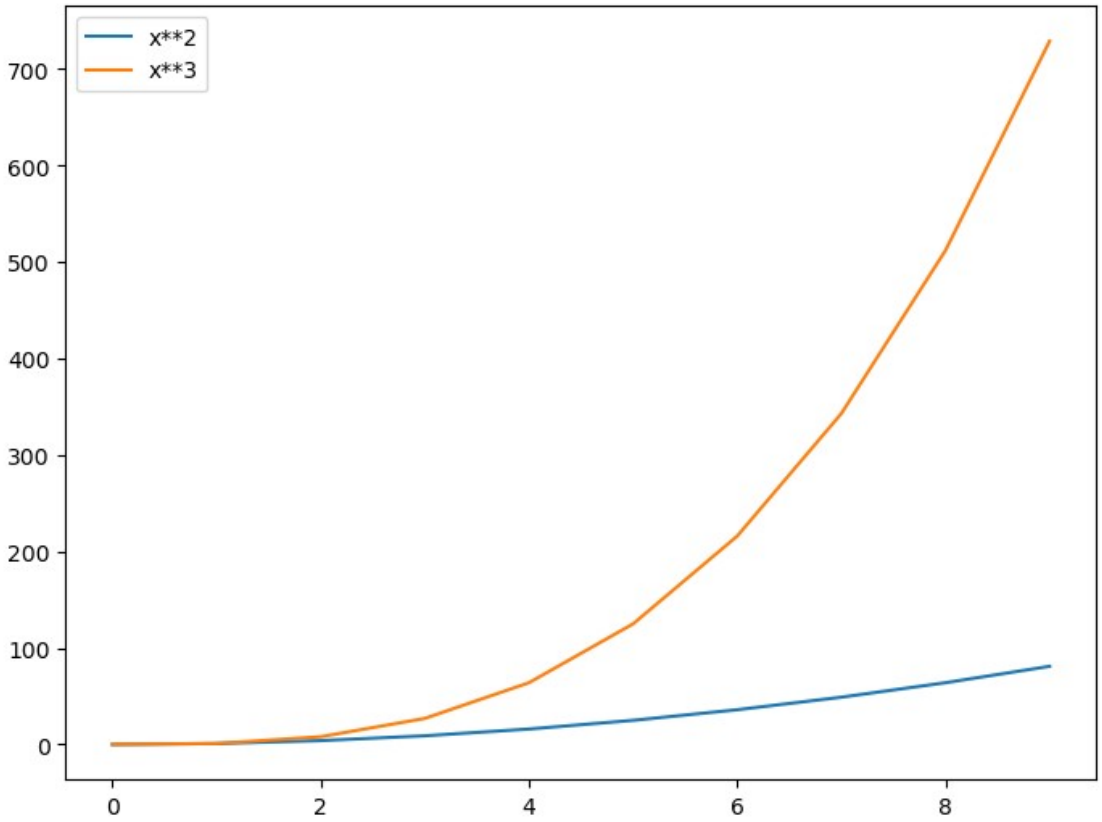
```
<matplotlib.legend.Legend at 0x1913f6d42e0>
```



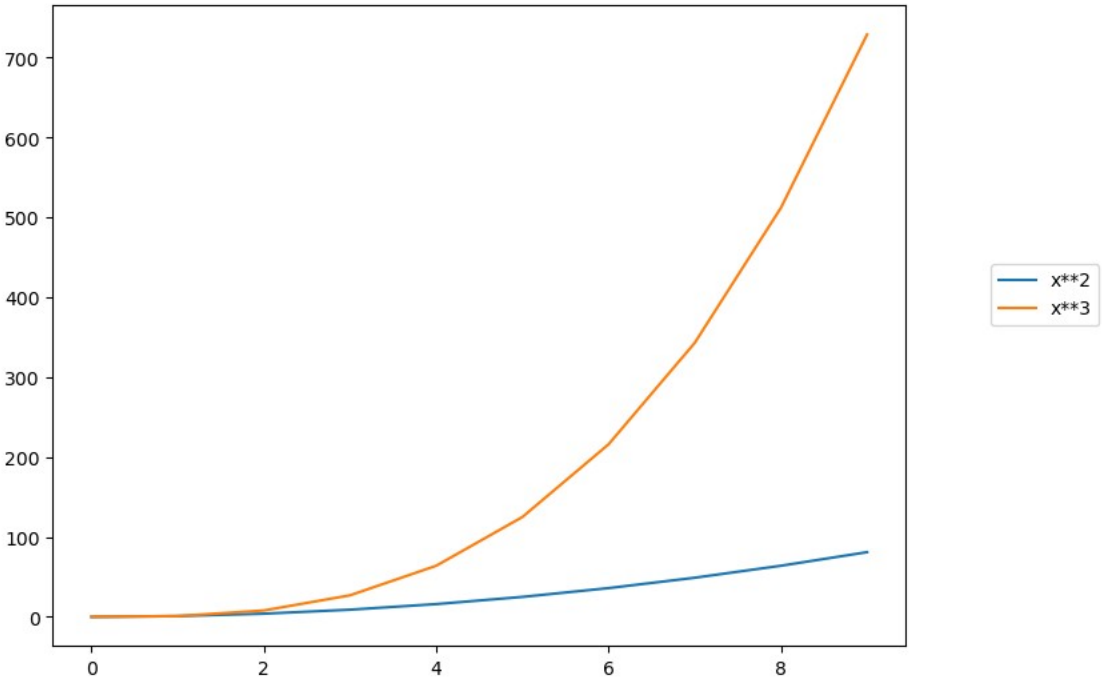
Notice how legend could potentially overlap some of the actual plot!

The **legend** function takes an optional keyword argument **loc** that can be used to specify where in the figure the legend is to be drawn. The allowed values of **loc** are numerical codes for the various places the legend can be drawn. See the documentation page for details. Some of the most common **loc** values are:

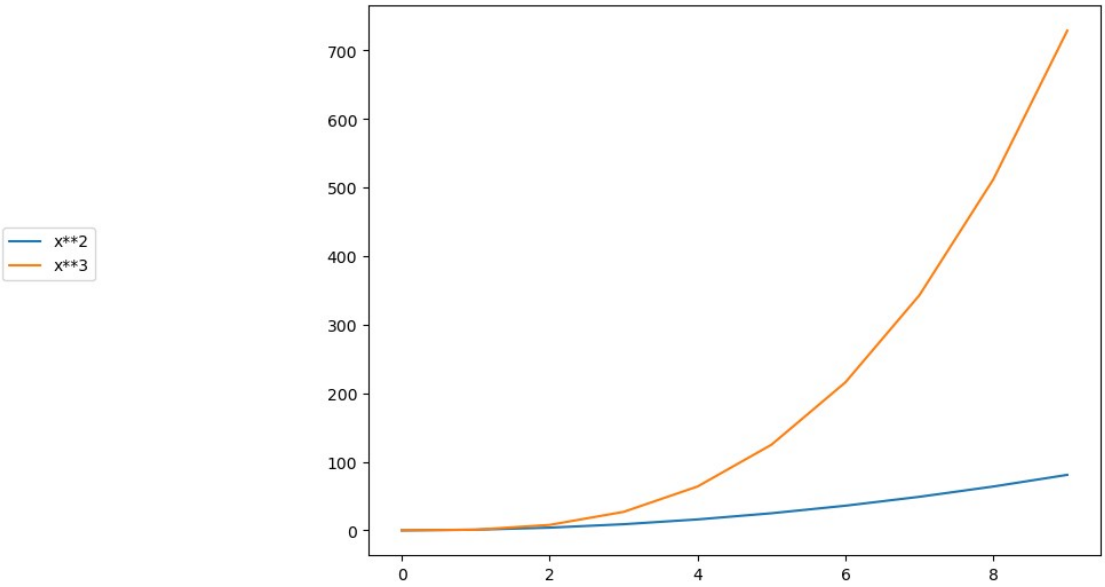
```
# Lots of options....

ax.legend(loc=1) # upper right corner
ax.legend(loc=2) # upper left corner
ax.legend(loc=3) # lower left corner
ax.legend(loc=4) # lower right corner

# .. many more options are available

# Most common to choose
ax.legend(loc=0) # let matplotlib decide the optimal location and we
can write (loc='upper left') and so on
fig
```




```
ax.legend(loc=(1.1,0.5)) # manually set location, more than 1 will
outside that plot
fig
```



```
ax.legend(loc=(-.5,.5)) # we can use -ve too for moving left and
bottom of plot
fig
```



Setting colors, linewidths, linetypes

Matplotlib gives you *a lot* of options for customizing colors, linewidths, and linetypes.

There is the basic MATLAB like syntax (which I would suggest you avoid using unless you already feel really comfortable with MATLAB). Instead let's focus on the keyword parameters.

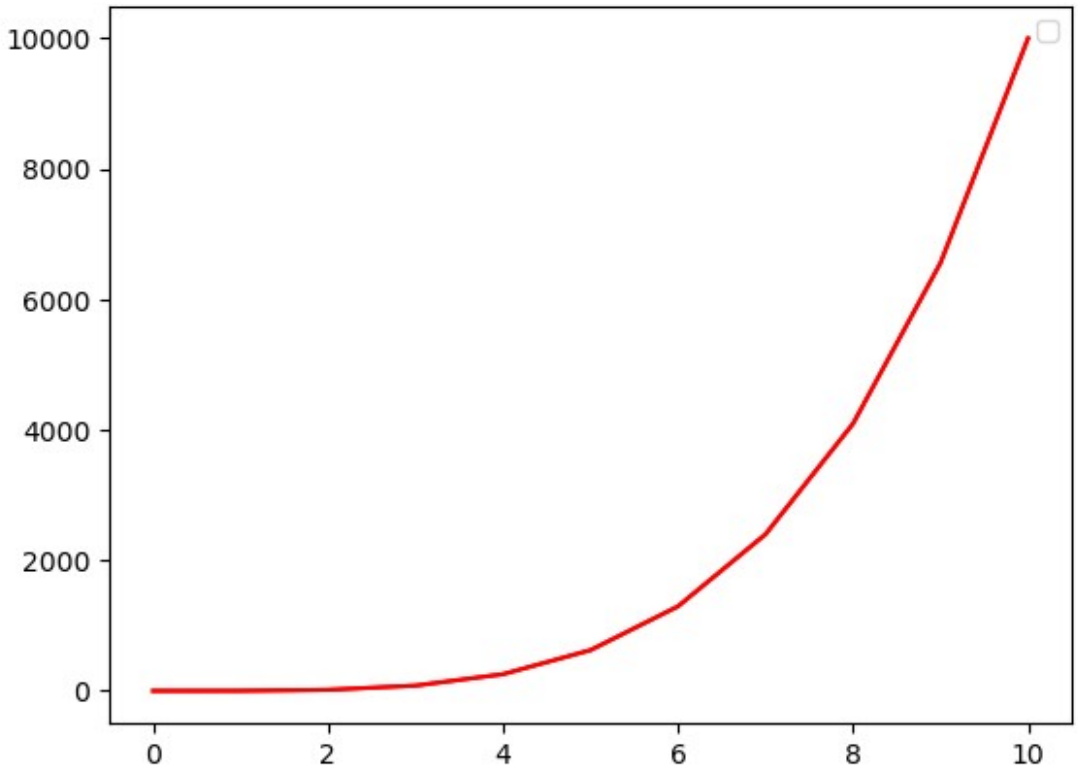
Quick View:

