—

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):

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

import matplotlib.pyplot as plt
```

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:

```
%matplotlib inline
```

NOTE: For users running .py scripts in an IDE like PyCharm or Sublime Text Editor. You will not see the plots in a notebook, instead if you are using another editor, you'll use: *plt.show()* at the end of all your plotting commands to have the figure pop up in another window.

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
```

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

```
y
```

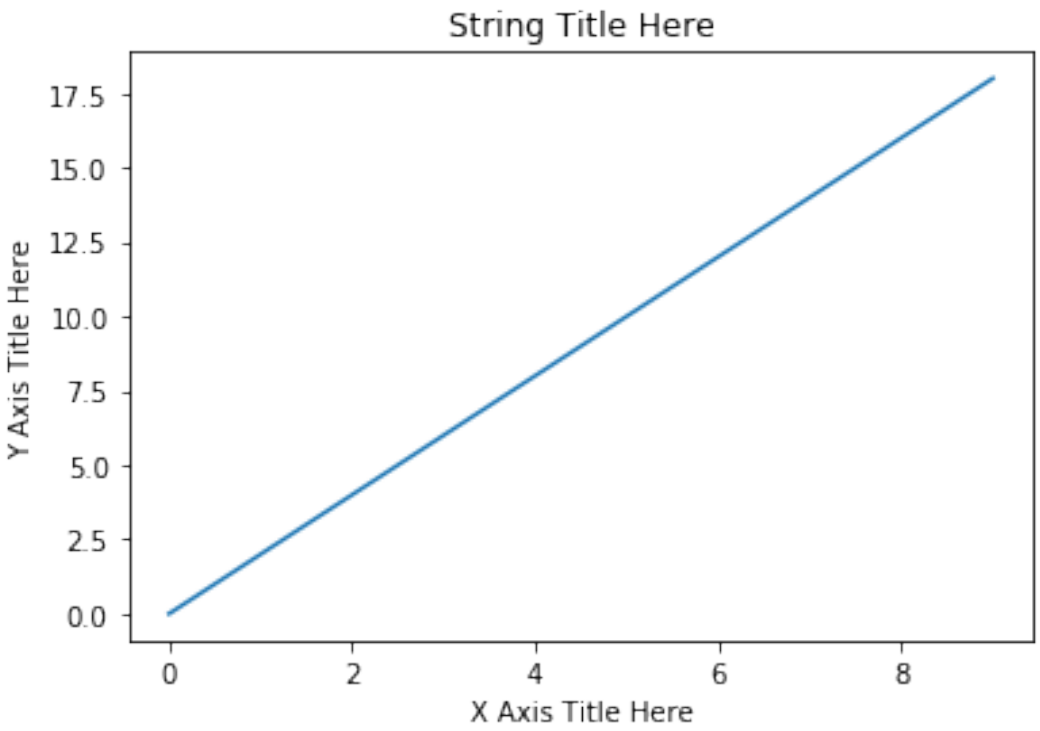
```
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])
```

Using Matplotlib with plt.plot() function calls

Basic Matplotlib Commands

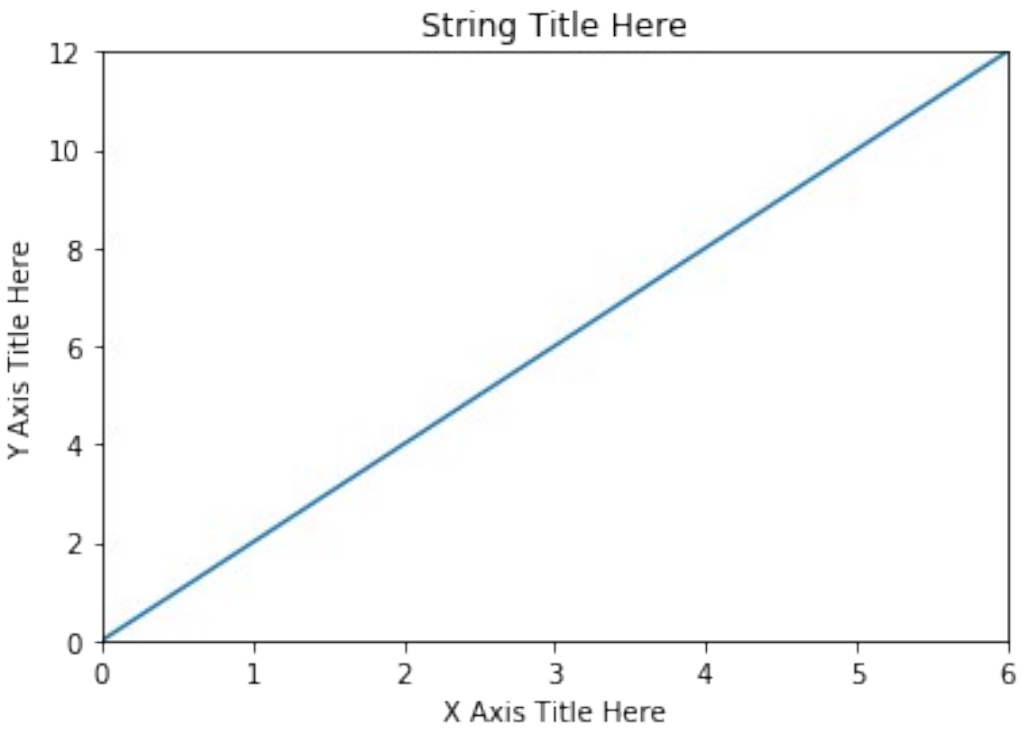
We can create a very simple line plot using the following (I encourage you to pause and use Shift+Tab along the way to check out the document strings for the functions we are using).

```
plt.plot(x, y)
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[]
info
```



Exporting a plot

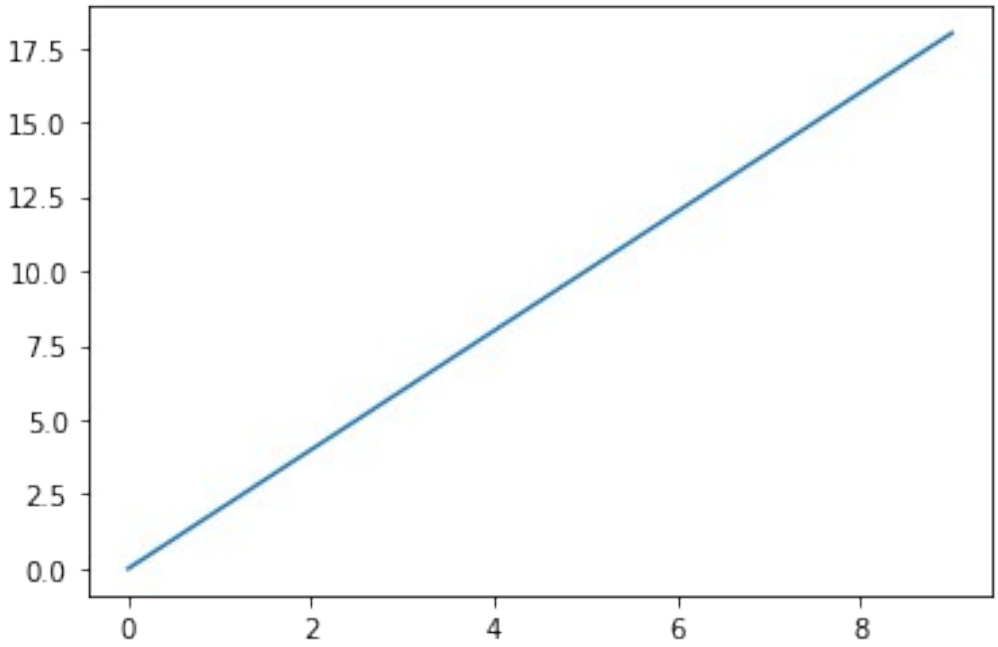
```
help(plt.savefig)
```

```
Help on function savefig in module matplotlib.pyplot:

savefig(*args, **kwargs)
    Save the current figure.

    Call signature::

      savefig(fname, dpi=None, facecolor='w', edgecolor='w',
              orientation='portrait', papertype=None, format=None,
              transparent=False, bbox_inches=None, pad_inches=0.1,
              frameon=None, metadata=None)

    The output formats available depend on the backend being used.

    Parameters
    ----------

    fname : str or PathLike or file-like object
        A path, or a Python file-like object, or
        possibly some backend-dependent object such as
        `matplotlib.backends.backend_pdf.PdfPages`.

        If *format* is not set, then the output format is inferred
from
```

```
        the extension of *fname*, if any, and
from :rc:`savefig.format`
        otherwise.  If *format* is set, it determines the output
format.

        Hence, if *fname* is not a path or has no extension, remember
to
        specify *format* to ensure that the correct backend is used.

    Other Parameters
    ----------------

    dpi : [ *None* | scalar > 0 | 'figure' ]
        The resolution in dots per inch.  If *None*, defaults to
        :rc:`savefig.dpi`.  If 'figure', uses the figure's dpi value.

    quality : [ *None* | 1 <= scalar <= 100 ]
        The image quality, on a scale from 1 (worst) to 95 (best).
        Applicable only if *format* is jpg or jpeg, ignored otherwise.
        If *None*, defaults to :rc:`savefig.jpeg_quality` (95 by
default).
        Values above 95 should be avoided; 100 completely disables the
        JPEG quantization stage.

    optimize : bool
        If *True*, indicates that the JPEG encoder should make an
extra
        pass over the image in order to select optimal encoder
settings.
        Applicable only if *format* is jpg or jpeg, ignored otherwise.
        Is *False* by default.

    progressive : bool
        If *True*, indicates that this image should be stored as a
        progressive JPEG file. Applicable only if *format* is jpg or
        jpeg, ignored otherwise. Is *False* by default.

    facecolor : color spec or None, optional
        The facecolor of the figure; if *None*, defaults to
        :rc:`savefig.facecolor`.

    edgecolor : color spec or None, optional
        The edgecolor of the figure; if *None*, defaults to
        :rc:`savefig.edgecolor`

    orientation : {'landscape', 'portrait'}
        Currently only supported by the postscript backend.

    papertype : str
        One of 'letter', 'legal', 'executive', 'ledger', 'a0' through
```

```
        'a10', 'b0' through 'b10'. Only supported for postscript
        output.

    format : str
        The file format, e.g. 'png', 'pdf', 'svg', ... The behavior
when
        this is unset is documented under *fname*.

    transparent : bool
        If *True*, the axes patches will all be transparent; the
        figure patch will also be transparent unless facecolor
        and/or edgecolor are specified via kwargs.
        This is useful, for example, for displaying
        a plot on top of a colored background on a web page.  The
        transparency of these patches will be restored to their
        original values upon exit of this function.

    bbox_inches : str or `~matplotlib.transforms.Bbox`, optional
        Bbox in inches. Only the given portion of the figure is
        saved. If 'tight', try to figure out the tight bbox of
        the figure. If None, use savefig.bbox

    pad_inches : scalar, optional
        Amount of padding around the figure when bbox_inches is
        'tight'. If None, use savefig.pad_inches

    bbox_extra_artists : list of `~matplotlib.artist.Artist`, optional
        A list of extra artists that will be considered when the
        tight bbox is calculated.

    metadata : dict, optional
        Key/value pairs to store in the image metadata. The supported
keys
        and defaults depend on the image format and backend:

        - 'png' with Agg backend: See the parameter ``metadata`` of
          `~.FigureCanvasAgg.print_png`.
        - 'pdf' with pdf backend: See the parameter ``metadata`` of
          `~.backend_pdf.PdfPages`.
        - 'eps' and 'ps' with PS backend: Only 'Creator' is supported.

    pil_kwargs : dict, optional
        Additional keyword arguments that are passed to
`PIL.Image.save`
        when saving the figure.  Only applicable for formats that are
saved
        using Pillow, i.e. JPEG, TIFF, and (if the keyword is set to a
        non-None value) PNG.
```

```
plt.plot(x,y)
plt.savefig('example.png')
```



—

Matplotlib Figure Object

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

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

import matplotlib.pyplot as plt
```

NOTE: For users running .py scripts in an IDE like PyCharm or Sublime Text Editor. You will not see the plots in a notebook, instead if you are using another editor, you'll use: *plt.show()* at the end of all your plotting commands to have the figure pop up in another window.

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

```
import numpy as np

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)
y = 2 * x

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.