Colors with MatLab like syntax

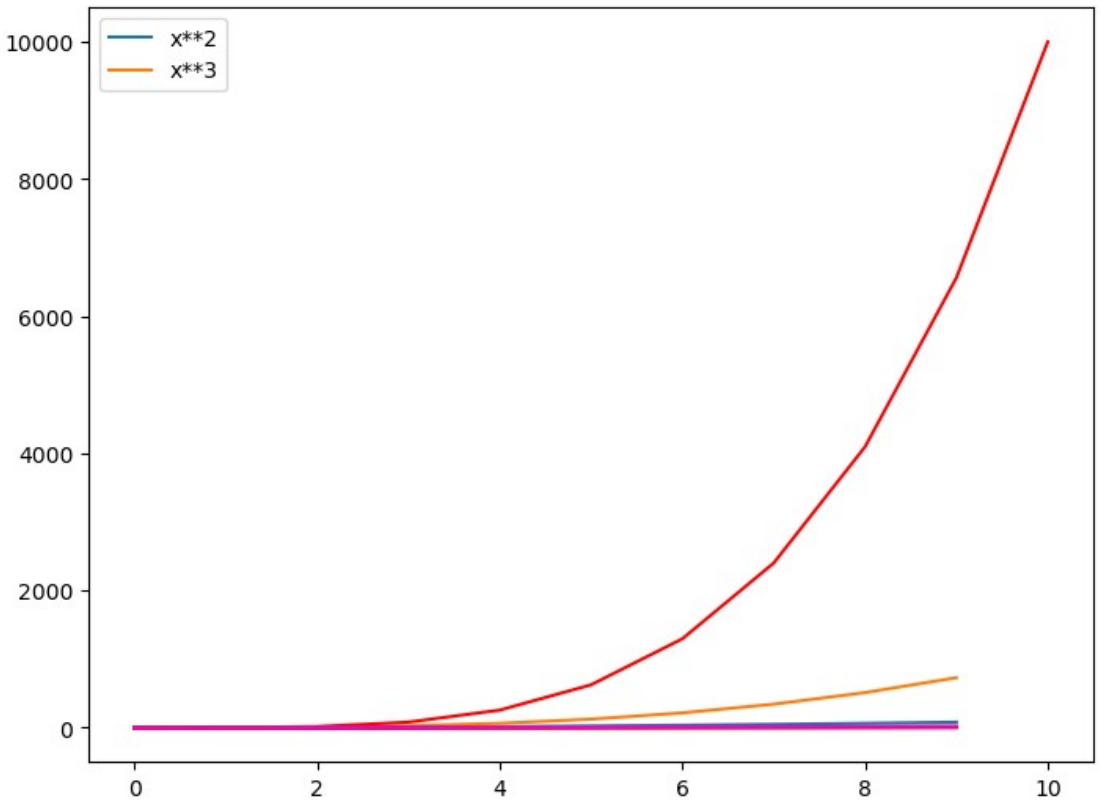
With matplotlib, we can define the colors of lines and other graphical elements in a number of ways. First of all, we can use the MATLAB-like syntax where `'b'` means blue, `'g'` means green, etc. The MATLAB API for selecting line styles are also supported: where, for example, 'b.-' means a blue line with dots:

```
ax.plot(a,b,color='red') # here we can write 'r' for short red and we
can give hex code we can google 'hex color picker'
ax.legend(loc=0)
fig
```

```
No artists with labels found to put in legend.  Note that artists
whose label start with an underscore are ignored when legend() is
called with no argument.
```




```
ax.plot(x,y,color='#de0bd0') # Here Hex code
fig
```



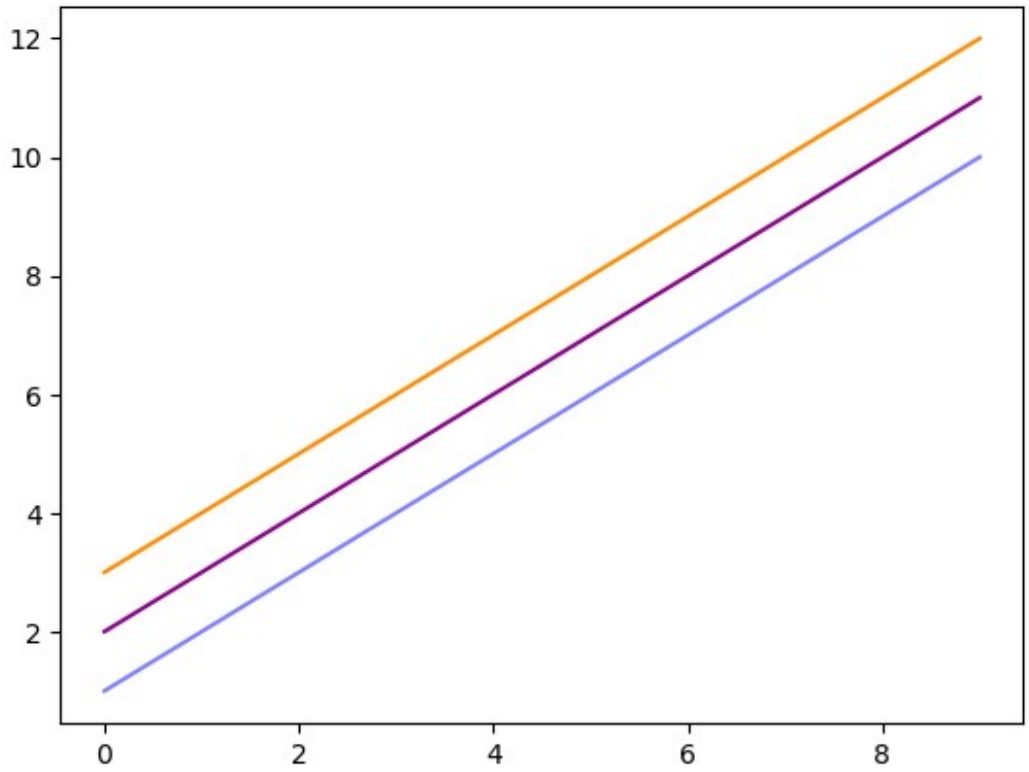
Suggested Approach: Use keyword arguments

Colors with the color parameter

We can also define colors by their names or RGB hex codes and optionally provide an alpha value using the `color` and `alpha` keyword arguments. Alpha indicates opacity.

```
fig, ax = plt.subplots()

ax.plot(x, x+1, color="blue", alpha=0.5) # half-transparant
ax.plot(x, x+2, color="#8B008B")        # RGB hex code
ax.plot(x, x+3, color="#FF8C00")        # RGB hex code
```

```
[<matplotlib.lines.Line2D at 0x1913deb6c10>]
```



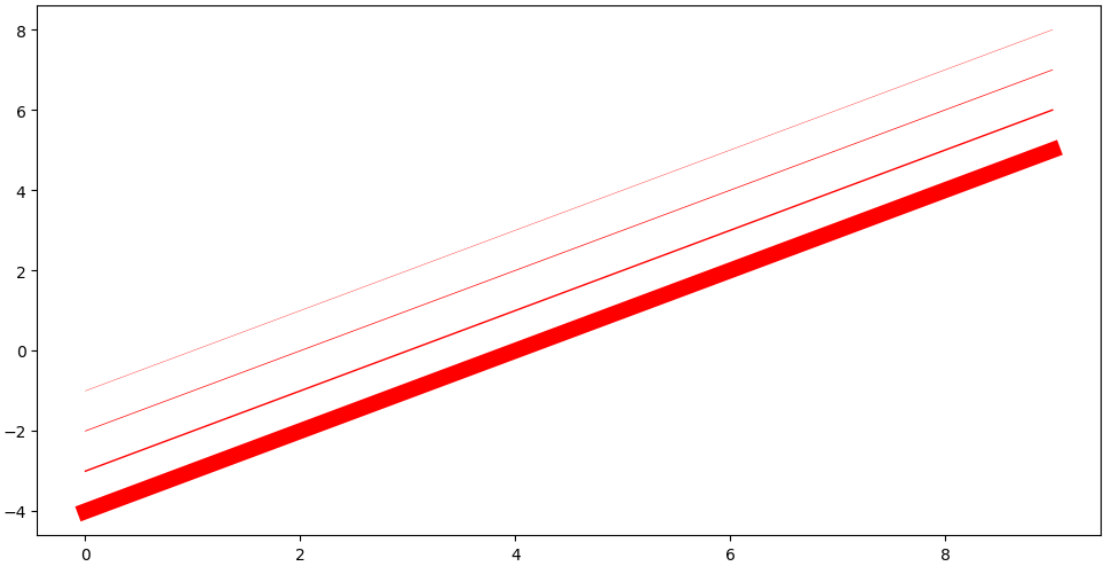
Line and marker styles

Linewidth

To change the line width, we can use the `linewidth` or `lw` keyword argument.

```
fig, ax = plt.subplots(figsize=(12,6))

# Use linewidth or lw
ax.plot(x, x-1, color="red", linewidth=0.25)
ax.plot(x, x-2, color="red", lw=0.50)
ax.plot(x, x-3, color="red", lw=1)
ax.plot(x, x-4, color="red", lw=10)
```

```
[<matplotlib.lines.Line2D at 0x1913e2be130>]
```

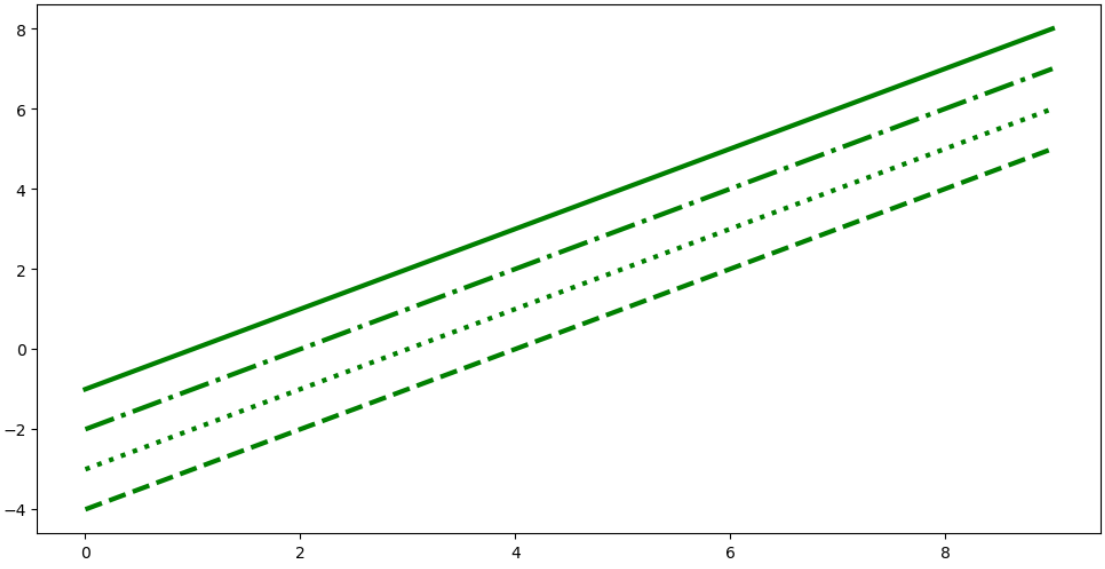


Linestyles

There are many linestyles to choose from, here is the selection:

```
# possible linestype options ‘--‘, ‘_’, ‘-.’, ‘:’, ‘steps’
fig, ax = plt.subplots(figsize=(12,6))

ax.plot(x, x-1, color="green", lw=3, linestyle='-') # solid
ax.plot(x, x-2, color="green", lw=3, ls='-.') # dash and dot
ax.plot(x, x-3, color="green", lw=3, ls=':') # dots
ax.plot(x, x-4, color="green", lw=3, ls='--') # dashes
```

```
[<matplotlib.lines.Line2D at 0x1913e1ab7c0>]
```



Custom linestyle dash

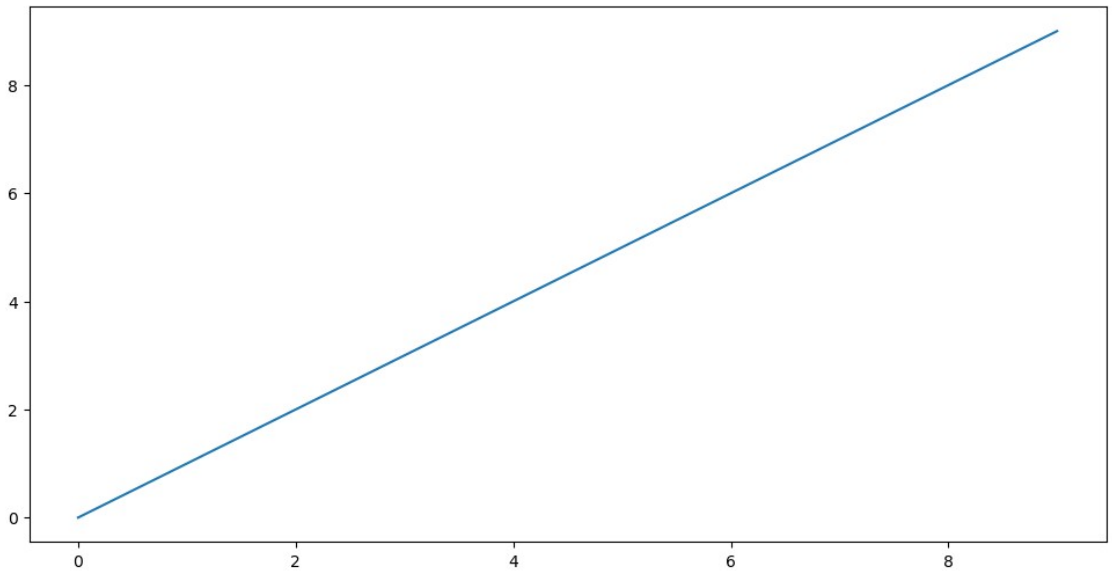
The dash sequence is a sequence of floats of even length describing the length of dashes and spaces in points.