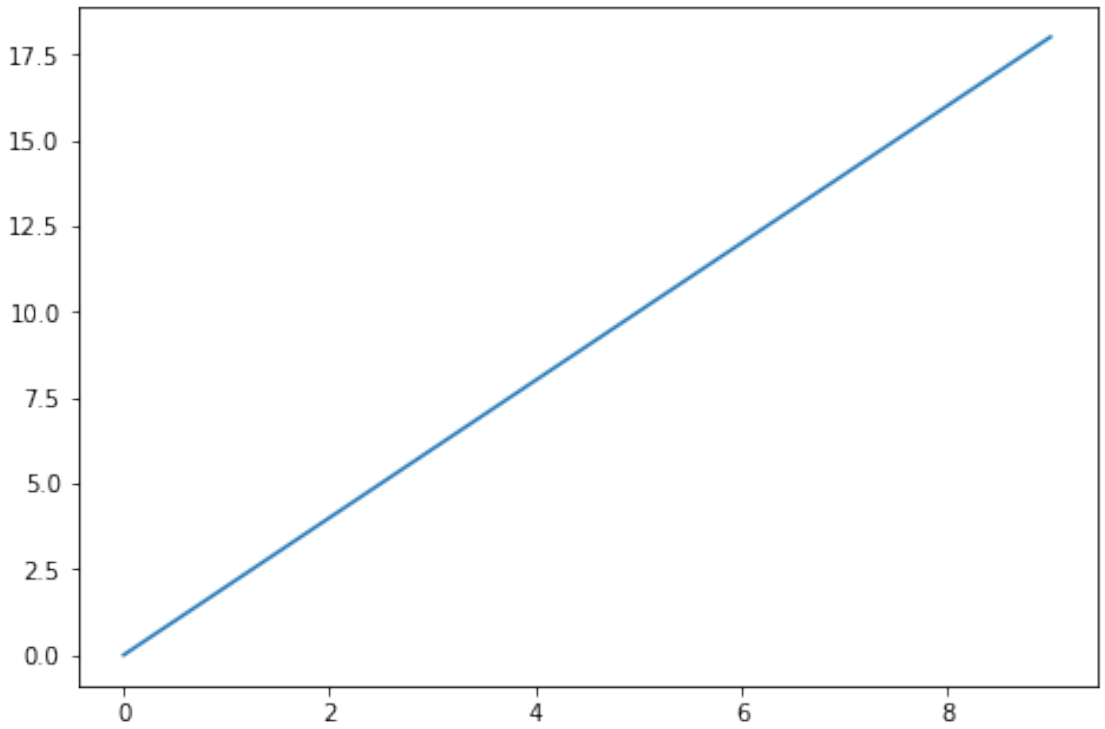
```
# Creates blank canvas
fig = plt.figure()
```

```
<Figure size 432x288 with 0 Axes>
```

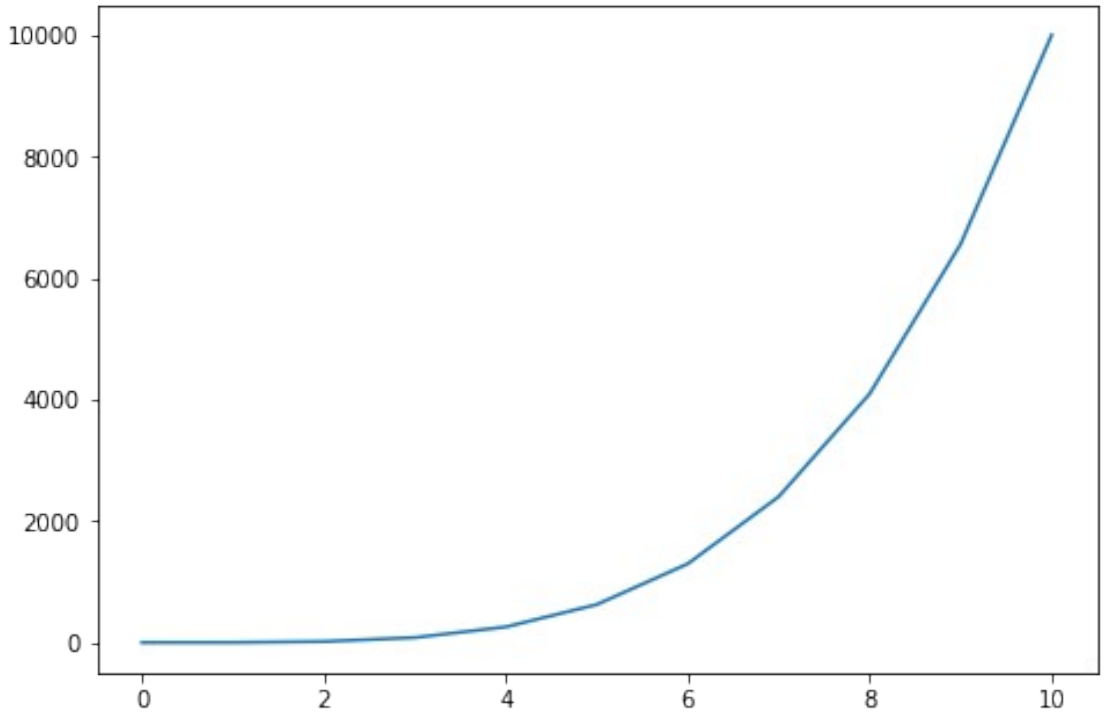
NOTE: ALL THE COMMANDS NEED TO GO IN THE SAME CELL!

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()
```



```
# 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.

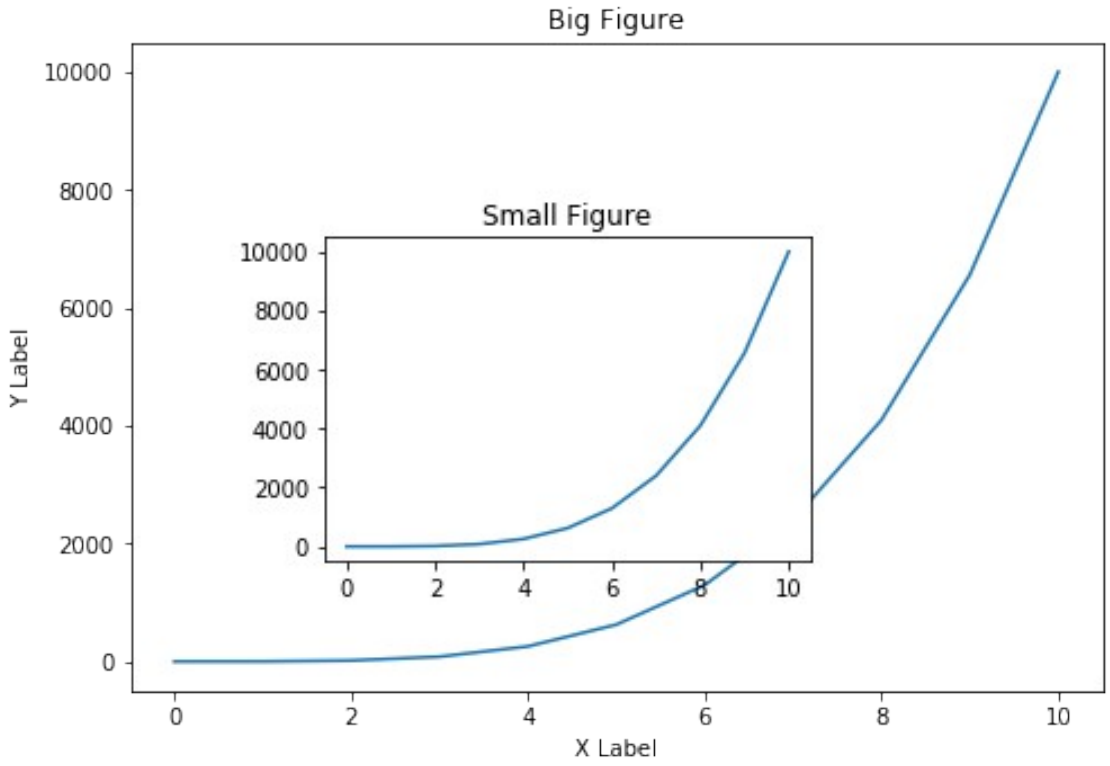
```
type(fig)
```

```
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
fig = plt.figure()
```

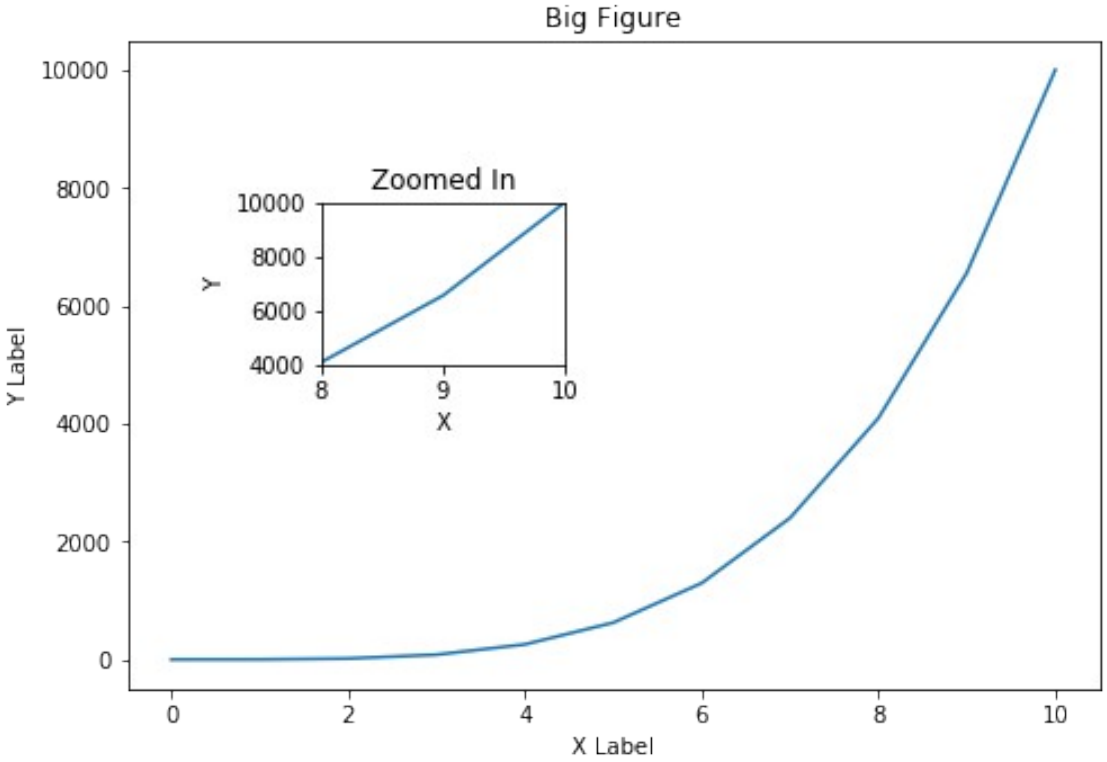
```
axes1 = fig.add_axes([0, 0, 1, 1]) # Large figure
axes2 = fig.add_axes([0.2, 0.2, 0.5, 0.5]) # Smaller figure

# 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');
```



Let's move the small figure and edit its parameters.

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

axes1 = fig.add_axes([0, 0, 1, 1]) # Large figure
axes2 = fig.add_axes([0.2, 0.5, 0.25, 0.25]) # 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_xlim(8,10)
axes2.set_ylim(4000,10000)
axes2.set_xlabel('X')
axes2.set_ylabel('Y')
axes2.set_title('Zoomed In');
```



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)
```

```
[<matplotlib.lines.Line2D at 0x1cd42ad2888>]
```

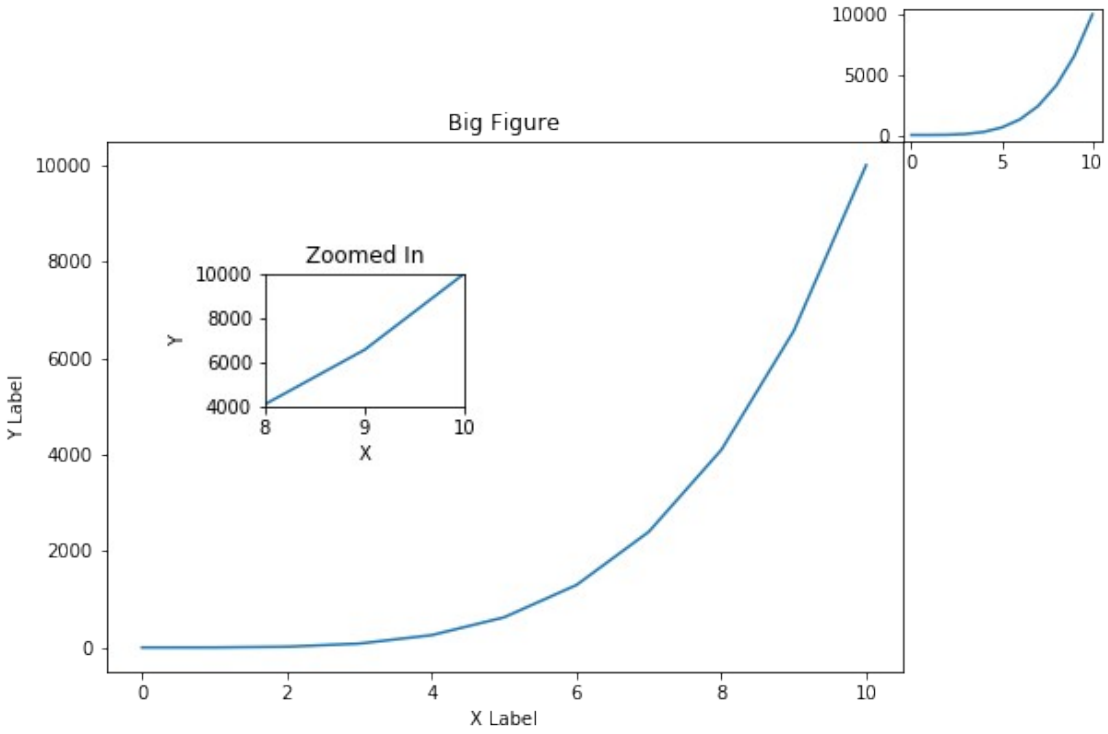
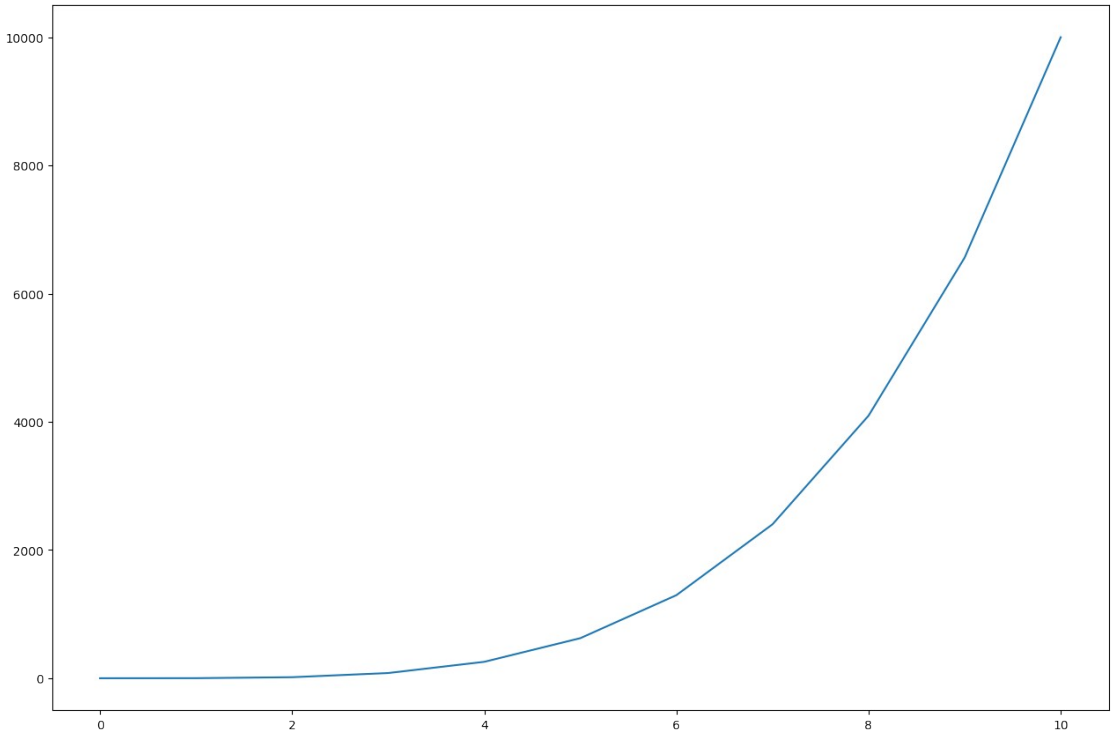