For example, (5, 2, 1, 2) describes a sequence of 5 point and 1 point dashes separated by 2 point spaces.

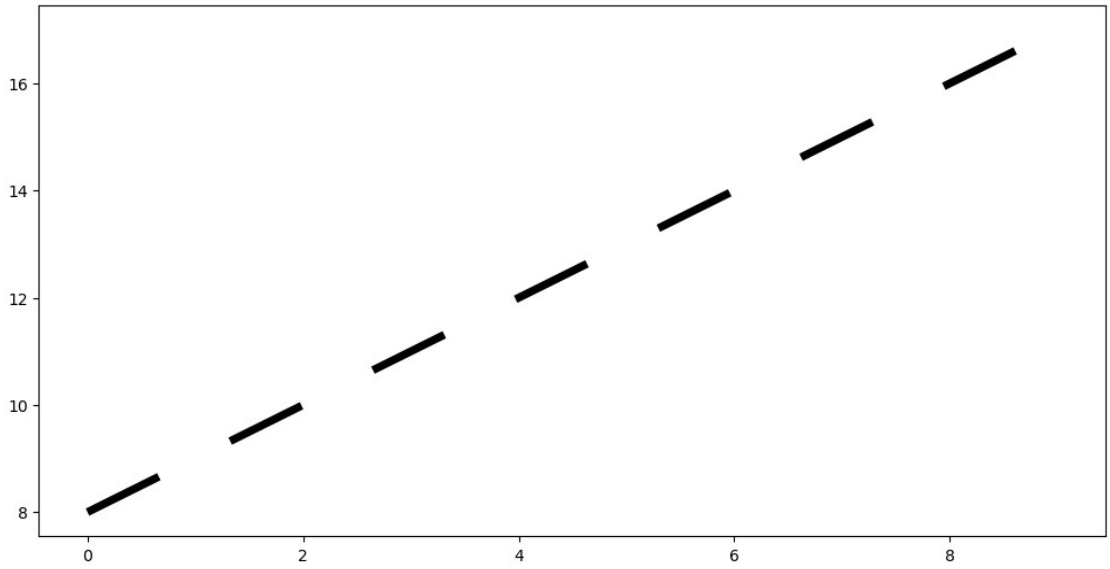
First, we see we can actually "grab" the line from the .plot() command

```
fig, ax = plt.subplots(figsize=(12,6))

lines = ax.plot(x,x)

print(type(lines))
```

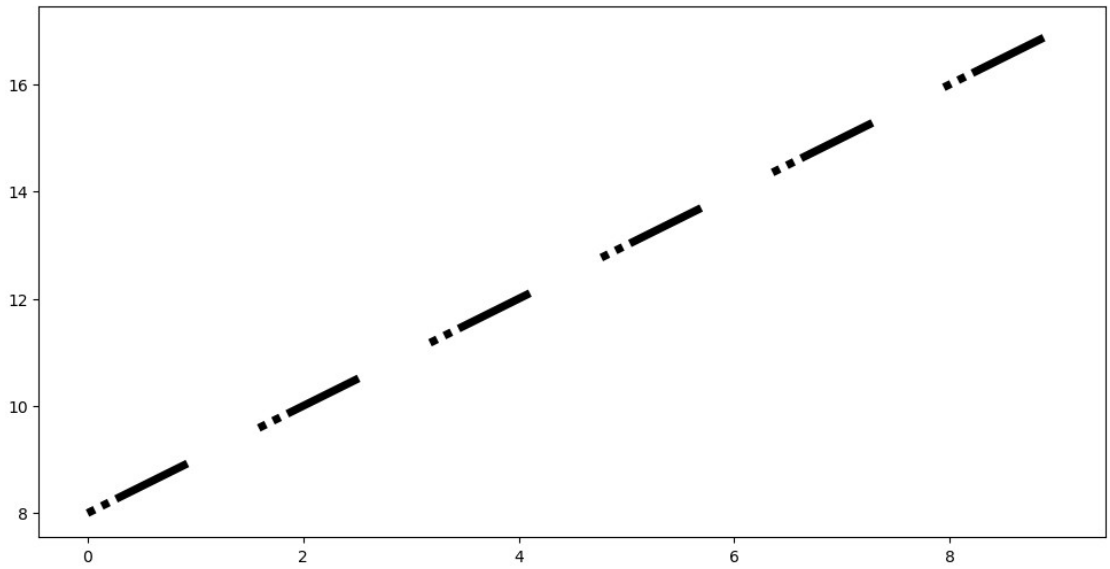
```
<class 'list'>
```




```
fig, ax = plt.subplots(figsize=(12,6))
# custom dash
lines = ax.plot(x, x+8, color="black", lw=5)
lines[0].set_dashes([10, 10]) # format: 10 line length, 10 space
length
```




```
fig, ax = plt.subplots(figsize=(12,6))
# custom dash
lines = ax.plot(x, x+8, color="black", lw=5)
lines[0].set_dashes([1,1,1,1,10,10]) # format: line length, space
length
```



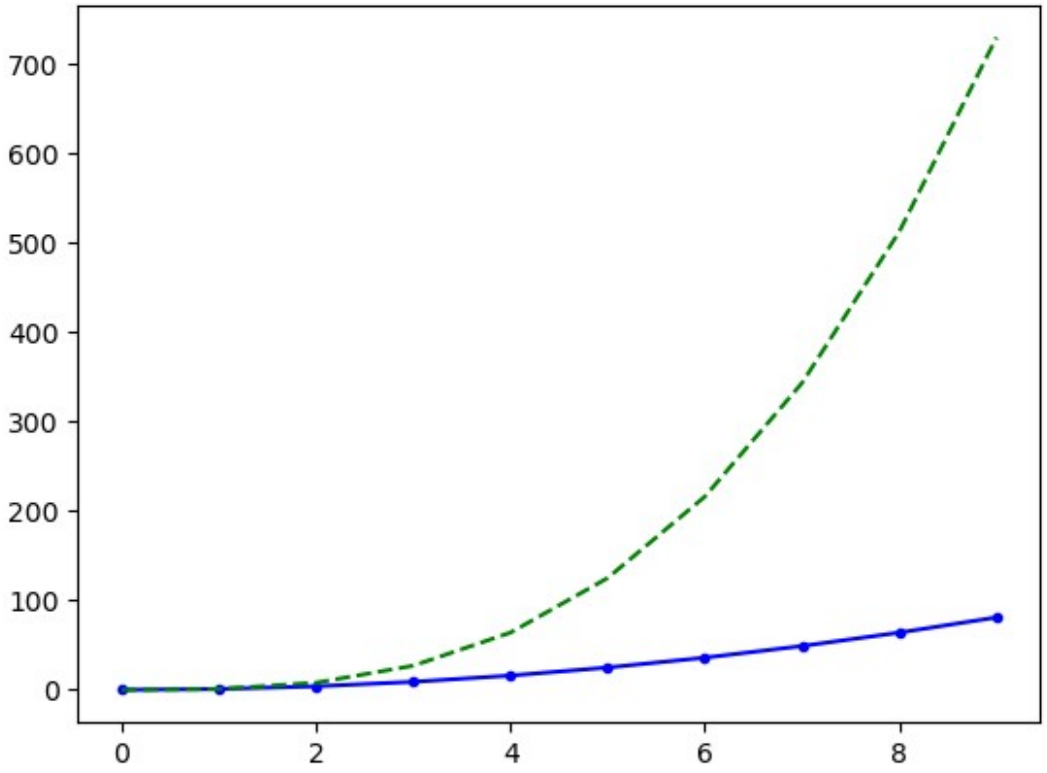
Colors with MatLab like syntax

With matplotlib, we can define the colors of lines and other graphical elements in a number of ways. First of all, we can use the MATLAB-like syntax where `'b'` means blue, `'g'` means green, etc. The MATLAB API for selecting line styles are also supported: where, for example, 'b.-' means a blue line with dots:

```
# MATLAB style line color and style
fig, ax = plt.subplots()
```

```
ax.plot(x, x**2, 'b.-') # blue line with dots
ax.plot(x, x**3, 'g--') # green dashed line
```

```
[<matplotlib.lines.Line2D at 0x1913e1a66a0>]
```



Markers

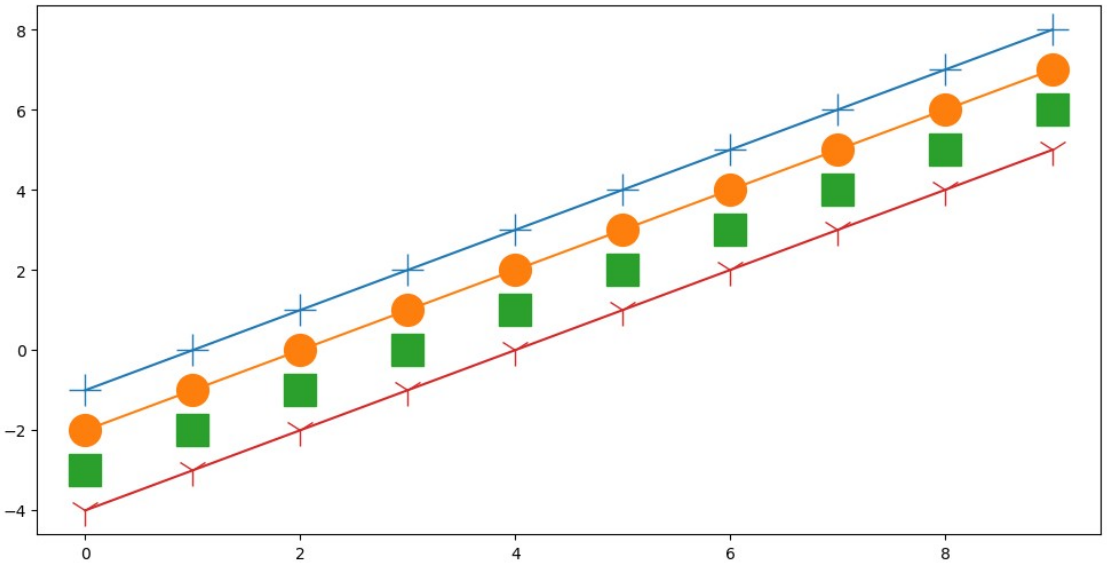
We've technically always been plotting points, and matplotlib has been automatically drawing a line between these points for us. Let's explore how to place markers at each of these points.

Markers Style

Huge list of marker types can be found here:
https://matplotlib.org/3.2.2/api/markers_api.html

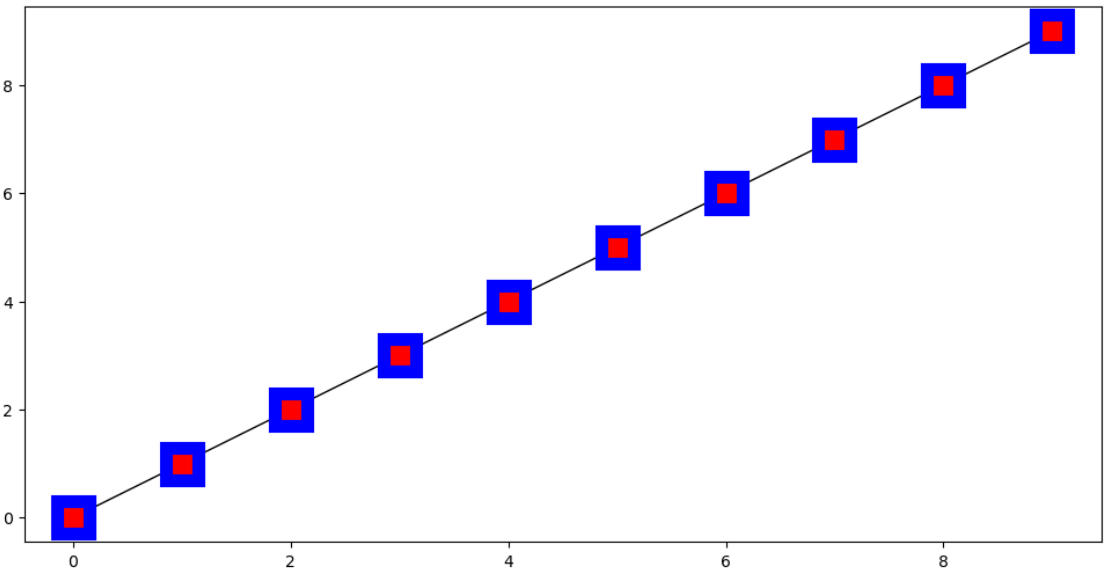
```
fig, ax = plt.subplots(figsize=(12,6))

# Use marker for string code
# Use markersize or ms for size

ax.plot(x, x-1,marker='+',markersize=20)
ax.plot(x, x-2,marker='o',ms=20) #ms can be used for markersize
ax.plot(x, x-3,marker='s',ms=20,lw=0) # make linewidth zero to see
only markers
ax.plot(x, x-4,marker='1',ms=20)
```

```
[<matplotlib.lines.Line2D at 0x1913fe1f9d0>]
```



Custom marker edges, thickness,size,and style

```
fig, ax = plt.subplots(figsize=(12,6))

# marker size and color
ax.plot(x, x, color="black", lw=1, ls='-', marker='s', markersize=20,
        markerfacecolor="red", markeredgewidth=8,
markeredgecolor="blue");
```



Final Thoughts

After these 4 notebooks on Matplotlib, you should feel comfortable creating simple quick plots, more advanced Figure plots and subplots, as well as styling them to your liking. You may have noticed we didn't cover statistical plots yet, like histograms or scatterplots, we

will use the seaborn library to create those plots instead. Matplotlib is capable of creating those plots, but seaborn is easier to use (and built on top of matplotlib!).

We have an additional notebook called "Additional-Matplotlib-Commands" which you can explore for other concepts, mainly as a quick reference. We also highly encourage you to always do a quick Google Search and StackOverflow search for matplotlib questions, as there are thousands of already answered questions there and almost any quick matplotlib question already has an answer there.

Advanced Matplotlib Commands Lecture

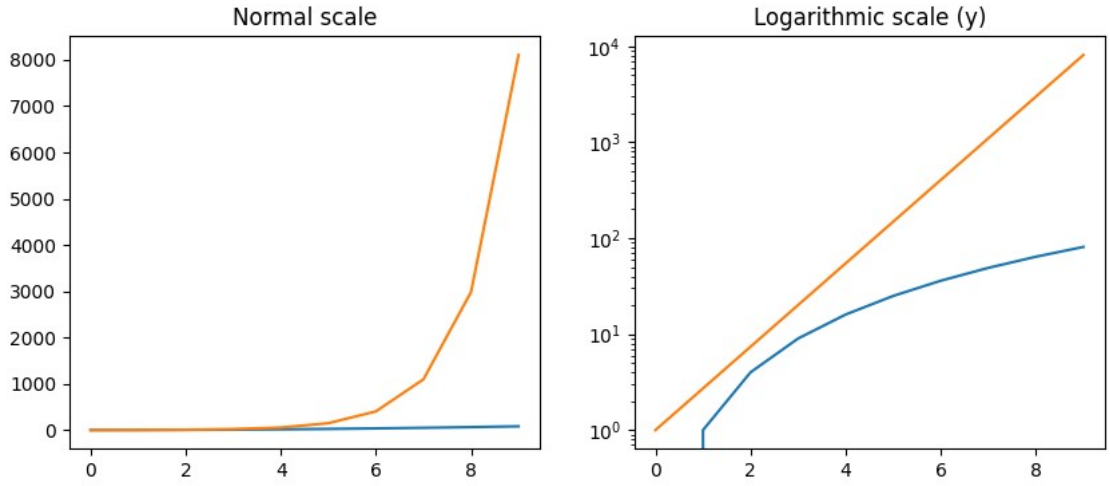
NOTE: There is no video for the notebook since its really just a reference for what method calls to look for. We also highly recommend doing a quick StackOverflow search if you're in need of a quick answer to what Matplotlib method to use for a particular case.

In this lecture we cover some more advanced topics which you won't usually use as often. You can always reference the documentation for more resources!

Logarithmic scale

It is also possible to set a logarithmic scale for one or both axes. This functionality is in fact only one application of a more general transformation system in Matplotlib. Each of the axes' scales are set seperately using `set_xscale` and `set_yscale` methods which accept one parameter (with the value "log" in this case):

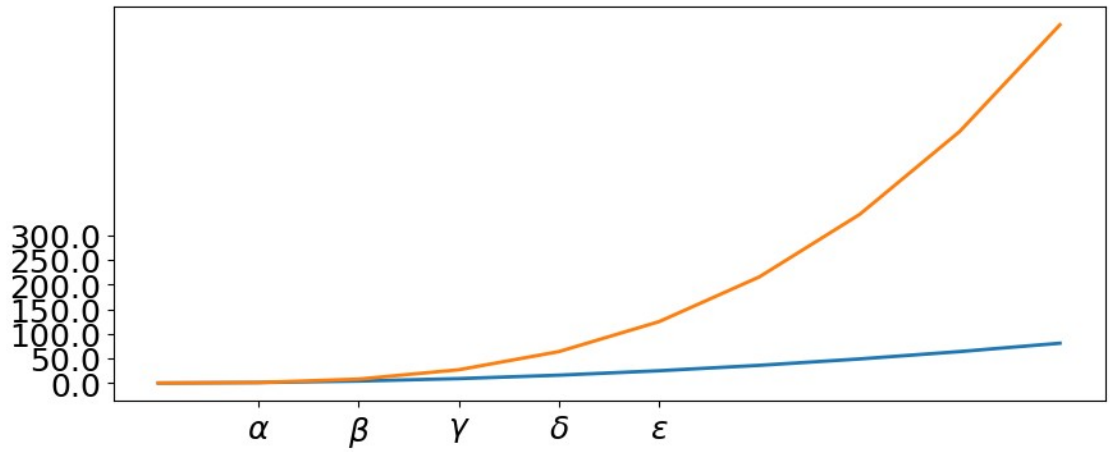
```
fig, axes = plt.subplots(1, 2, figsize=(10,4))

axes[0].plot(x, x**2, x, np.exp(x))
axes[0].set_title("Normal scale")

axes[1].plot(x, x**2, x, np.exp(x))
axes[1].set_yscale("log")
axes[1].set_title("Logarithmic scale (y)");
```



Placement of ticks and custom tick labels

We can explicitly determine where we want the axis ticks with `set_xticks` and `set_yticks`, which both take a list of values for where on the axis the ticks are to be placed. We can also use the `set_xticklabels` and `set_yticklabels` methods to provide a list of custom text labels for each tick location:

```
fig, ax = plt.subplots(figsize=(10, 4))

ax.plot(x, x**2, x, x**3, lw=2)

ax.set_xticks([1, 2, 3, 4, 5])
ax.set_xticklabels([r'$\alpha$', r'$\beta$', r'$\gamma$', r'$\delta$',
r'$\epsilon$'], fontsize=18)

yticks = [0, 50, 100, 150,200,250,300]
ax.set_yticks(yticks)
ax.set_yticklabels(["$%.1f$" % y for y in yticks], fontsize=18); # use
LaTeX formatted labels
```

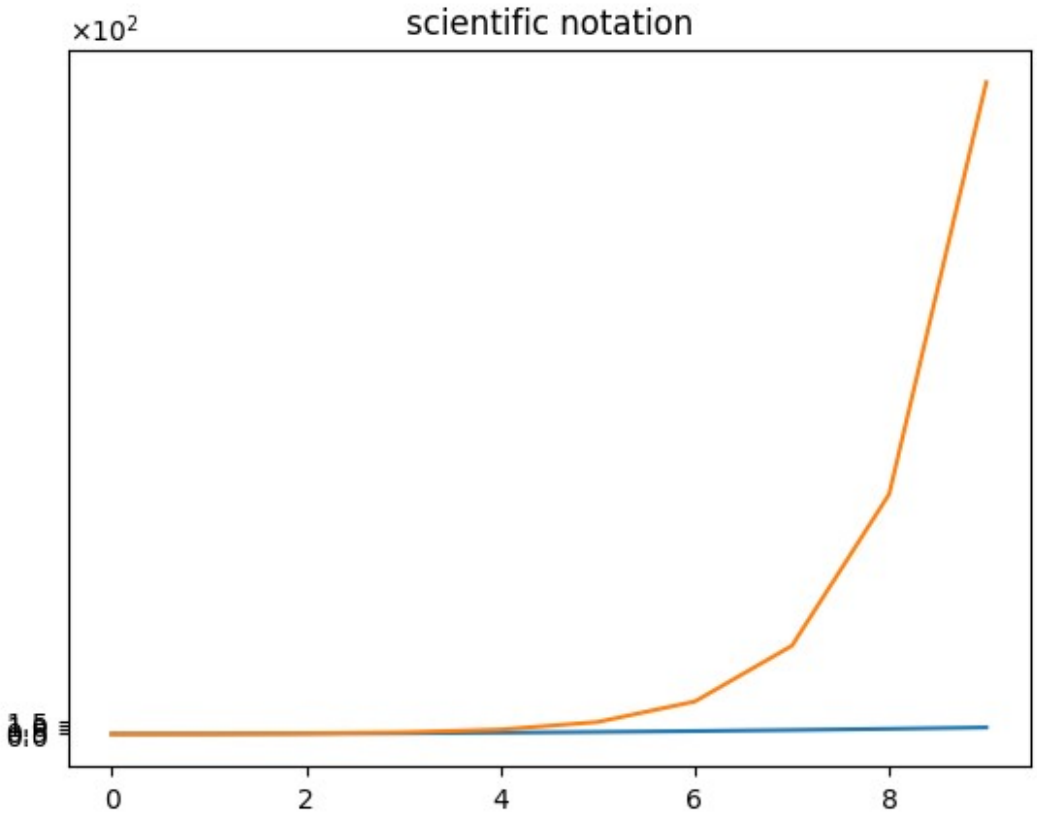


There are a number of more advanced methods for controlling major and minor tick placement in matplotlib figures, such as automatic placement according to different policies. See http://matplotlib.org/api/ticker_api.html for details.

Scientific notation

With large numbers on axes, it is often better use scientific notation:

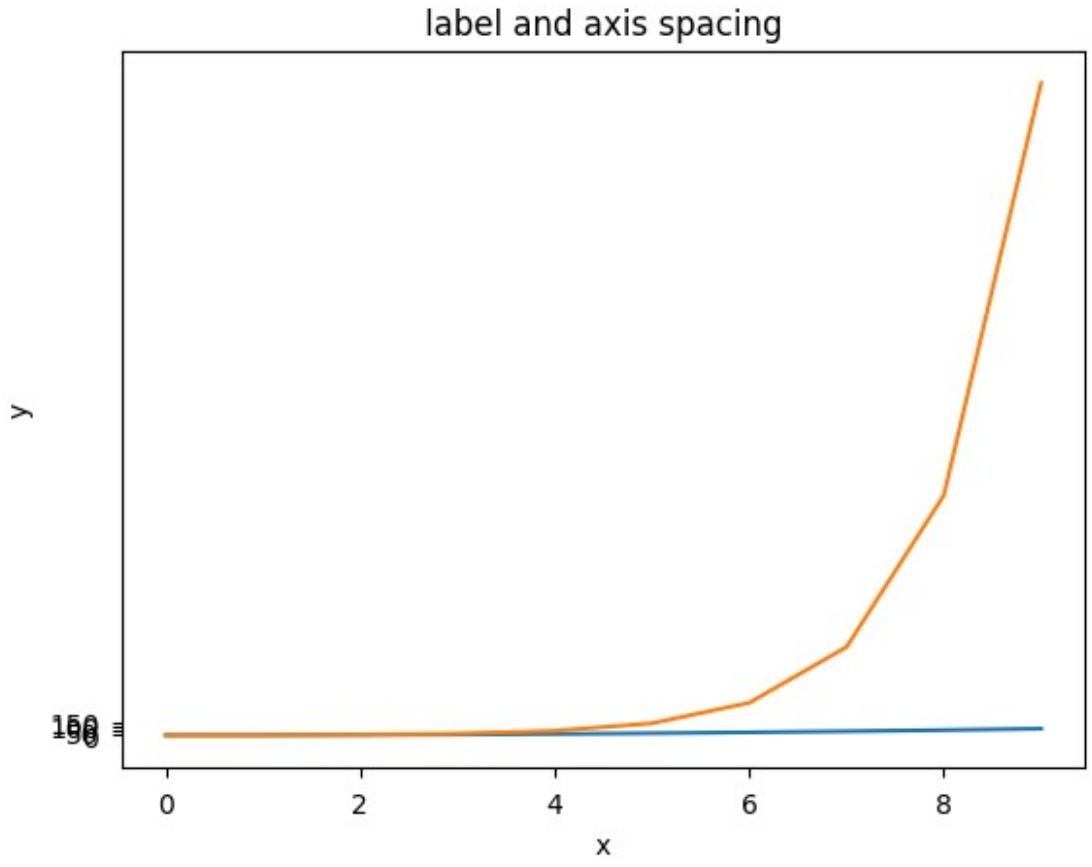
```
fig, ax = plt.subplots(1, 1)

ax.plot(x, x**2, x, np.exp(x))
ax.set_title("scientific notation")

ax.set_yticks([0, 50, 100, 150])

from matplotlib import ticker
formatter = ticker.ScalarFormatter(useMathText=True)
formatter.set_scientific(True)
formatter.set_powerlimits((-1,1))
ax.yaxis.set_major_formatter(formatter)
```