Figure Parameters

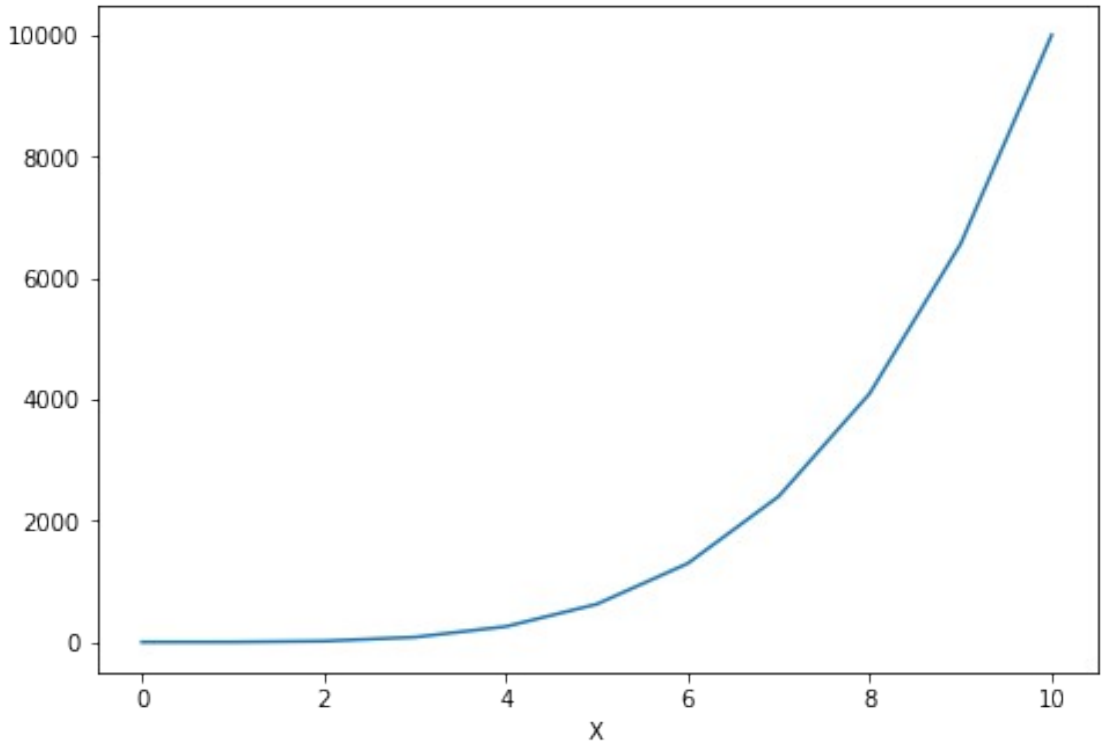
```
# Creates blank canvas
fig = plt.figure(figsize=(12,8),dpi=100)

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

axes1.plot(a,b)
```

```
[<matplotlib.lines.Line2D at 0x1cd42d53848>]
```

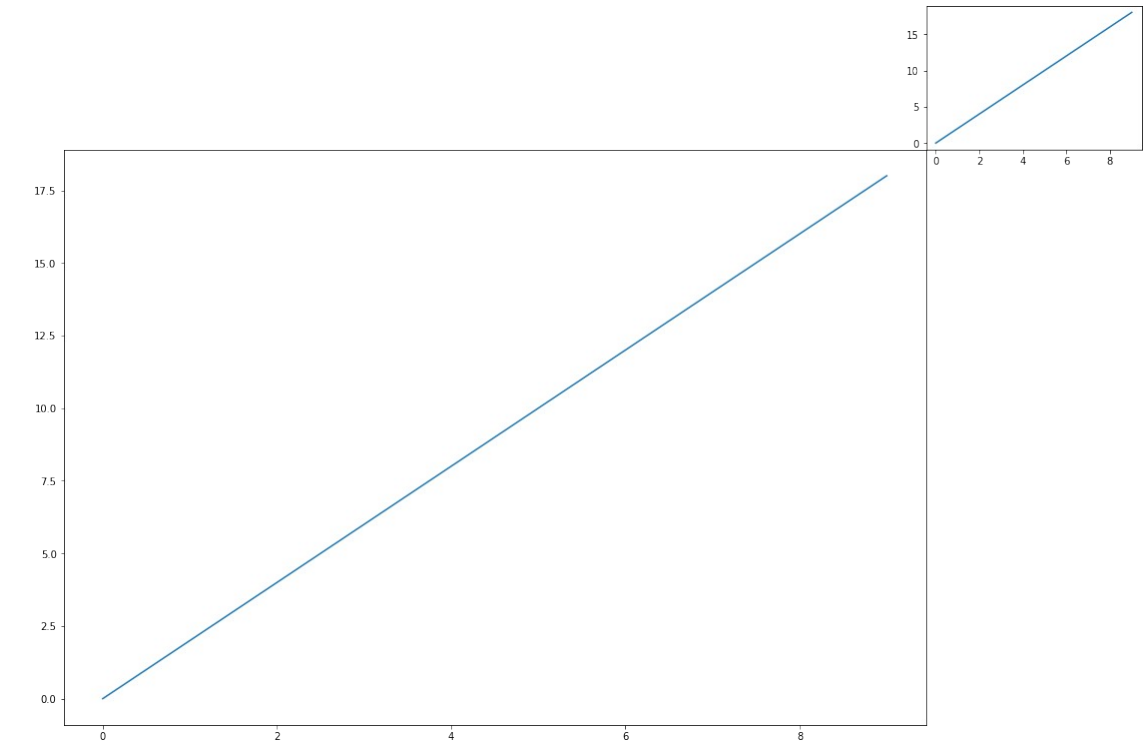


Exporting a Figure

```
fig = plt.figure()

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

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

# bbox_inches ='tight' automatically makes sure the bounding box is
correct
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')
```


15
10
5
0
0
2
4
6
8
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
0
2
4
6
8

—

Matplotlib Sub Plots

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

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

import matplotlib.pyplot as plt
```

NOTE: For users running .py scripts in an IDE like PyCharm or Sublime Text Editor. You will not see the plots in a notebook, instead if you are using another editor, you'll use: *plt.show()* at the end of all your plotting commands to have the figure pop up in another window.

The Data

```
import numpy as np

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)
y = 2 * x

x

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

y

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

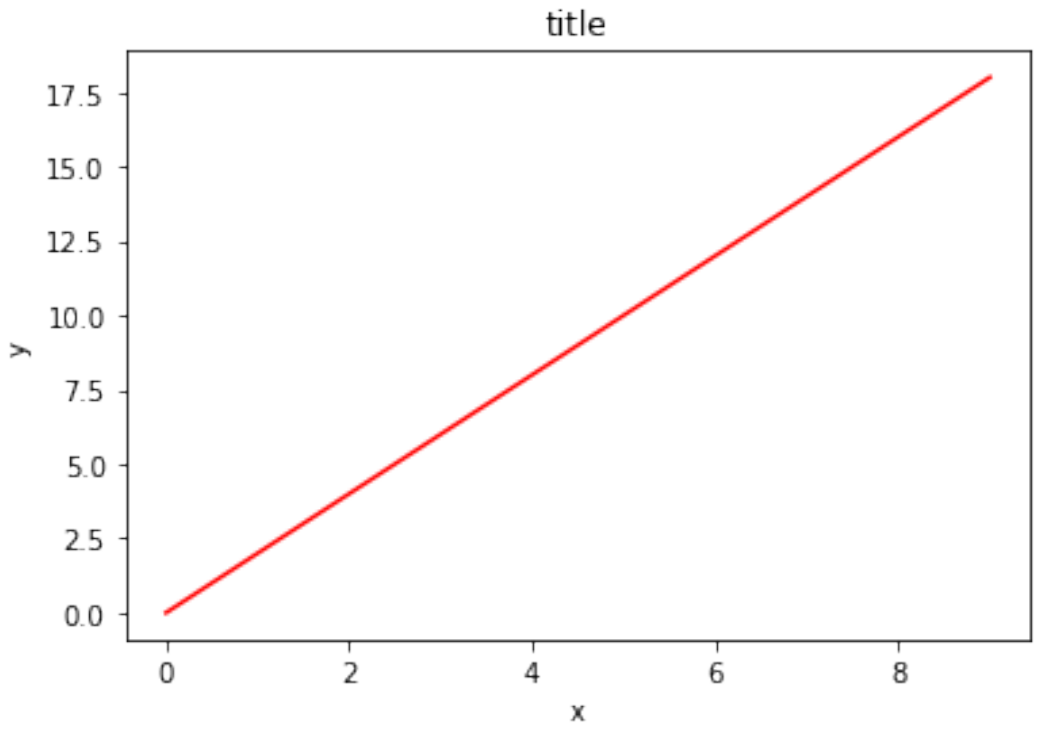
plt.subplots()

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:

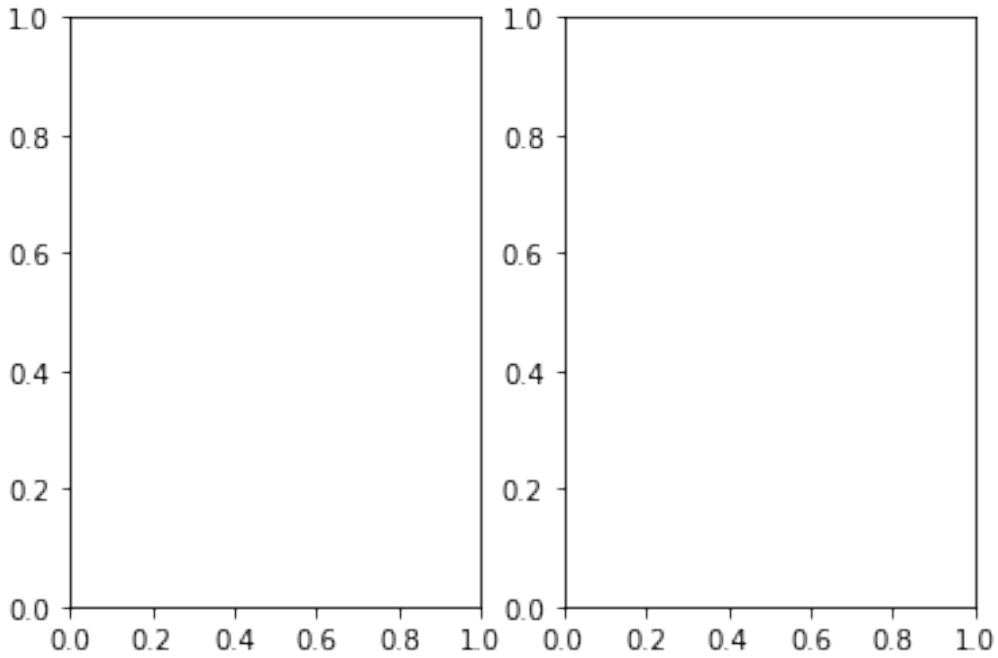
```
# Use similar to plt.figure() except use tuple unpacking to grab fig
and axes
fig, axes = plt.subplots()

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



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



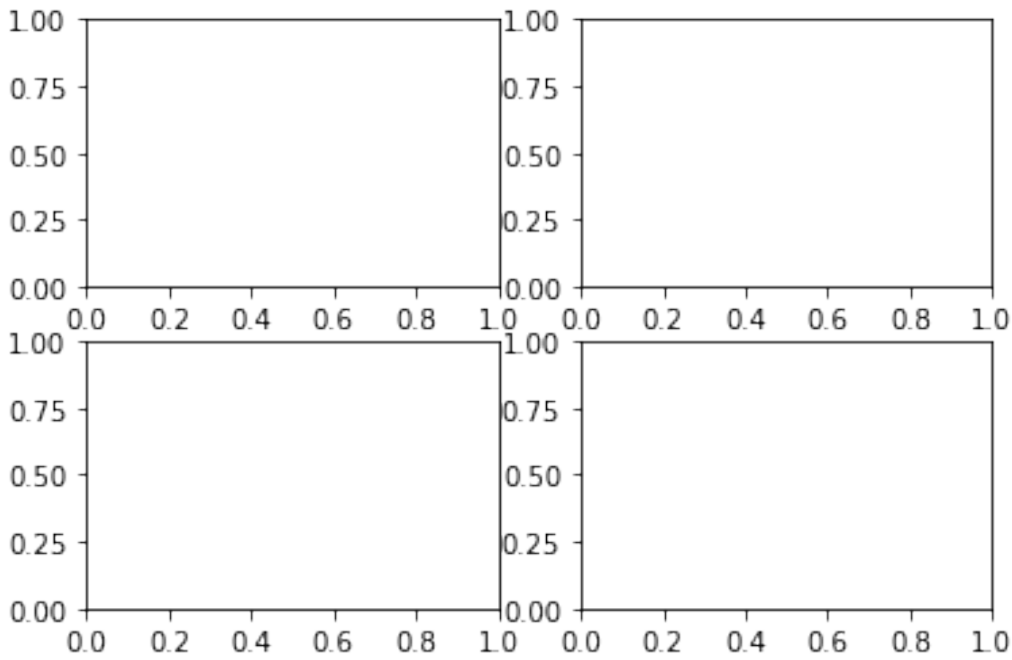

```
# Axes is an array of axes to plot on
axes
```

```
array([<matplotlib.axes._subplots.AxesSubplot object at
0x0000023521E20588>,
       <matplotlib.axes._subplots.AxesSubplot object at
0x0000023521E5D8C8>],
      dtype=object)
```

```
axes.shape
```

```
(2,)
```

```
# Empty canvas of 2 by 2 subplots
fig, axes = plt.subplots(nrows=2, ncols=2)
```




```
axes
```

```
array([[<matplotlib.axes._subplots.AxesSubplot object at
0x0000023521ED5E48>,
        <matplotlib.axes._subplots.AxesSubplot object at
0x0000023521F09D88>],
       [<matplotlib.axes._subplots.AxesSubplot object at
0x0000023521F45308>,
        <matplotlib.axes._subplots.AxesSubplot object at
0x0000023521F79D88>]],
      dtype=object)
```

```
axes.shape
```

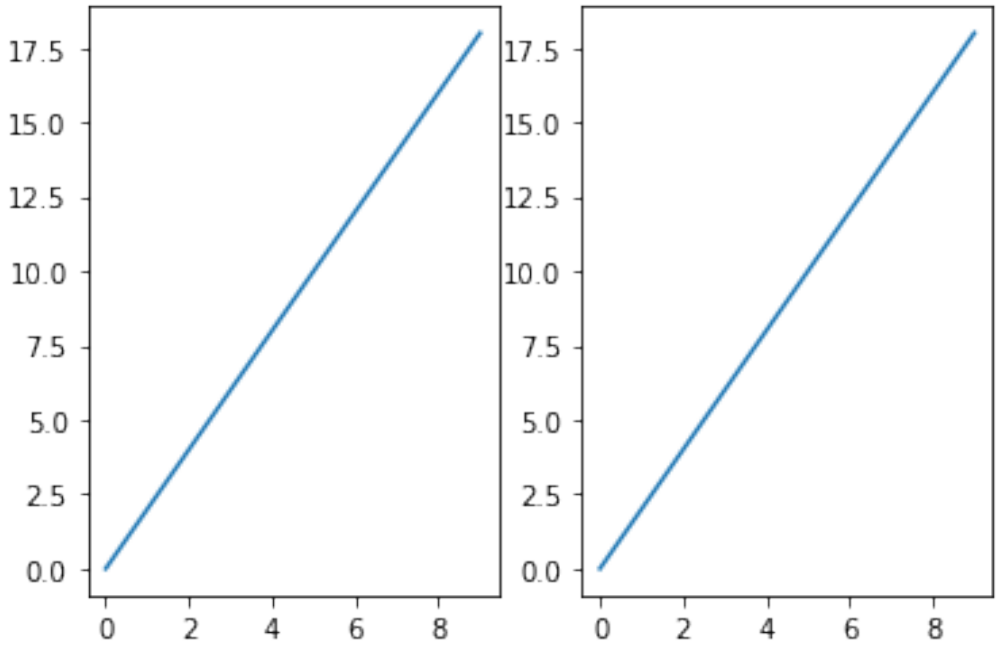
```
(2, 2)
```

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.

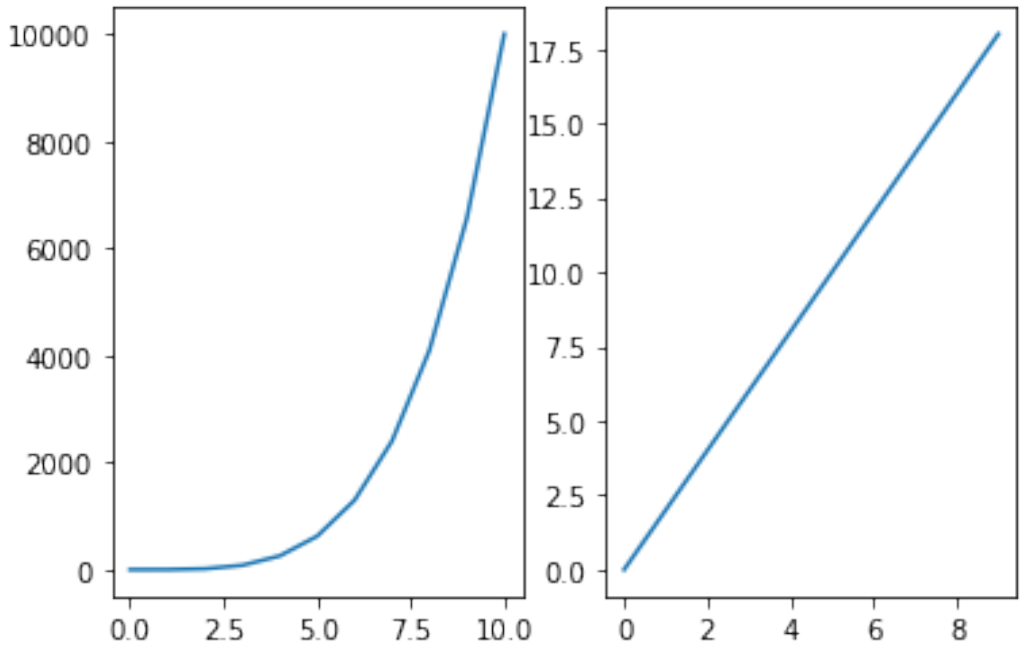
Let's explore this, make sure this is all in the same cell:

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

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




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

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

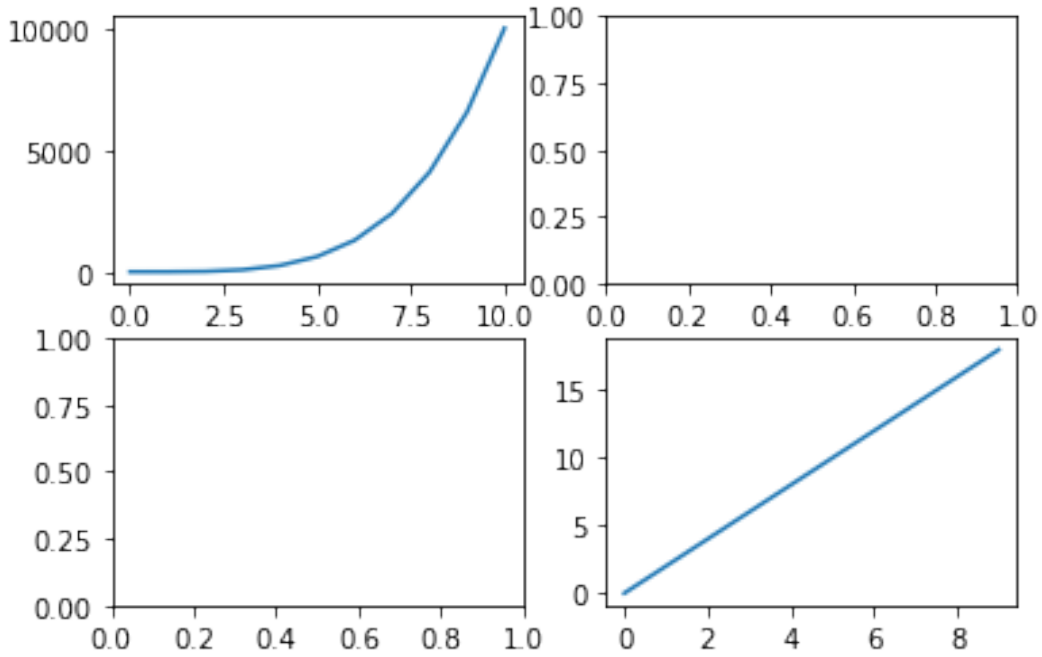
```
[<matplotlib.lines.Line2D at 0x2352216ce88>]
```




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

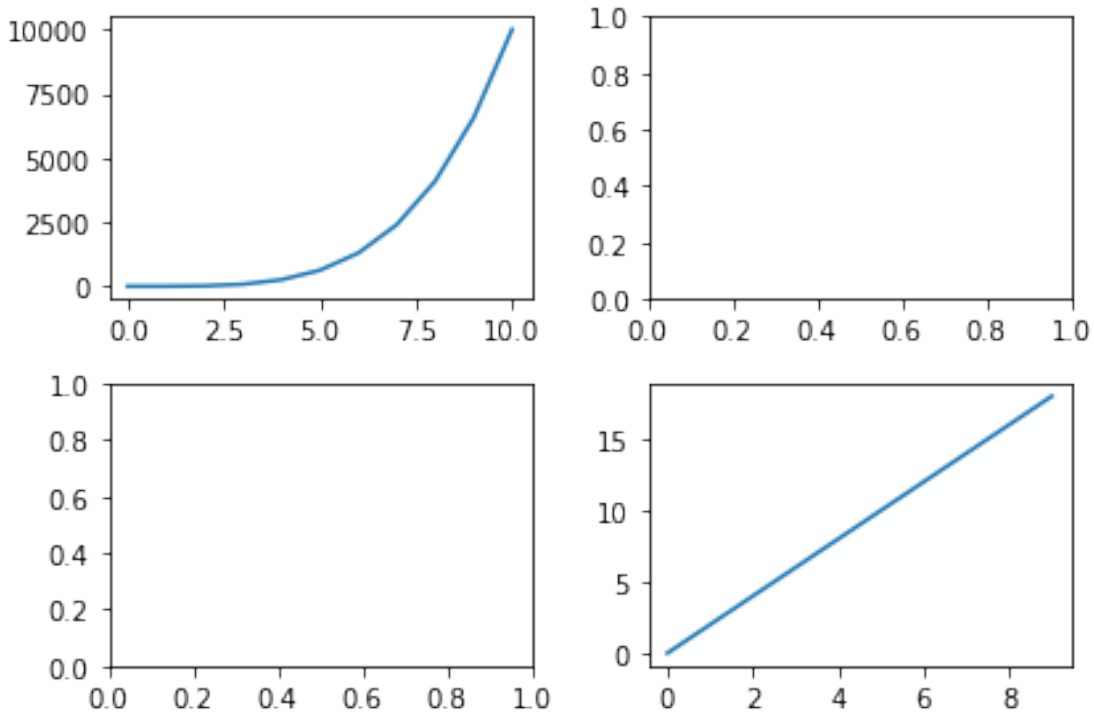
```
axes[0][0].plot(a,b)
axes[1][1].plot(x,y)
```

```
[<matplotlib.lines.Line2D at 0x2352229c648>]
```



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:

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

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

plt.tight_layout()
```

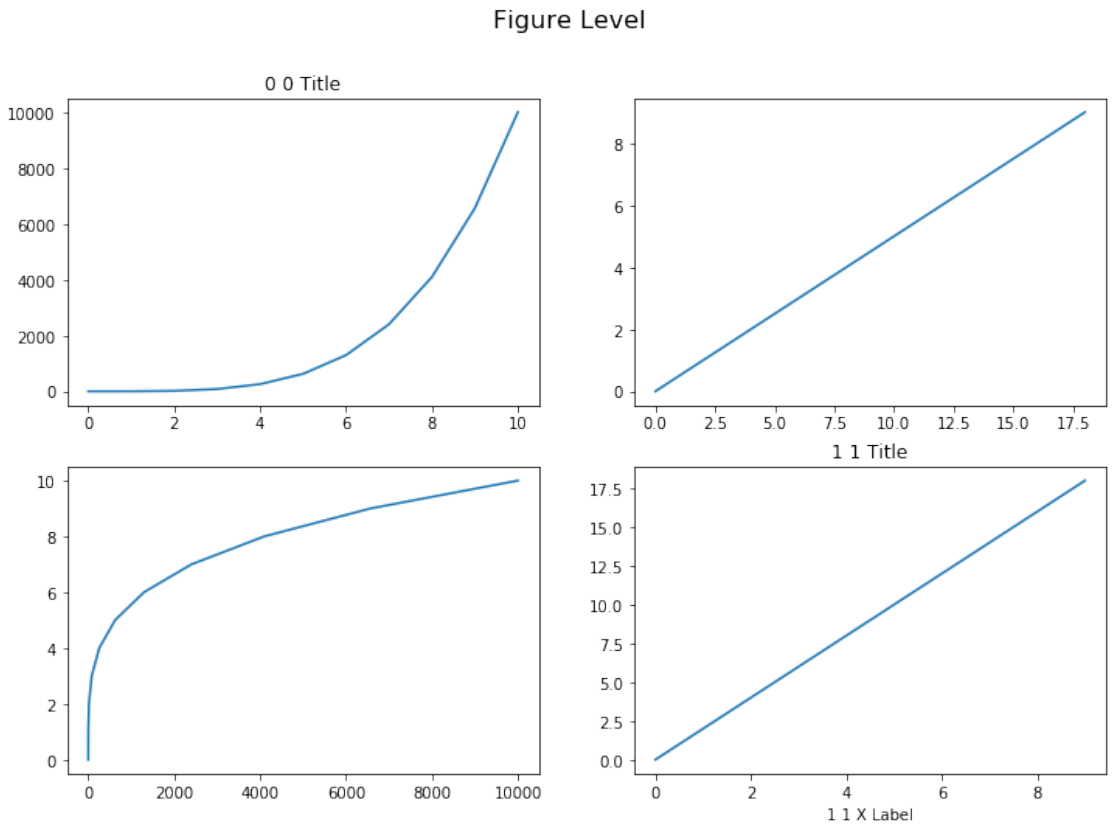


Parameters on subplots()