Axis number and axis label spacing

```
import matplotlib as matplotlib
```

```
# distance between x and y axis and the numbers on the axes
matplotlib.rcParams['xtick.major.pad'] = 5
matplotlib.rcParams['ytick.major.pad'] = 5

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

ax.plot(x, x**2, x, np.exp(x))
ax.set_yticks([0, 50, 100, 150])

ax.set_title("label and axis spacing")

# padding between axis label and axis numbers
ax.xaxis.labelpad = 5
ax.yaxis.labelpad = 5

ax.set_xlabel("x")
ax.set_ylabel("y");
```

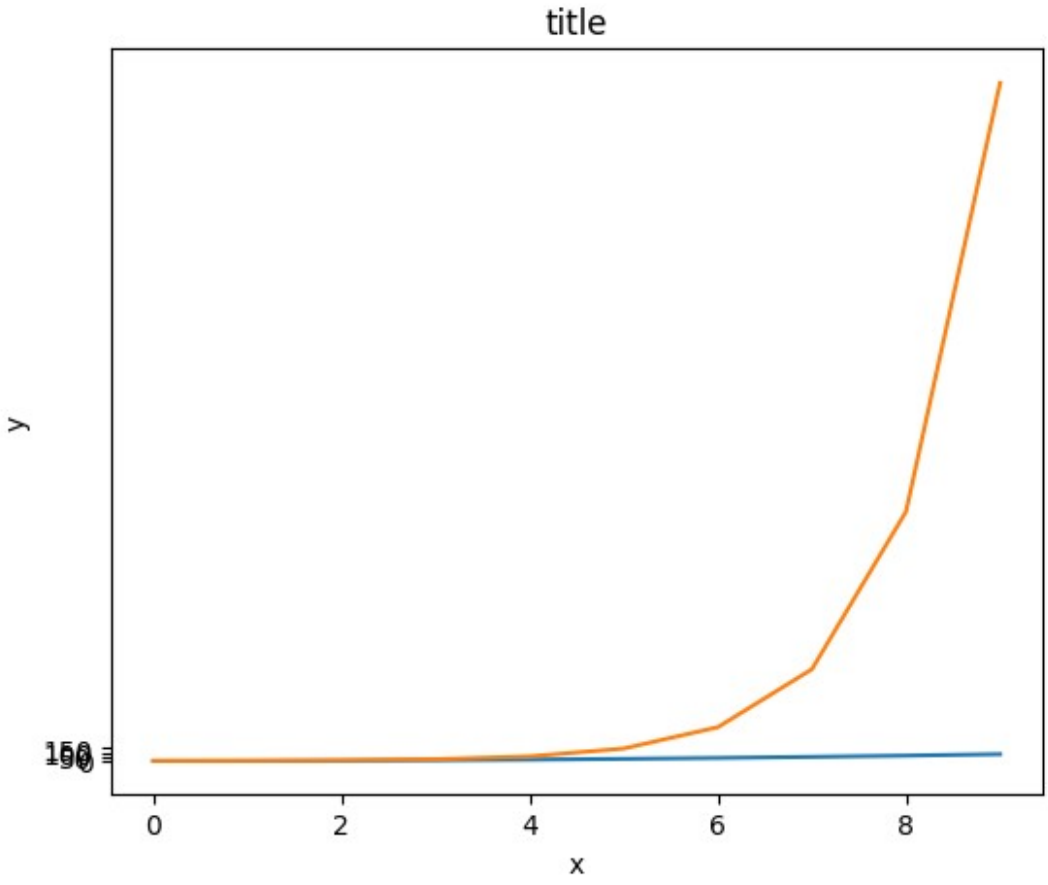



```
# restore defaults
matplotlib.rcParams['xtick.major.pad'] = 3
matplotlib.rcParams['ytick.major.pad'] = 3
```

Axis position adjustments

Unfortunately, when saving figures the labels are sometimes clipped, and it can be necessary to adjust the positions of axes a little bit. This can be done using `subplots_adjust`:

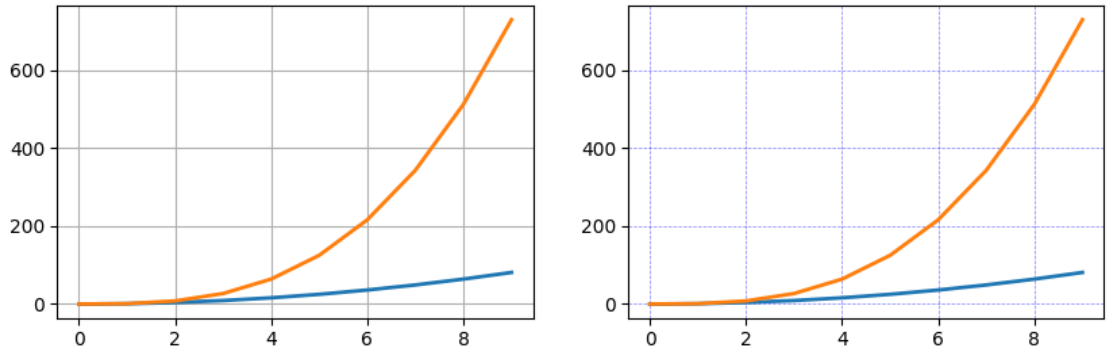
```
fig, ax = plt.subplots(1, 1)

ax.plot(x, x**2, x, np.exp(x))
ax.set_yticks([0, 50, 100, 150])

ax.set_title("title")
ax.set_xlabel("x")
ax.set_ylabel("y")

fig.subplots_adjust(left=0.15, right=.9, bottom=0.1, top=0.9);
```



Axis grid

With the `grid` method in the axis object, we can turn on and off grid lines. We can also customize the appearance of the grid lines using the same keyword arguments as the `plot` function:

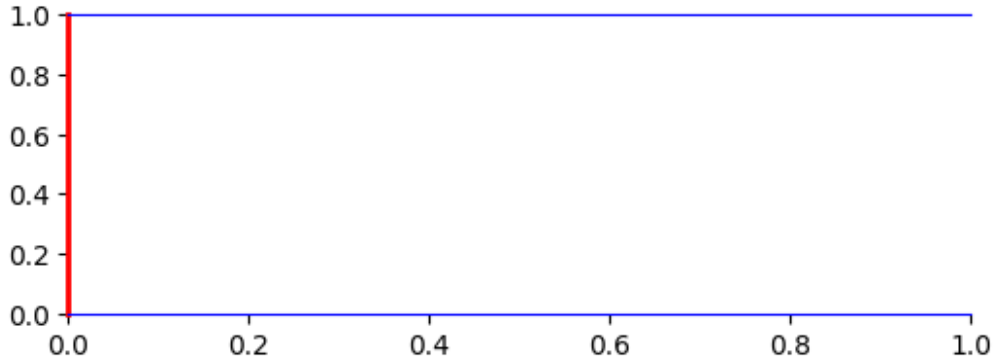
```
fig, axes = plt.subplots(1, 2, figsize=(10,3))

# default grid appearance
axes[0].plot(x, x**2, x, x**3, lw=2)
axes[0].grid(True)

# custom grid appearance
axes[1].plot(x, x**2, x, x**3, lw=2)
axes[1].grid(color='b', alpha=0.5, linestyle='dashed', linewidth=0.5)
```



Axis spines

We can also change the properties of axis spines:

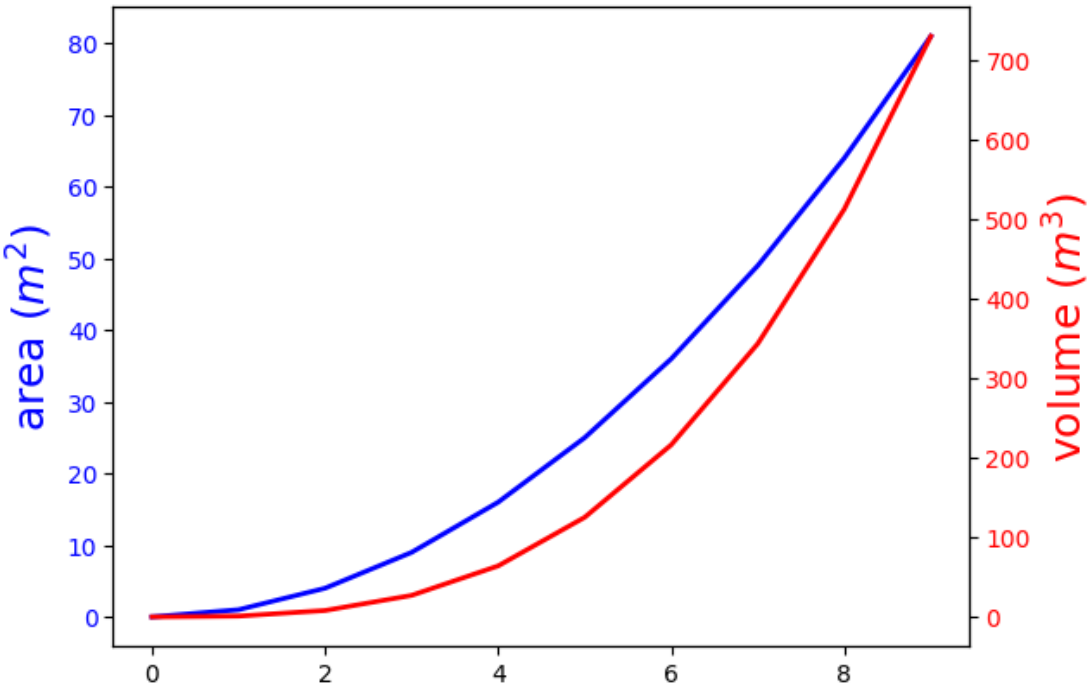
```
fig, ax = plt.subplots(figsize=(6,2))

ax.spines['bottom'].set_color('blue')
ax.spines['top'].set_color('blue')

ax.spines['left'].set_color('red')
ax.spines['left'].set_linewidth(2)

# turn off axis spine to the right
ax.spines['right'].set_color("none")
ax.yaxis.tick_left() # only ticks on the left side
```



Twin axes

Sometimes it is useful to have dual x or y axes in a figure; for example, when plotting curves with different units together. Matplotlib supports this with the `twinx` and `twiny` functions:

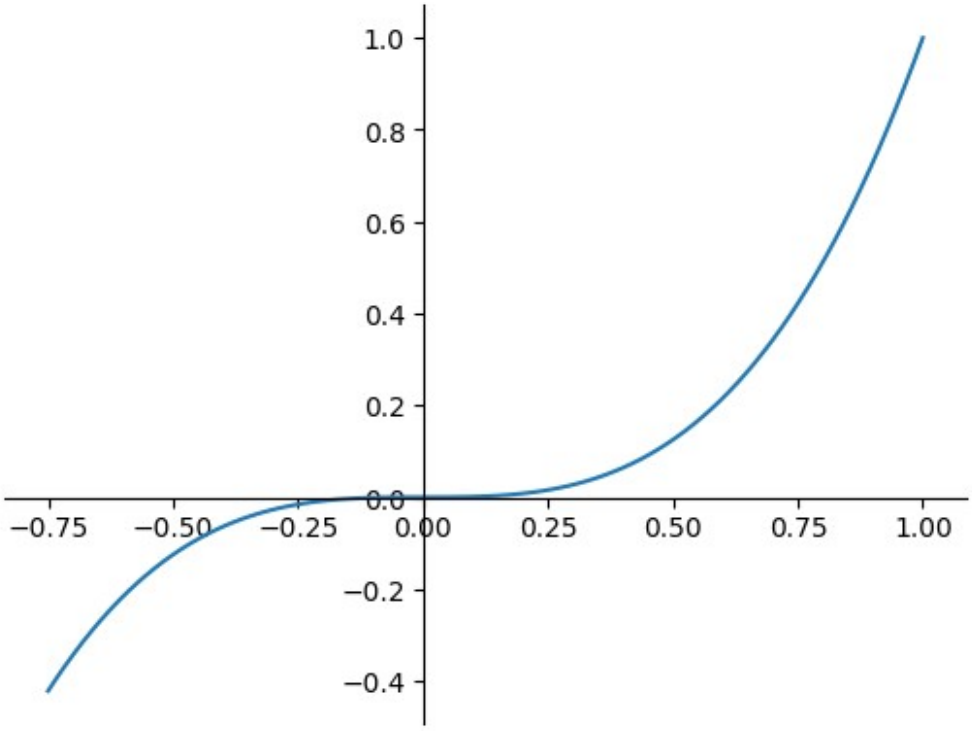
```
fig, ax1 = plt.subplots()

ax1.plot(x, x**2, lw=2, color="blue")
ax1.set_ylabel(r"area $(m^2)$", fontsize=18, color="blue")
for label in ax1.get_yticklabels():
    label.set_color("blue")

ax2 = ax1.twinx()
ax2.plot(x, x**3, lw=2, color="red")
ax2.set_ylabel(r"volume $(m^3)$", fontsize=18, color="red")
for label in ax2.get_yticklabels():
    label.set_color("red")
```



Axes where x and y is zero

```
fig, ax = plt.subplots()

ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')

ax.xaxis.set_ticks_position('bottom')
ax.spines['bottom'].set_position(('data',0)) # set position of x spine
to x=0
```

```
ax.yaxis.set_ticks_position('left')
ax.spines['left'].set_position(('data',0))   # set position of y spine
to y=0

xx = np.linspace(-0.75, 1., 100)
ax.plot(xx, xx**3);
```

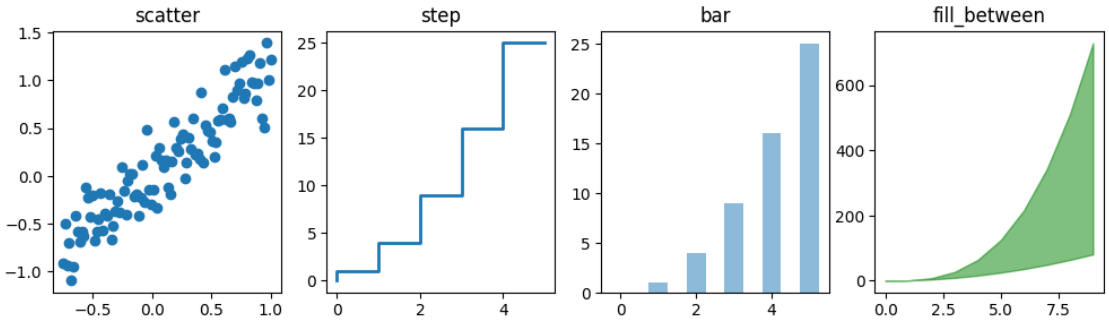


Other 2D plot styles

In addition to the regular `plot` method, there are a number of other functions for generating different kind of plots. See the matplotlib plot gallery for a complete list of available plot types: http://matplotlib.org/gallery.html. Some of the more useful ones are show below:

```
n = np.array([0,1,2,3,4,5])

fig, axes = plt.subplots(1, 4, figsize=(12,3))

axes[0].scatter(xx, xx + 0.25*np.random.randn(len(xx)))
axes[0].set_title("scatter")

axes[1].step(n, n**2, lw=2)
axes[1].set_title("step")

axes[2].bar(n, n**2, align="center", width=0.5, alpha=0.5)
axes[2].set_title("bar")
```

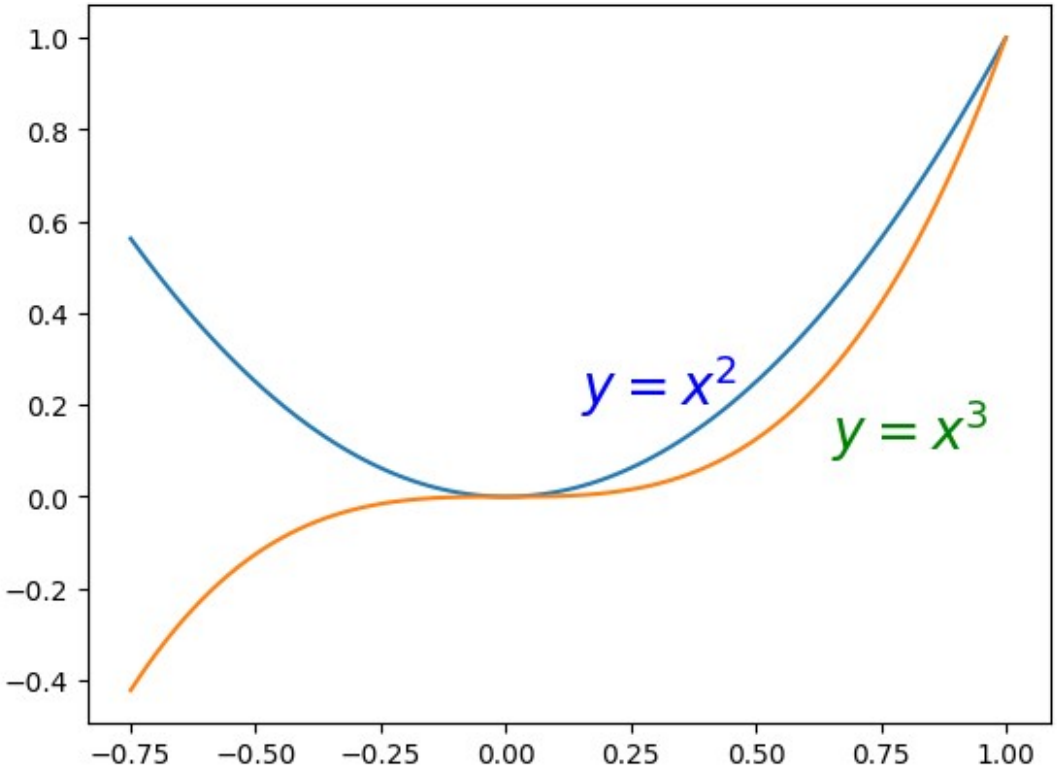
```
axes[3].fill_between(x, x**2, x**3, color="green", alpha=0.5);
axes[3].set_title("fill_between");
```



Text annotation

Annotating text in matplotlib figures can be done using the `text` function. It supports LaTeX formatting just like axis label texts and titles:

```
fig, ax = plt.subplots()

ax.plot(xx, xx**2, xx, xx**3)

ax.text(0.15, 0.2, r"$y=x^2$", fontsize=20, color="blue")
ax.text(0.65, 0.1, r"$y=x^3$", fontsize=20, color="green");
```

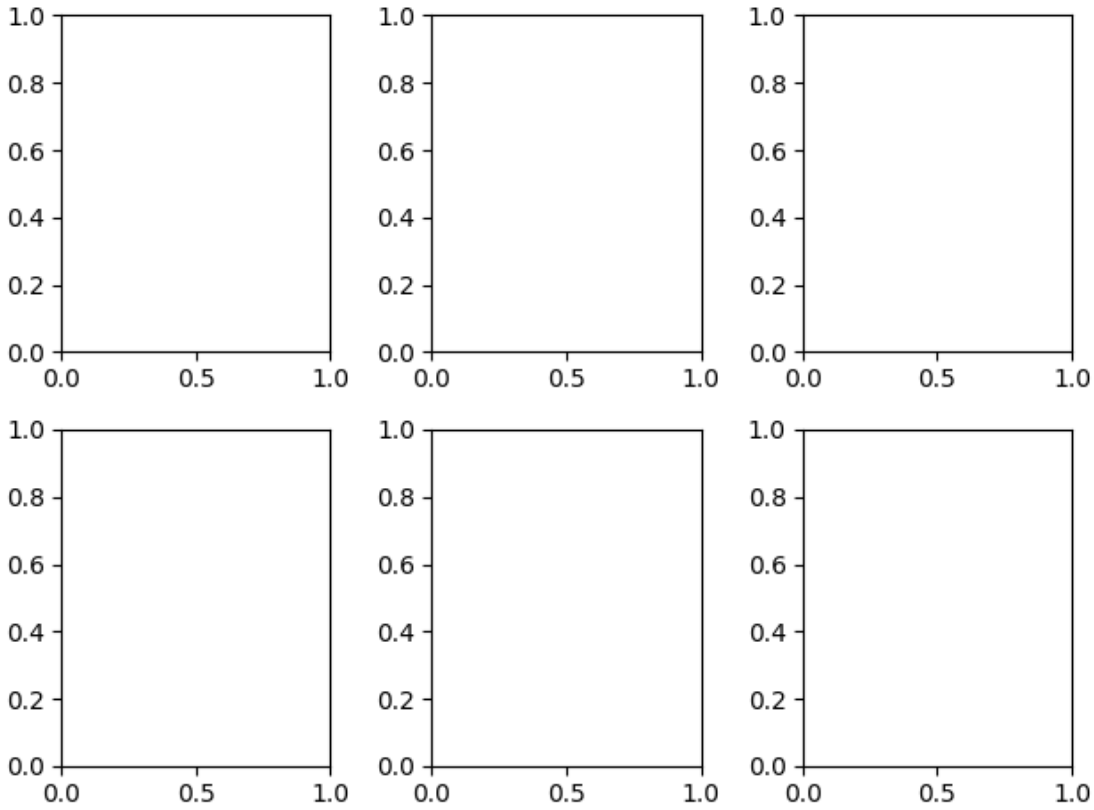


Figures with multiple subplots and insets

Axes can be added to a matplotlib Figure canvas manually using `fig.add_axes` or using a sub-figure layout manager such as `subplots`, `subplot2grid`, or `gridspec`:

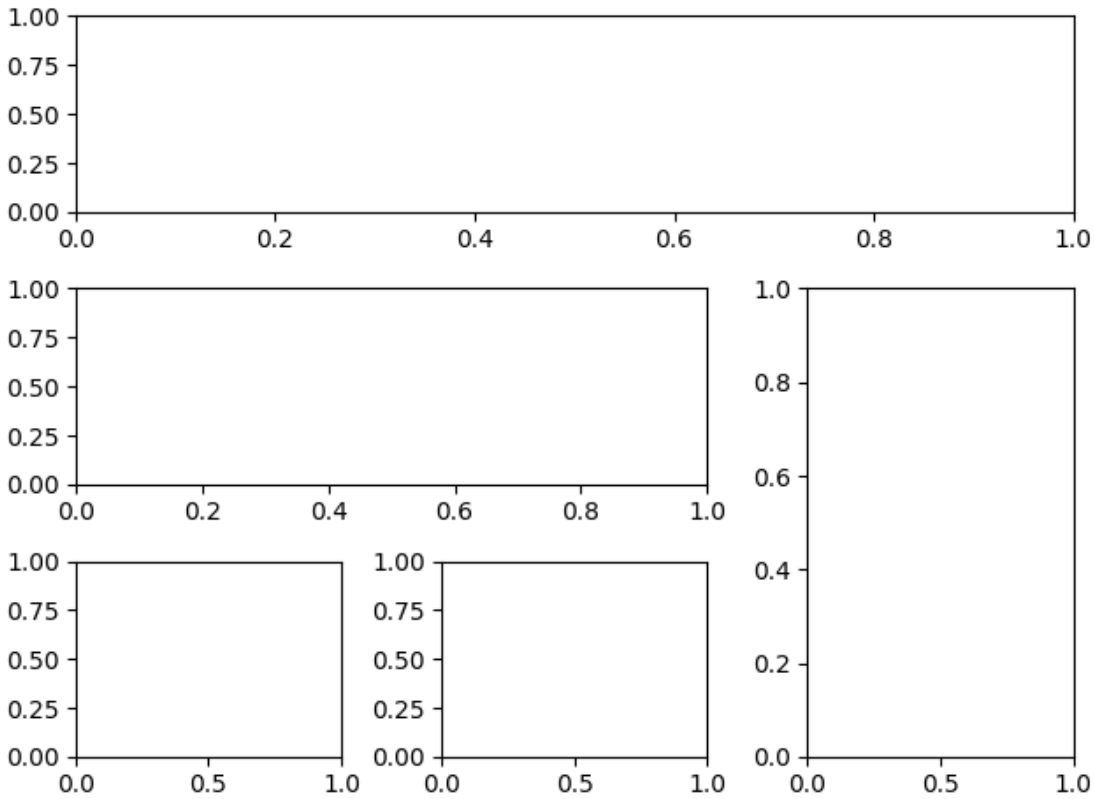
subplots

```
fig, ax = plt.subplots(2, 3)
fig.tight_layout()
```



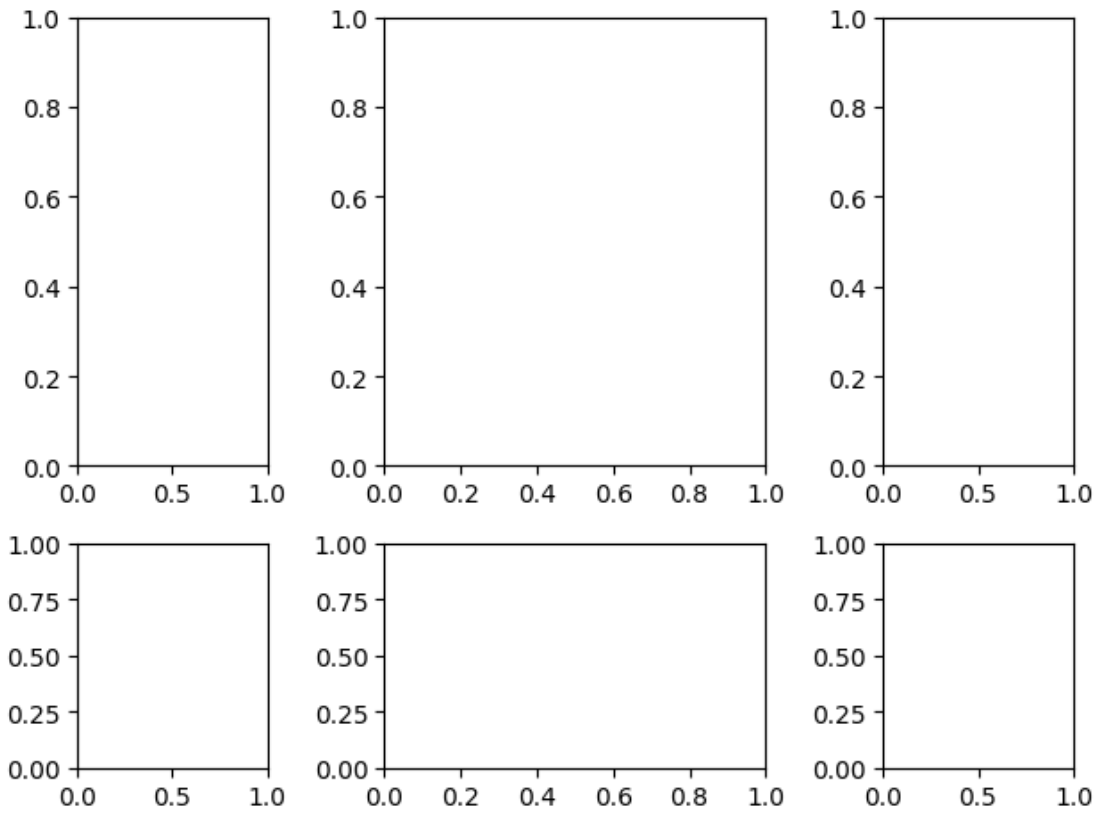
subplot2grid

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



gridspec

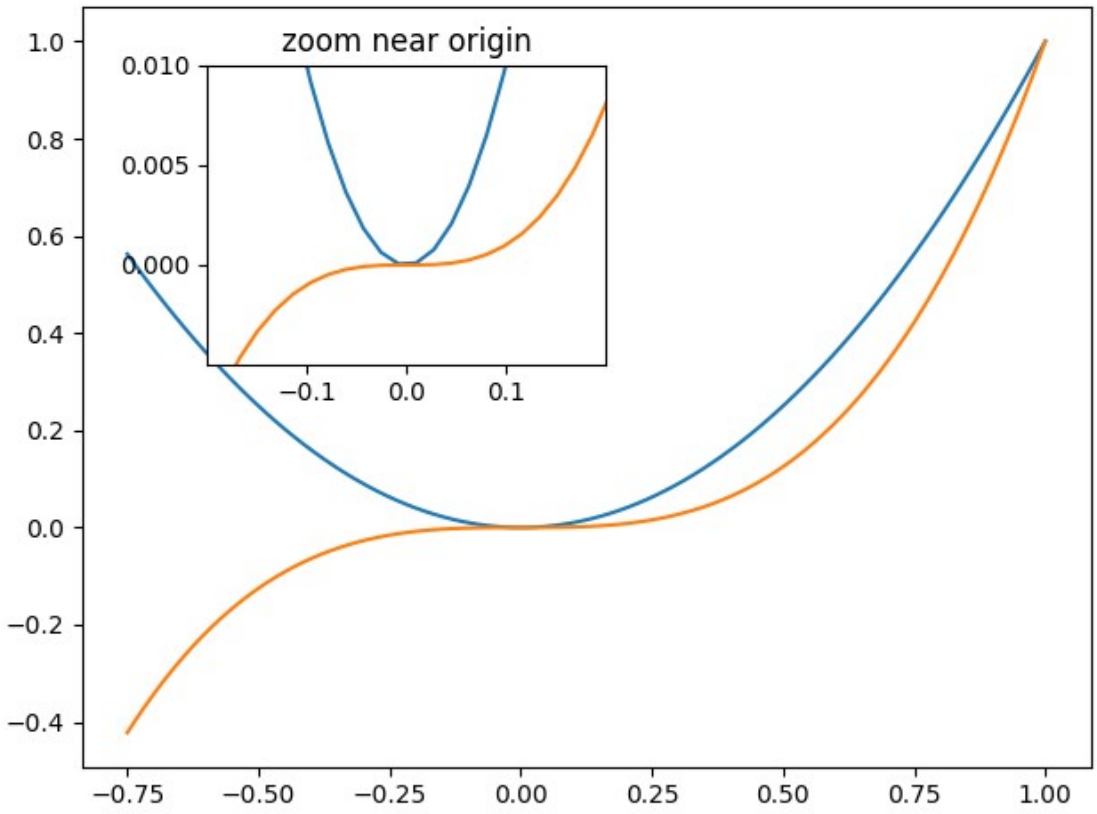
```
import matplotlib.gridspec as gridspec

fig = plt.figure()

gs = gridspec.GridSpec(2, 3, height_ratios=[2,1],
width_ratios=[1,2,1])
for g in gs:
    ax = fig.add_subplot(g)

fig.tight_layout()
```



add_axes

Manually adding axes with `add_axes` is useful for adding insets to figures:

```
fig, ax = plt.subplots()

ax.plot(xx, xx**2, xx, xx**3)
fig.tight_layout()

# inset
inset_ax = fig.add_axes([0.2, 0.55, 0.35, 0.35]) # X, Y, width, height

inset_ax.plot(xx, xx**2, xx, xx**3)
inset_ax.set_title('zoom near origin')

# set axis range
inset_ax.set_xlim(-.2, .2)
inset_ax.set_ylim(-.005, .01)

# set axis tick locations
inset_ax.set_yticks([0, 0.005, 0.01])
inset_ax.set_xticks([-0.1,0,.1]);
```

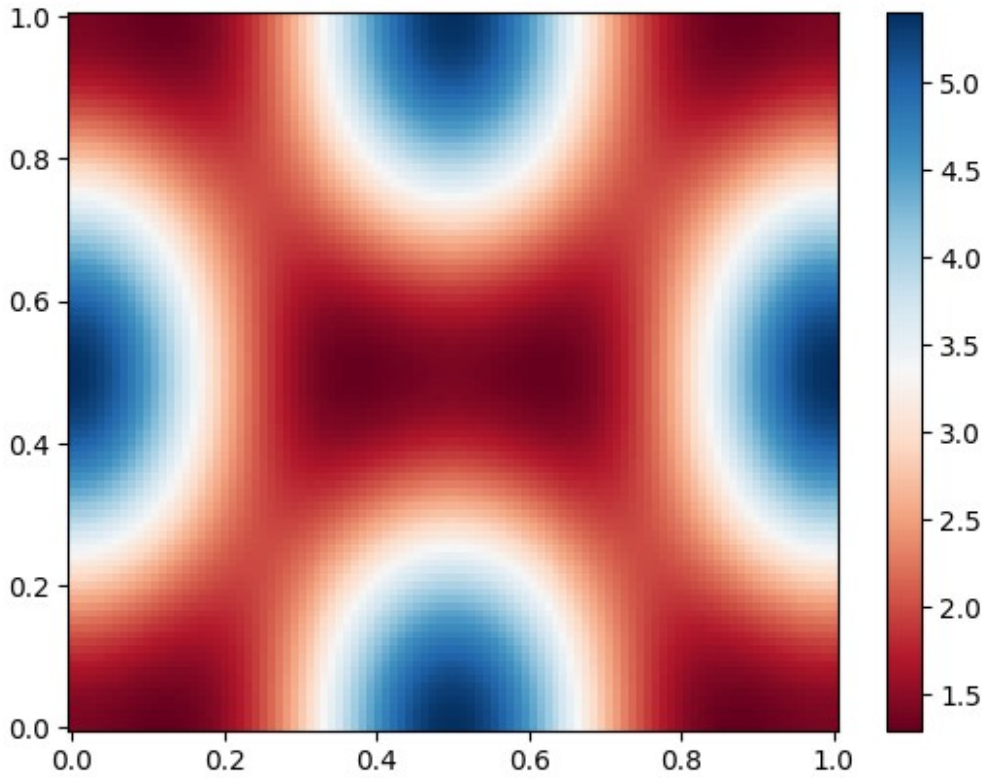


Colormap and contour figures

Colormaps and contour figures are useful for plotting functions of two variables. In most of these functions we will use a colormap to encode one dimension of the data. There are a number of predefined colormaps. It is relatively straightforward to define custom colormaps. For a list of pre-defined colormaps, see:
http://www.scipy.org/Cookbook/Matplotlib/Show_colormaps

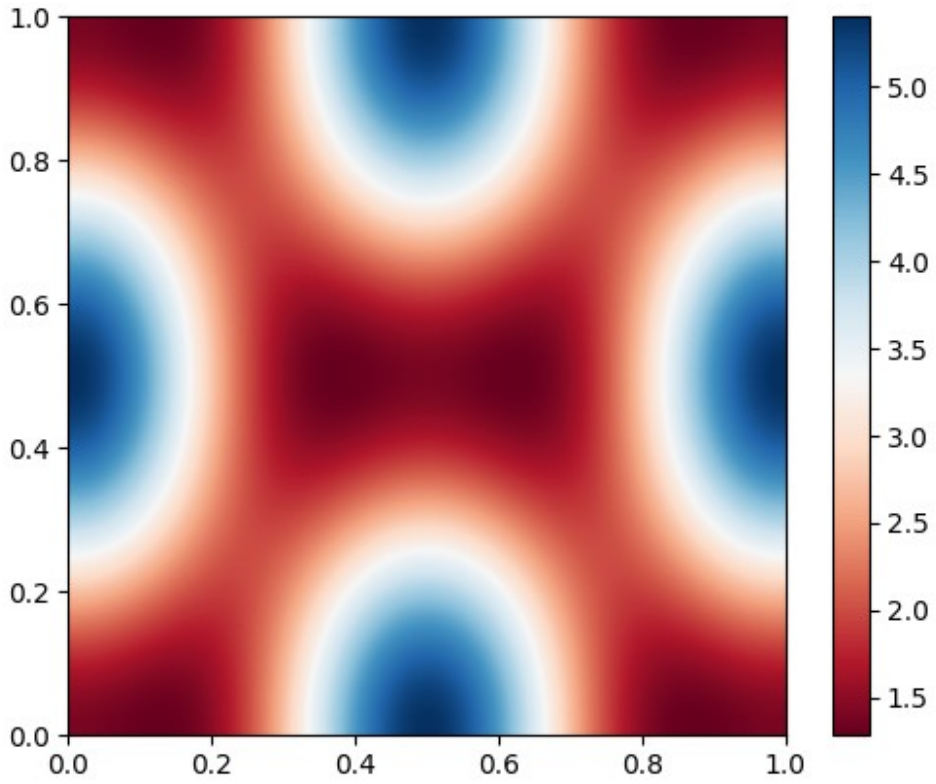
```
alpha = 0.7
phi_ext = 2 * np.pi * 0.5

def flux_qubit_potential(phi_m, phi_p):
    return 2 + alpha - 2 * np.cos(phi_p) * np.cos(phi_m) - alpha *
np.cos(phi_ext - 2*phi_p)

phi_m = np.linspace(0, 2*np.pi, 100)
phi_p = np.linspace(0, 2*np.pi, 100)
X,Y = np.meshgrid(phi_p, phi_m)
Z = flux_qubit_potential(X, Y).T

color
fig, ax = plt.subplots()

p = ax.pcolor(X/(2*np.pi), Y/(2*np.pi), Z, cmap=matplotlib.cm.RdBu,
```

```
vmin=abs(Z).min(), vmax=abs(Z).max())
cb = fig.colorbar(p, ax=ax)
```