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()
```



Manual spacing on subplots()

Use .subplots_adjust to adjust spacing manually.

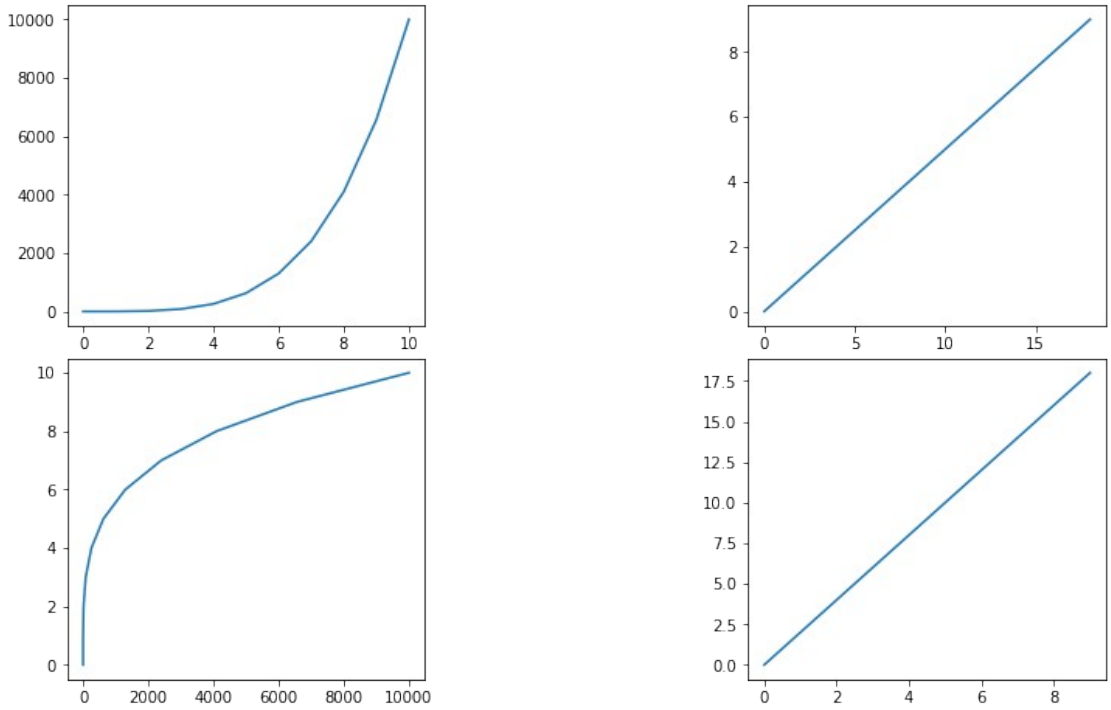
Full Details Here:
https://matplotlib.org/3.2.2/api/_as_gen/matplotlib.pyplot.subplots_adjust.html

Example from link:

- left = 0.125 # the left side of the subplots of the figure
- right = 0.9 # the right side of the subplots of the figure
- bottom = 0.1 # the bottom of the subplots of the figure
- top = 0.9 # the top of the subplots of the figure
- wspace = 0.2 # the amount of width reserved for space between subplots, # expressed as a fraction of the average axis width
- hspace = 0.2 # the amount of height reserved for space between subplots, # expressed as a fraction of the average axis height

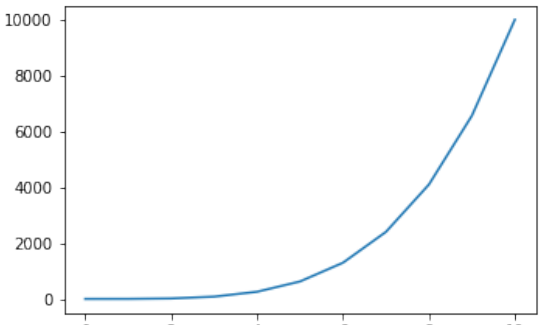
```
fig,axes = plt.subplots(nrows=2,ncols=2,figsize=(12,8))

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

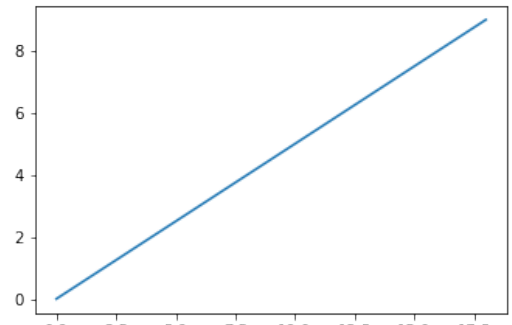
```
axes[1][1].plot(x,y)
axes[0][1].plot(y,x)
axes[1][0].plot(b,a)

# 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()
```

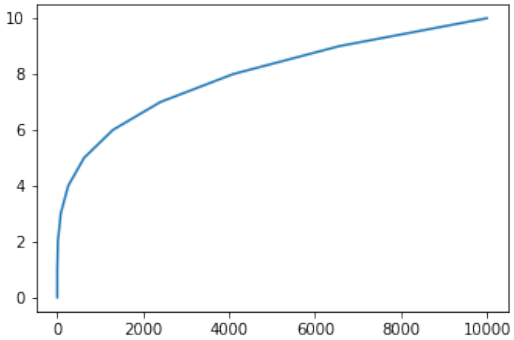


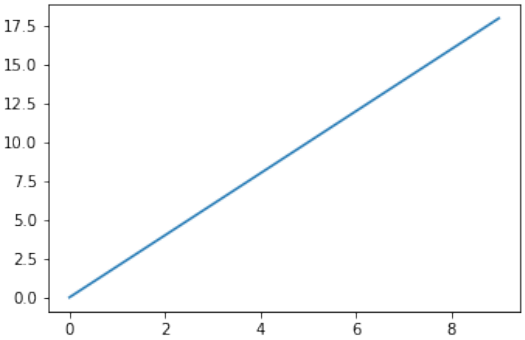
Exporting plt.subplots()

```
# 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()
```


10000
8000
6000
4000
2000
0
0
2
4
6
8
10


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


10
8
6
4
2
0
0
2000
4000
6000
8000
10000


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

—

Matplotlib Styling

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

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

import matplotlib.pyplot as plt
```

NOTE: For users running .py scripts in an IDE like PyCharm or Sublime Text Editor. You will not see the plots in a notebook, instead if you are using another editor, you'll use: *plt.show()* at the end of all your plotting commands to have the figure pop up in another window.

The Data

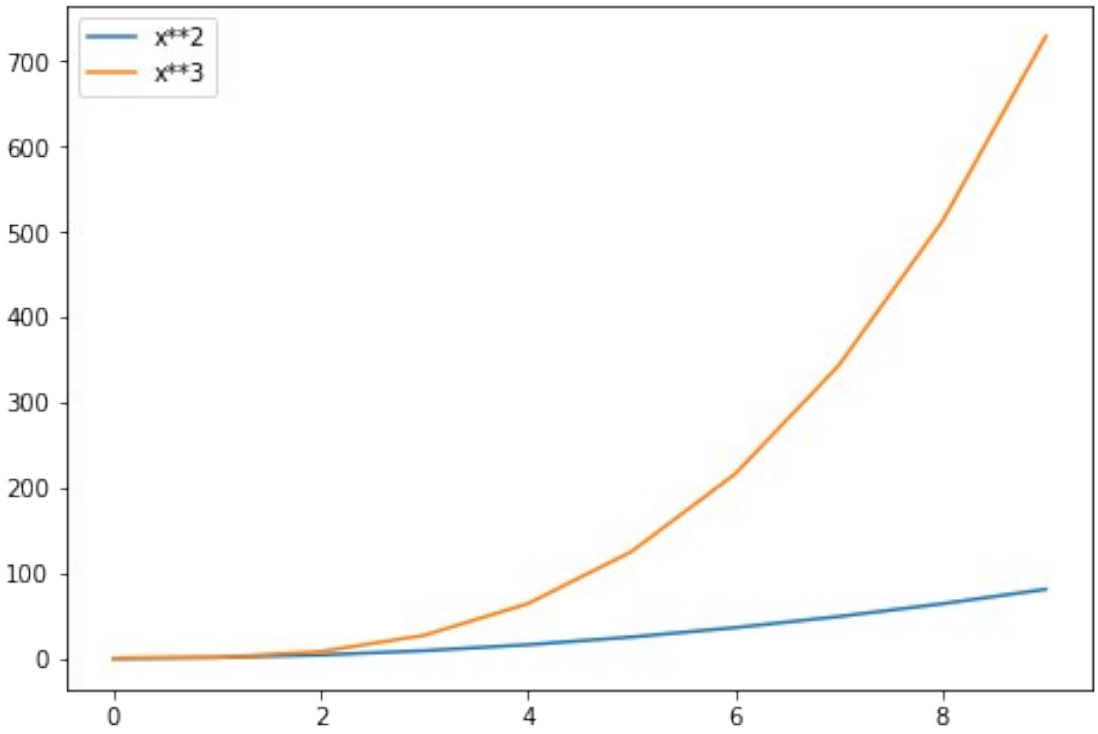
```
import numpy as np

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

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()
```

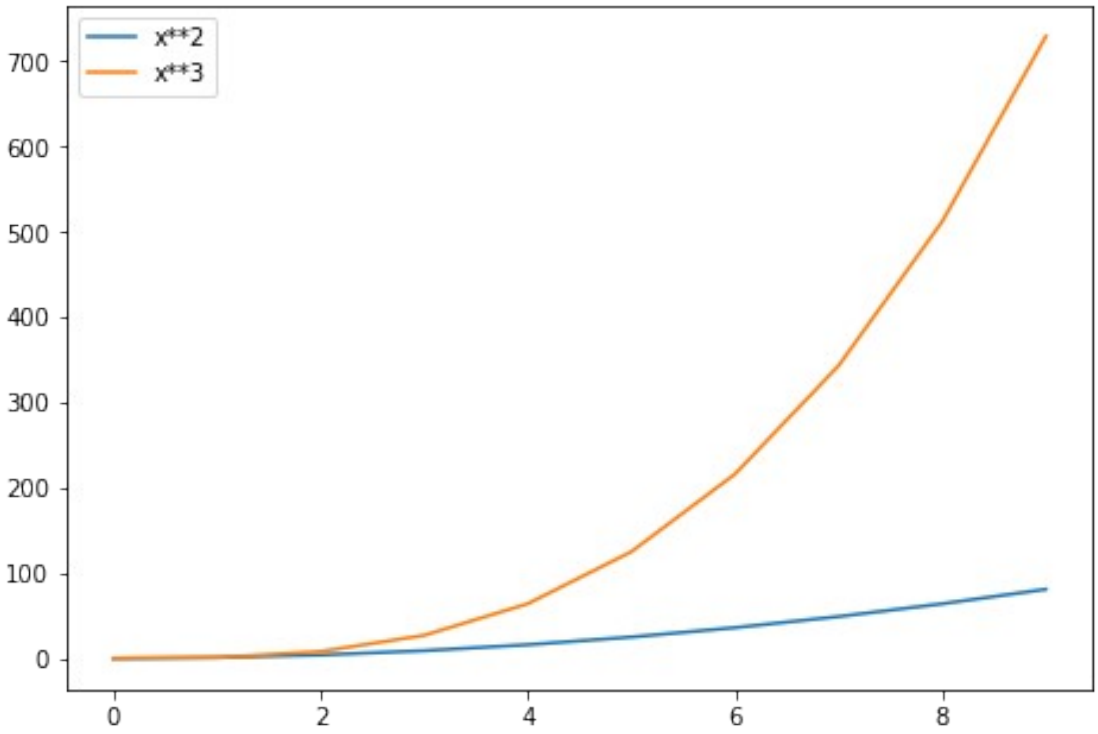
```
<matplotlib.legend.Legend at 0x151b84e0c08>
```



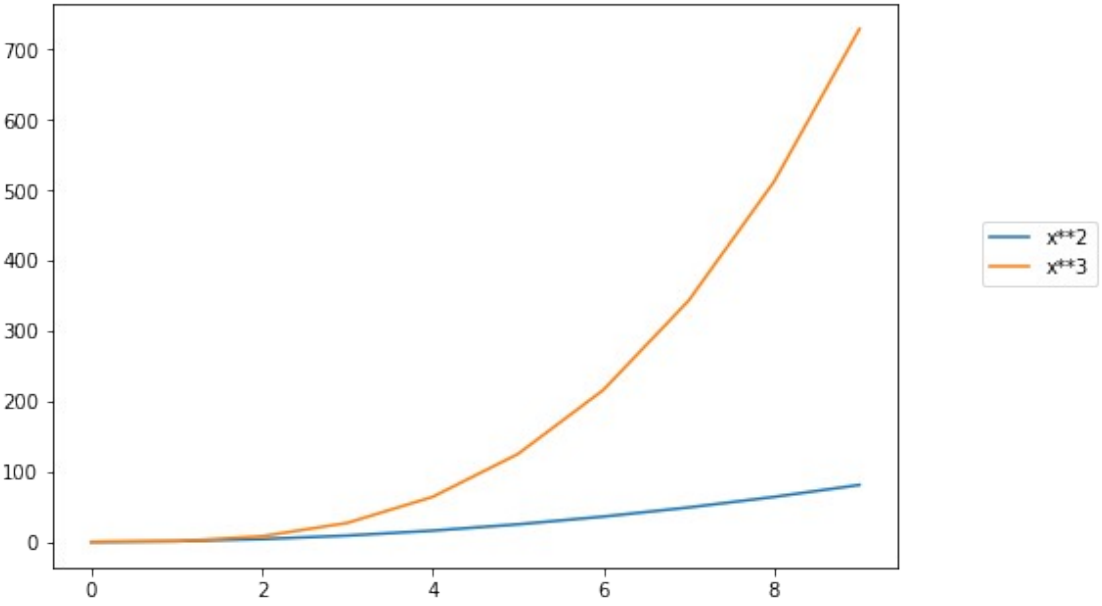
Notice how legend could potentially overlap some of the actual plot!

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
fig
```




```
ax.legend(loc=(1.1,0.5)) # manually set location
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.

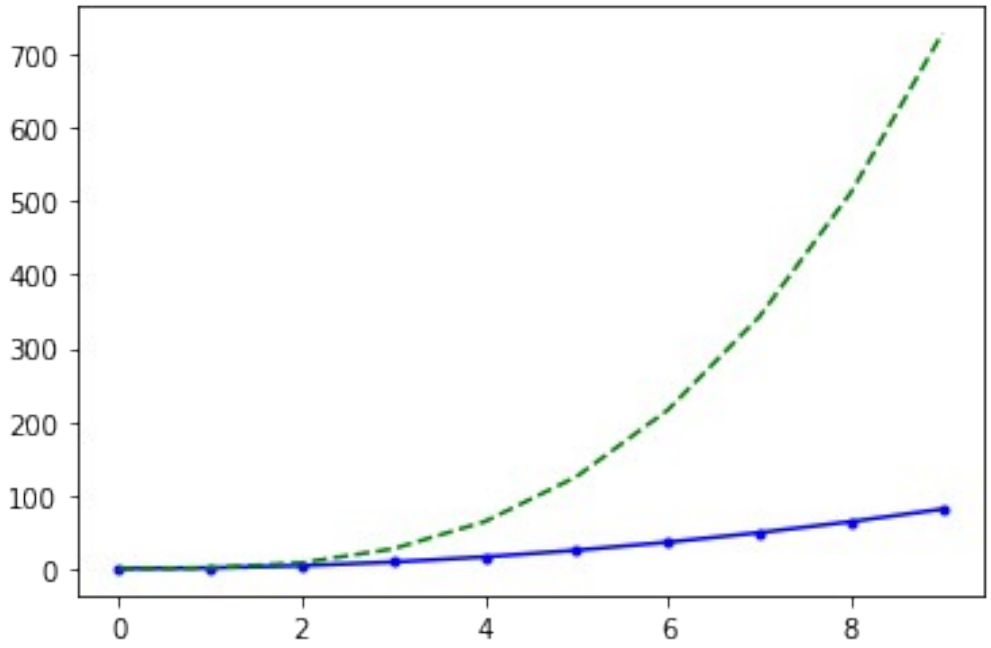
Quick View:

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:

```
# 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 0x151b8c263c8>]
```



Suggested Approach: Use keyword arguments

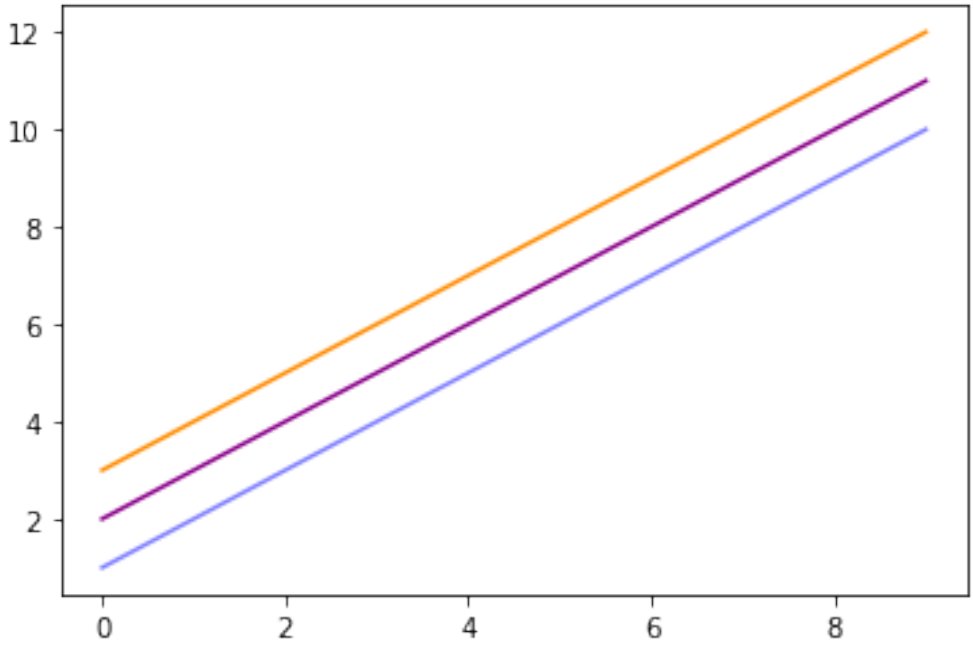
Colors with the color parameter

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 0x151b8c7fa08>]
```



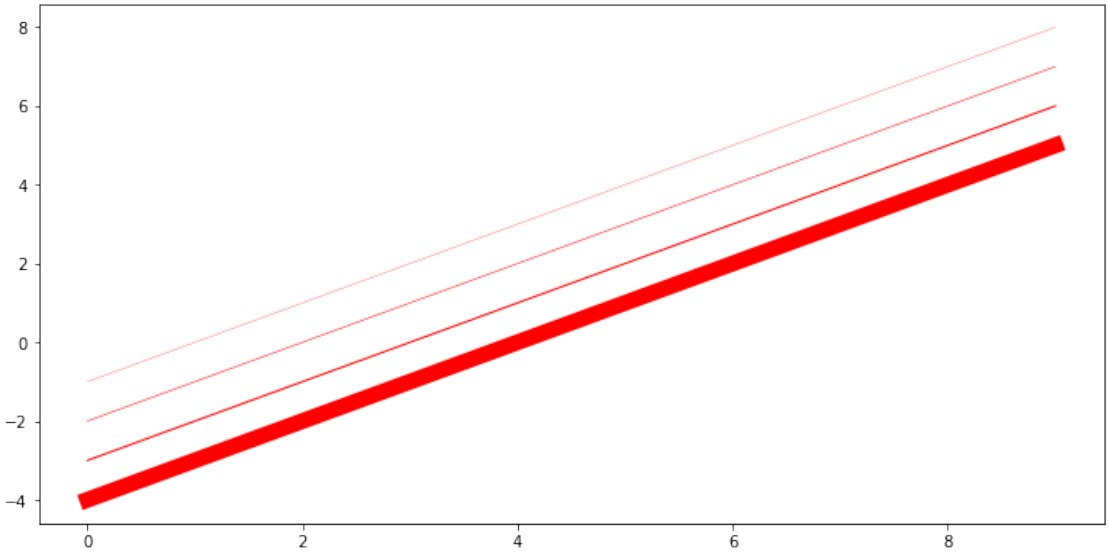
Line and marker styles

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 0x151b6dda608>]
```

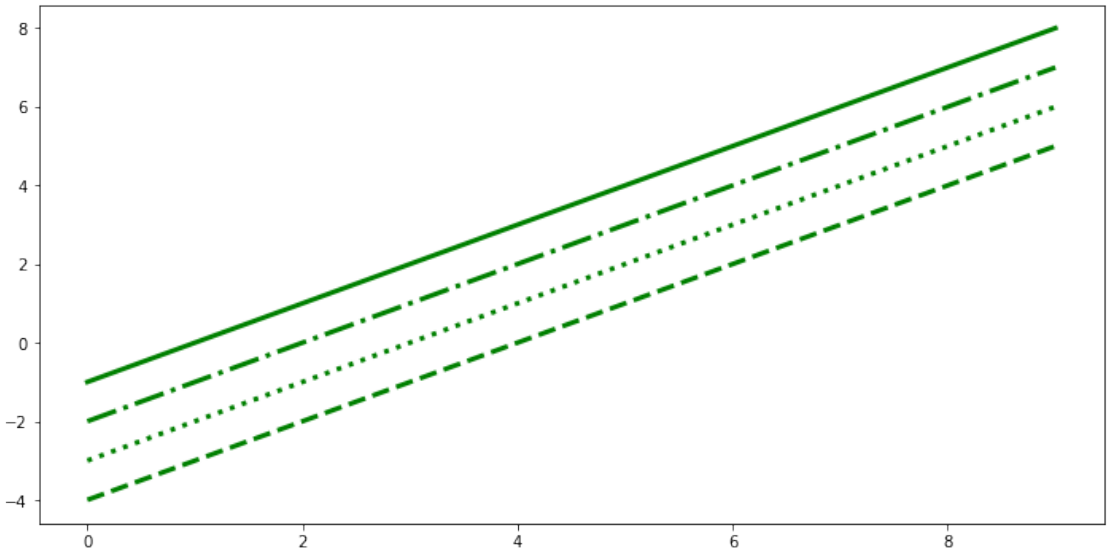


Linestyles

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 0x151b6df5d48>]
```



Custom linestyle dash

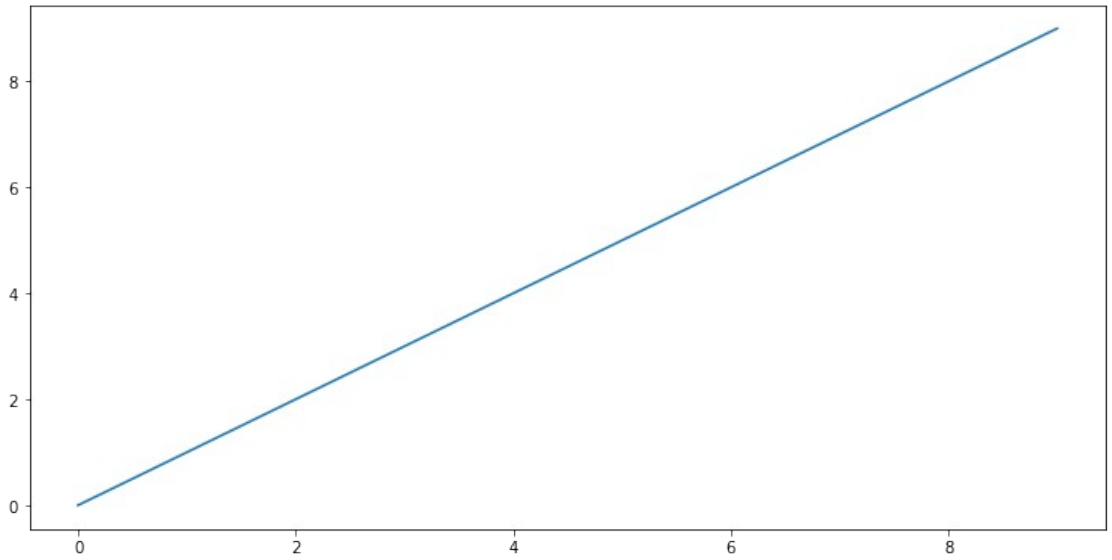
The dash sequence is a sequence of floats of even length describing the length of dashes and spaces in points.

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

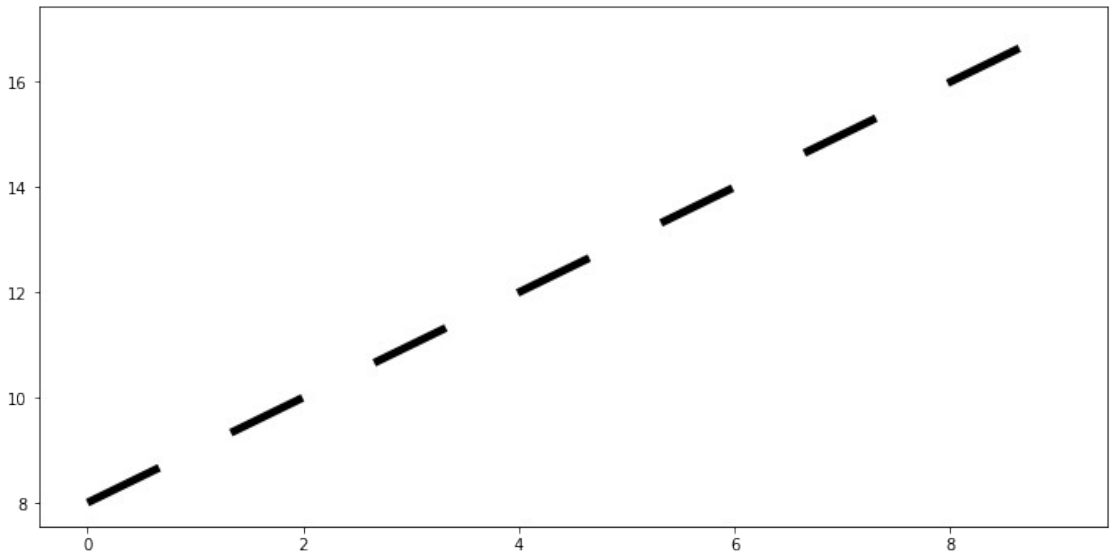
First, we see we can actually "grab" the line from the .plot() command

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

lines = ax.plot(x,x)

print(type(lines))
```

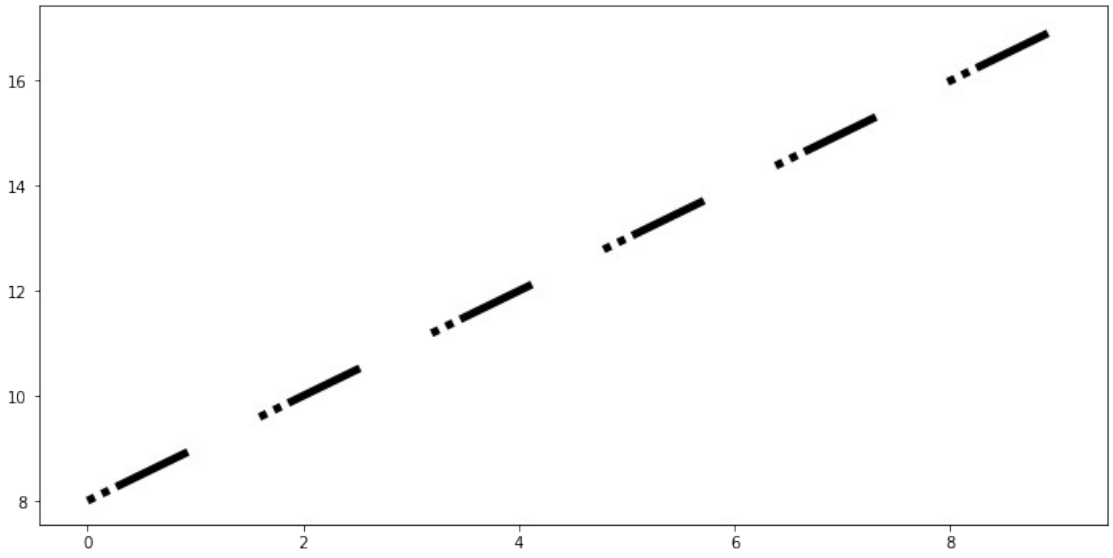
```
<class 'list'>
```




```
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: line length, 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
```



Markers

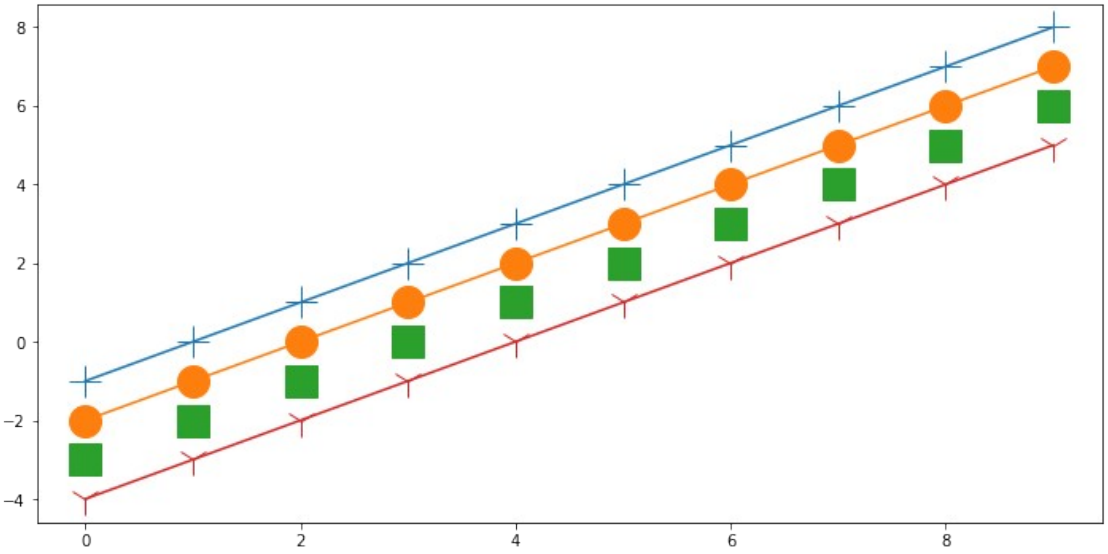
We've technically always been plotting points, and matplotlib has been automatically drawing a line between these points for us. Let's explore how to place markers at each of these points.

Markers Style

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

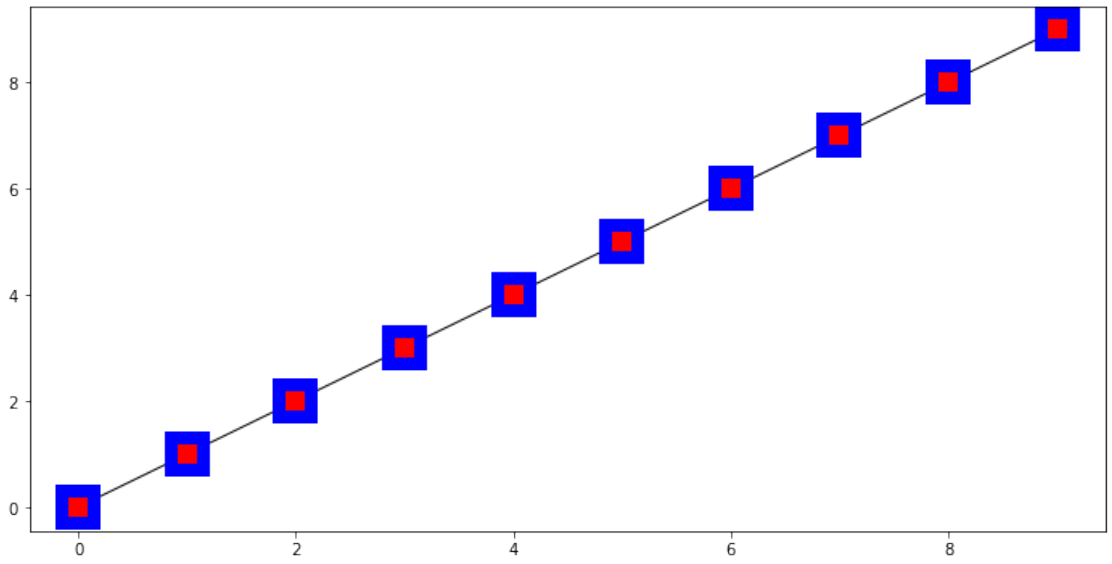
```
fig, ax = plt.subplots(figsize=(12,6))

# 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 0x151b89eed08>]
```



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.

—

Matplotlib Exercises

Welcome to the exercises for reviewing matplotlib! Take your time with these, Matplotlib can be tricky to understand at first. These are relatively simple plots, but they can be hard if this is your first time with matplotlib, feel free to reference the solutions as you go along.

Also don't worry if you find the matplotlib syntax frustrating, we actually won't be using it that often throughout the course, we will switch to using seaborn and pandas built-in visualization capabilities. But, those are built-off of matplotlib, which is why it is still important to get exposure to it!

NOTE: ALL THE COMMANDS FOR PLOTTING A FIGURE SHOULD ALL GO IN THE SAME CELL. SEPARATING THEM OUT INTO MULTIPLE CELLS MAY CAUSE NOTHING TO SHOW UP.

Exercises

We will focus on two commons tasks, plotting a known relationship from an equation and plotting raw data points.

Follow the instructions to complete the tasks to recreate the plots using this data:

Task One: Creating data from an equation

It is important to be able to directly translate a real equation into a plot. Your first task actually is pure numpy, then we will explore how to plot it out with Matplotlib. The world famous equation from Einstein:

$$E=mc^2$$

Use your knowledge of Numpy to create two arrays: E and m , where **m** is simply 11 evenly spaced values representing 0 grams to 10 grams. E should be the equivalent energy for the mass. You will need to figure out what to provide for **c** for the units m/s, a quick google search will easily give you the answer (we'll use the close approximation in our solutions).

NOTE: If this confuses you, then hop over to the solutions video for a guided walkthrough.

```
# CODE HERE
```

The array m should look like this:

```
[ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9. 10.]
```

```
# CODE HERE
```

The array E should look like this:

```
 [0.0e+00 9.0e+16 1.8e+17 2.7e+17 3.6e+17 4.5e+17 5.4e+17 6.3e+17
7.2e+17
 8.1e+17 9.0e+17]
```

Part Two: Plotting E=mc^2

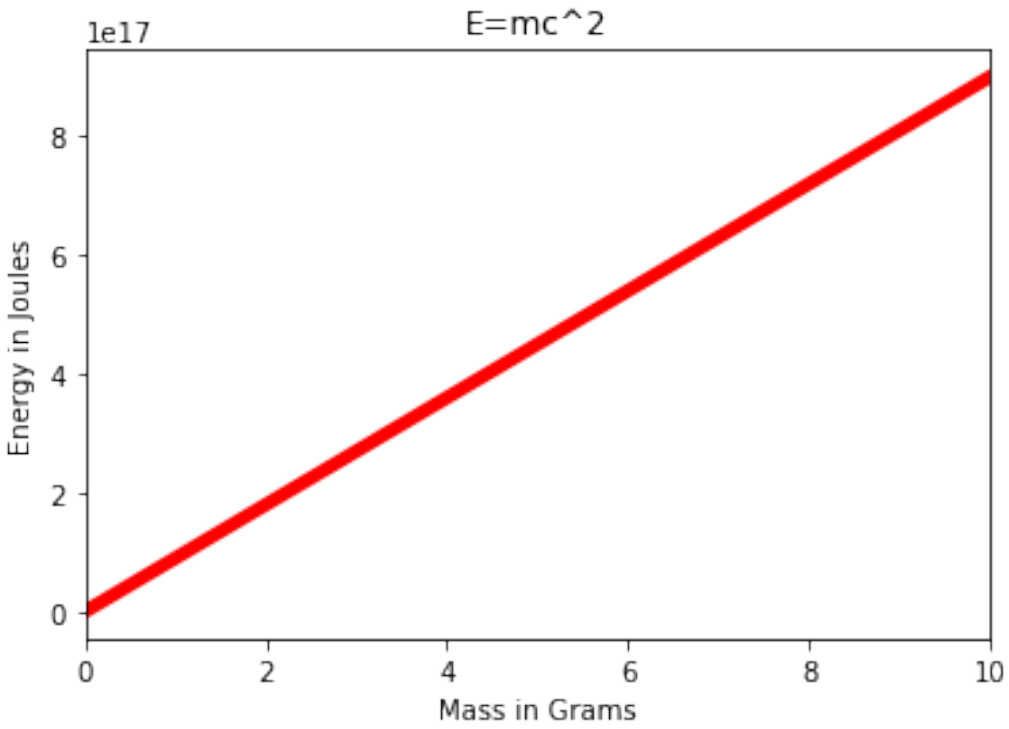
Now that we have the arrays E and m, we can plot this to see the relationship between Energy and Mass.

TASK: Import what you need from Matplotlib to plot out graphs:

TASK: Recreate the plot shown below which maps out E=mc^2 using the arrays we created in the previous task. Note the labels, titles, color, and axis limits. You don't need to match perfectly, but you should attempt to re-create each major component.

```
# CODE HERE
```

```
# DON'T RUN THE CELL BELOW< THAT WILL ERASE THE PLOT!
```

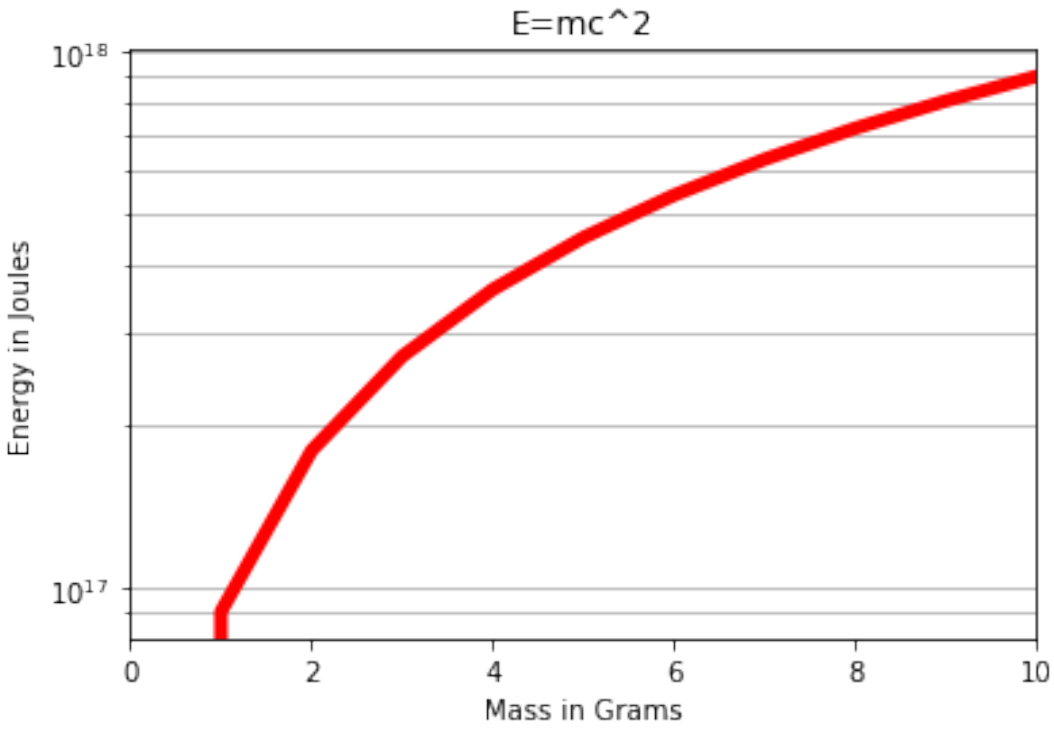


Part Three (BONUS)

Can you figure out how to plot this on a logarthimic scale on the y axis? Place a grid along the y axis ticks as well. We didn't show this in the videos, but you should be able to figure this out by referencing Google, StackOverflow, Matplotlib Docs, or even our "Additional Matplotlib Commands" notebook. The plot we show here only required two more lines of code for the changes.