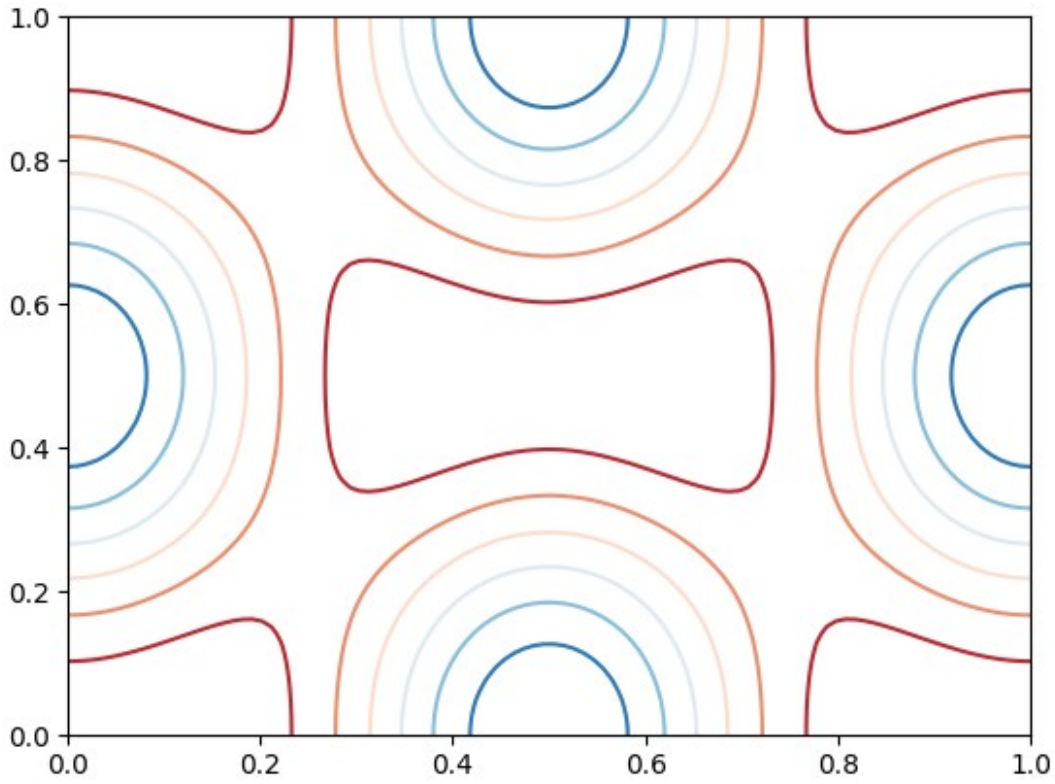
imshow

```
fig, ax = plt.subplots()

im = ax.imshow(Z, cmap=matplotlib.cm.RdBu, vmin=abs(Z).min(),
vmax=abs(Z).max(), extent=[0, 1, 0, 1])
im.set_interpolation('bilinear')

cb = fig.colorbar(im, ax=ax)
```



contour

```
fig, ax = plt.subplots()

cnt = ax.contour(Z, cmap=matplotlib.cm.RdBu, vmin=abs(Z).min(),
vmax=abs(Z).max(), extent=[0, 1, 0, 1])
```



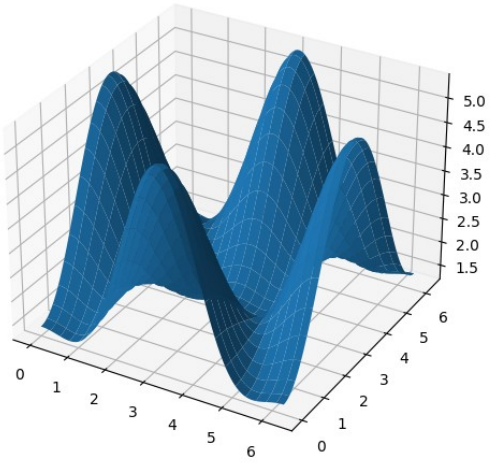
3D figures

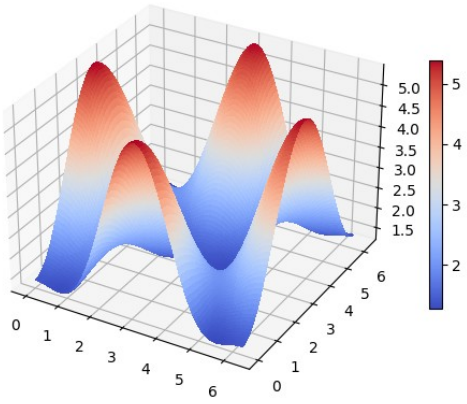
To use 3D graphics in matplotlib, we first need to create an instance of the `Axes3D` class. 3D axes can be added to a matplotlib figure canvas in exactly the same way as 2D axes; or, more conveniently, by passing a `projection='3d'` keyword argument to the `add_axes` or `add_subplot` methods.

```
from mpl_toolkits.mplot3d.axes3d import Axes3D
```

Surface plots

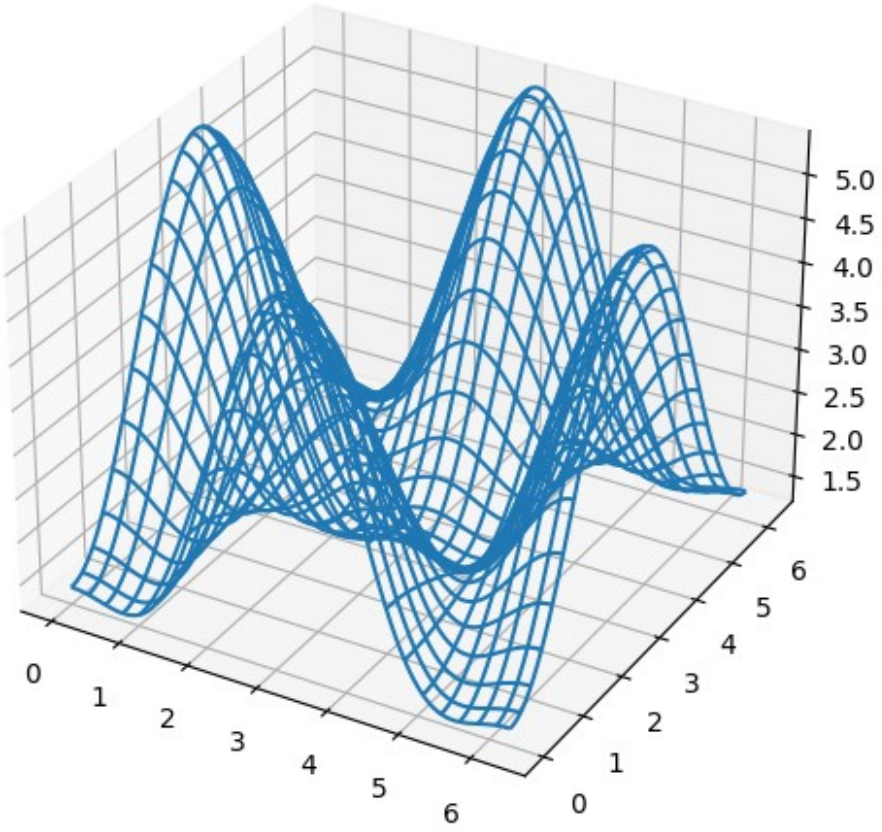
```
fig = plt.figure(figsize=(14,6))

# `ax` is a 3D-aware axis instance because of the projection='3d'
keyword argument to add_subplot
ax = fig.add_subplot(1, 2, 1, projection='3d')

p = ax.plot_surface(X, Y, Z, rstride=4, cstride=4, linewidth=0)

# surface_plot with color grading and color bar
ax = fig.add_subplot(1, 2, 2, projection='3d')
p = ax.plot_surface(X, Y, Z, rstride=1, cstride=1,
cmap=matplotlib.cm.coolwarm, linewidth=0, antialiased=False)
cb = fig.colorbar(p, shrink=0.5)
```





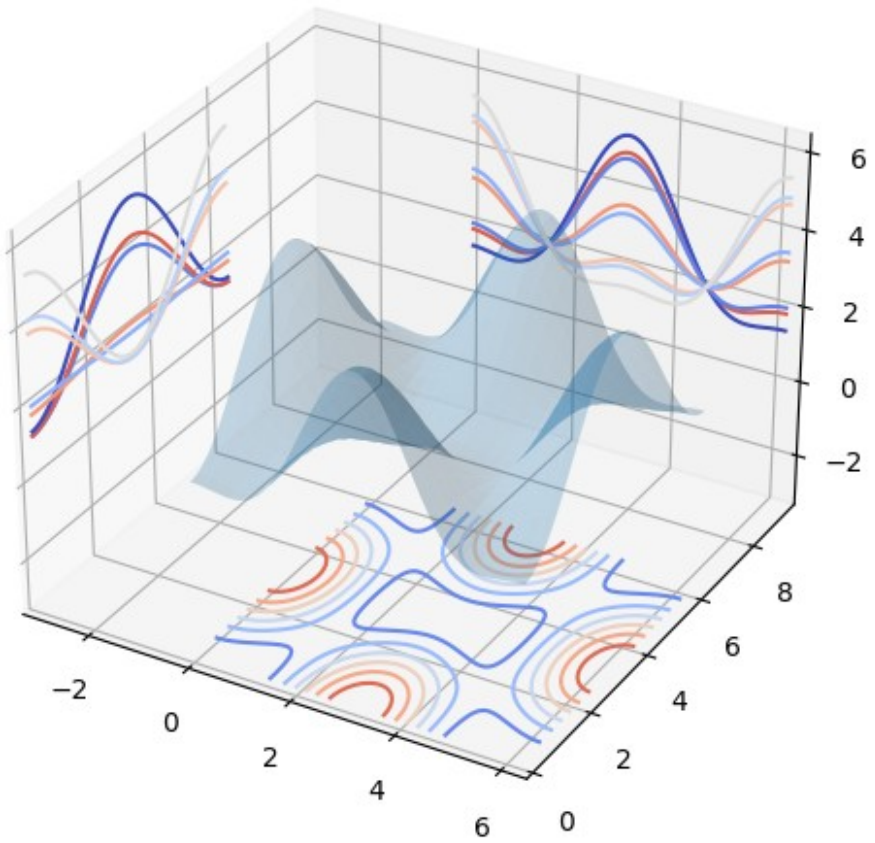
Wire-frame plot

```
fig = plt.figure(figsize=(8,6))

ax = fig.add_subplot(1, 1, 1, projection='3d')

p = ax.plot_wireframe(X, Y, Z, rstride=4, cstride=4)
```



Coutour plots with projections

```
fig = plt.figure(figsize=(8,6))

ax = fig.add_subplot(1,1,1, projection='3d')

ax.plot_surface(X, Y, Z, rstride=4, cstride=4, alpha=0.25)
cset = ax.contour(X, Y, Z, zdir='z', offset=-np.pi,
cmap=matplotlib.cm.coolwarm)
cset = ax.contour(X, Y, Z, zdir='x', offset=-np.pi,
cmap=matplotlib.cm.coolwarm)
cset = ax.contour(X, Y, Z, zdir='y', offset=3*np.pi,
cmap=matplotlib.cm.coolwarm)

ax.set_xlim3d(-np.pi, 2*np.pi);
ax.set_ylim3d(0, 3*np.pi);
ax.set_zlim3d(-np.pi, 2*np.pi);
```



Further reading

- http://www.matplotlib.org - The project web page for matplotlib.
- https://github.com/matplotlib/matplotlib - The source code for matplotlib.

- http://matplotlib.org/gallery.html - A large gallery showcaseing various types of plots matplotlib can create. Highly recommended!