```
# CODE HERE
```

```
# DONT RUN THE CELL BELOW! THAT WILL ERASE THE PLOT!
```



Task Two: Creating plots from data points

In finance, the yield curve is a curve showing several yields to maturity or interest rates across different contract lengths (2 month, 2 year, 20 year, etc. ...) for a similar debt contract. The curve shows the relation between the (level of the) interest rate (or cost of borrowing) and the time to maturity, known as the "term", of the debt for a given borrower in a given currency.

The U.S. dollar interest rates paid on U.S. Treasury securities for various maturities are closely watched by many traders, and are commonly plotted on a graph such as the one on the right, which is informally called "the yield curve".

For this exercise, we will give you the data for the yield curves at two separate points in time. Then we will ask you to create some plots from this data.

Part One: Yield Curve Data

We've obtained some yeild curve data for you from the US Treasury Dept.. The data shows the interest paid for a US Treasury bond for a certain contract length. The labels list shows the corresponding contract length per index position.

TASK: Run the cell below to create the lists for plotting.

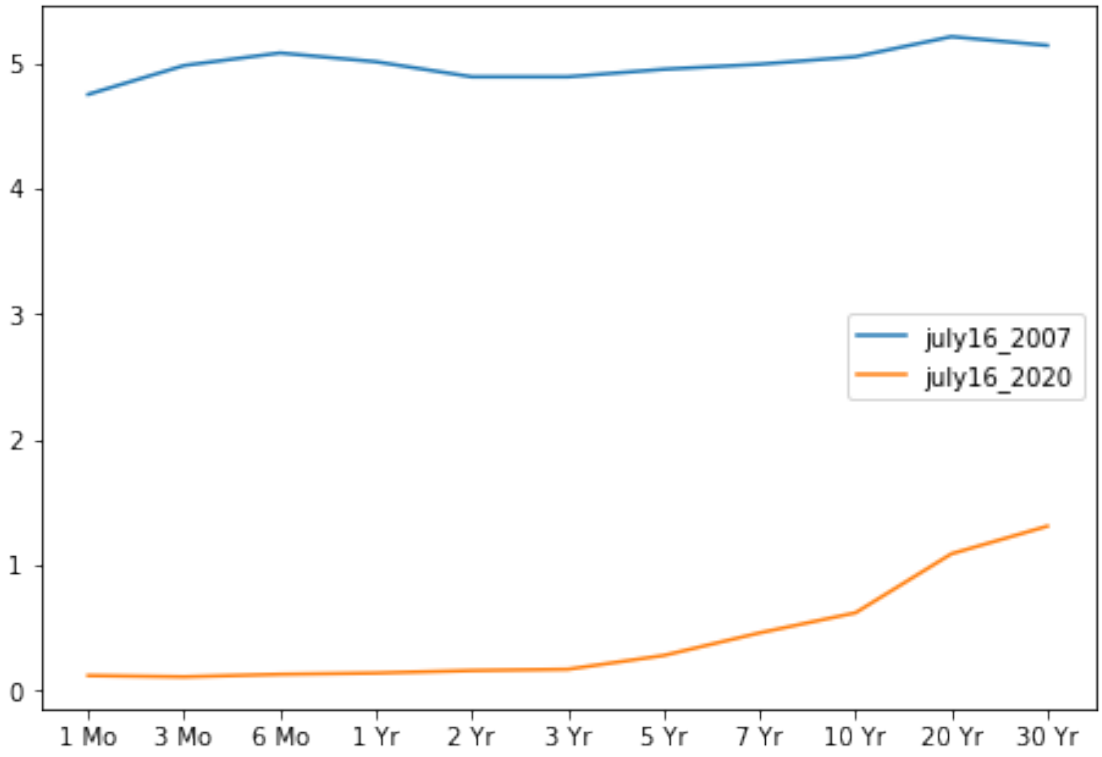
```
labels = ['1 Mo','3 Mo','6 Mo','1 Yr','2 Yr','3 Yr','5 Yr','7 Yr','10 Yr','20 Yr','30 Yr']
```

```
july16_2007 =[4.75,4.98,5.08,5.01,4.89,4.89,4.95,4.99,5.05,5.21,5.14]
july16_2020 = [0.12,0.11,0.13,0.14,0.16,0.17,0.28,0.46,0.62,1.09,1.31]
```

TASK: Figure out how to plot both curves on the same Figure. Add a legend to show which curve corresponds to a certain year.

```
# CODE HERE
```

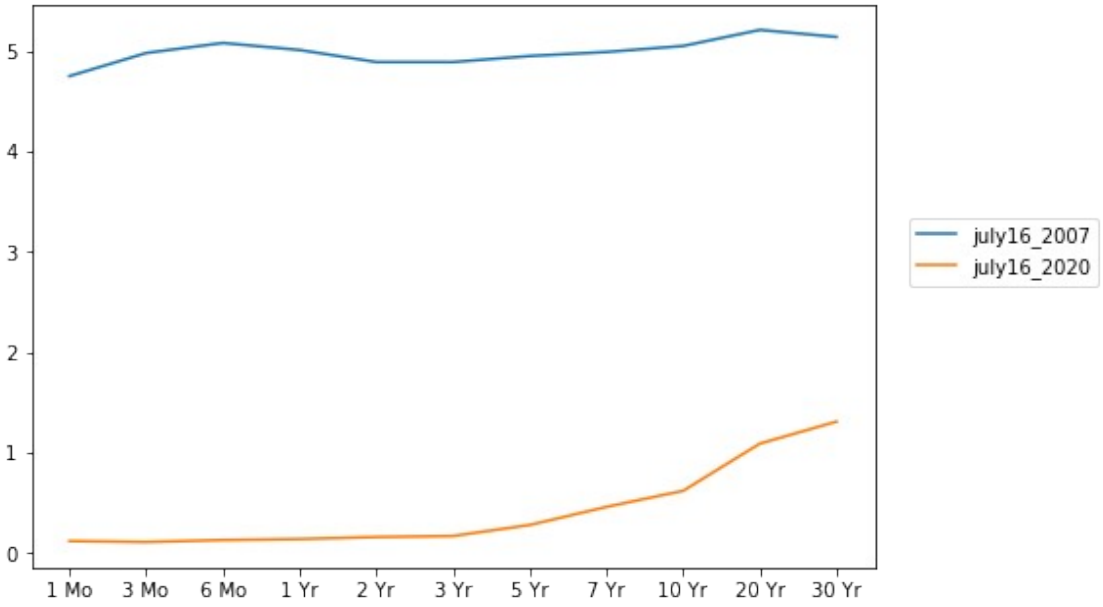
```
# DONT RUN THE CELL BELOW! IT WILL ERASE THE PLOT!
```



TASK: The legend in the plot above looks a little strange in the middle of the curves. While it is not blocking anything, it would be nicer if it were *outside* the plot. Figure out how to move the legend outside the main Figure plot.

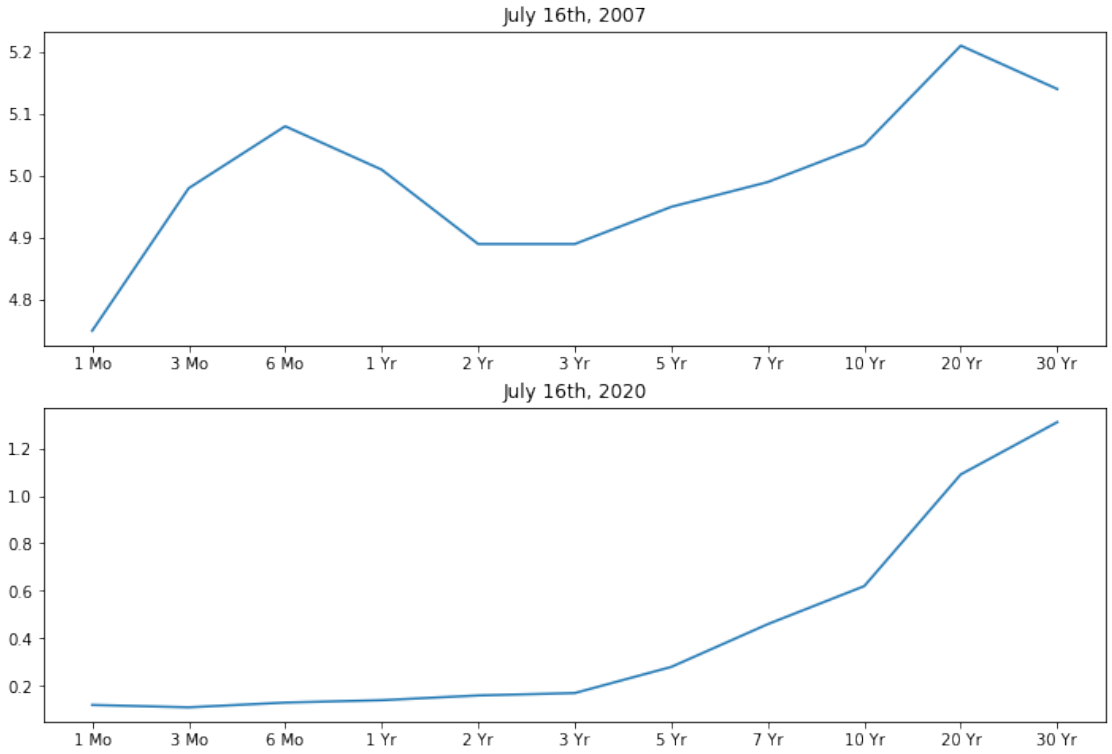
```
# CODE HERE
```

```
# DONT RUN THE CELL BELOW! IT WILL ERASE THE PLOT!
```



TASK: While the plot above clearly shows how rates fell from 2007 to 2020, putting these on the same plot makes it difficult to discern the rate differences within the same year. Use .suplots() to create the plot figure below, which shows each year's yield curve.

```
# CODE HERE

# DONT RUN THE CELL BELOW! IT WILL ERASE THE PLOT!
```

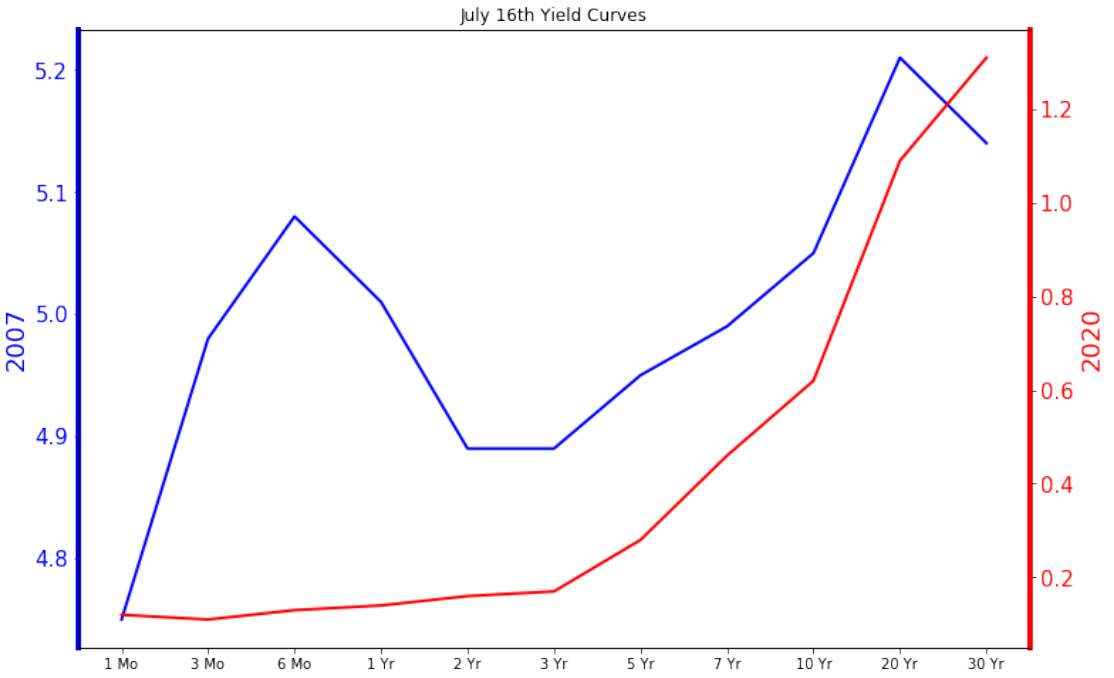


BONUS CHALLENGE TASK: Try to recreate the plot below that uses twin axes. While this plot may actually be more confusing than helpful, its a good exercise in Matplotlib control.

```
# CODE HERE
```

```
# DONT RUN THE CELL BELOW! IT ERASES THE PLOT!
```

```
(array([0. , 0.2, 0.4, 0.6, 0.8, 1. , 1.2, 1.4]),
 <a list of 8 Text yticklabel objects>)
```


July 16th Yield Curves
2007
2020
5.2
5.1
5.0
4.9
4.8
1.2
1.0
0.8
0.6
0.4
0.2
1 Mo
3 Mo
6 Mo
1 Yr
2 Yr
3 Yr
5 Yr
7 Yr
10 Yr
20 Yr
30 Yr

—

Matplotlib Exercises - Solutions

Welcome to the exercises for reviewing matplotlib! Take your time with these, Matplotlib can be tricky to understand at first. These are relatively simple plots, but they can be hard if this is your first time with matplotlib, feel free to reference the solutions as you go along.

Also don't worry if you find the matplotlib syntax frustrating, we actually won't be using it that often throughout the course, we will switch to using seaborn and pandas built-in visualization capabilities. But, those are built-off of matplotlib, which is why it is still important to get exposure to it!

NOTE: ALL THE COMMANDS FOR PLOTTING A FIGURE SHOULD ALL GO IN THE SAME CELL. SEPARATING THEM OUT INTO MULTIPLE CELLS MAY CAUSE NOTHING TO SHOW UP.

Exercises

We will focus on two commons tasks, plotting a known relationship from an equation and plotting raw data points.

Follow the instructions to complete the tasks to recreate the plots using this data:

Task One: Creating data from an equation

It is important to be able to directly translate a real equation into a plot. Your first task actually is pure numpy, then we will explore how to plot it out with Matplotlib. The world famous equation from Einstein:

$$E=mc^2$$

Use your knowledge of Numpy to create two arrays: E and m , where **m** is simply 11 evenly spaced values representing 0 grams to 10 grams. E should be the equivalent energy for the mass. You will need to figure out what to provide for **c** for the units m/s, a quick google search will easily give you the answer (we'll use the close approximation in our solutions).

NOTE: If this confuses you, then hop over to the solutions video for a guided walkthrough.

```
import numpy as np
```

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

print(f"The array m should look like this: \n\n{m}")
```

```
The array m should look like this:

[ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9. 10.]
```

```
c = 3 * 10**8 # Speed of Light

E = m*c**2

print(f"The array E should look like this: \n\n {E}")
```

```
The array E should look like this:

 [0.0e+00 9.0e+16 1.8e+17 2.7e+17 3.6e+17 4.5e+17 5.4e+17 6.3e+17
7.2e+17
 8.1e+17 9.0e+17]
```

Part Two: Plotting E=mc^2

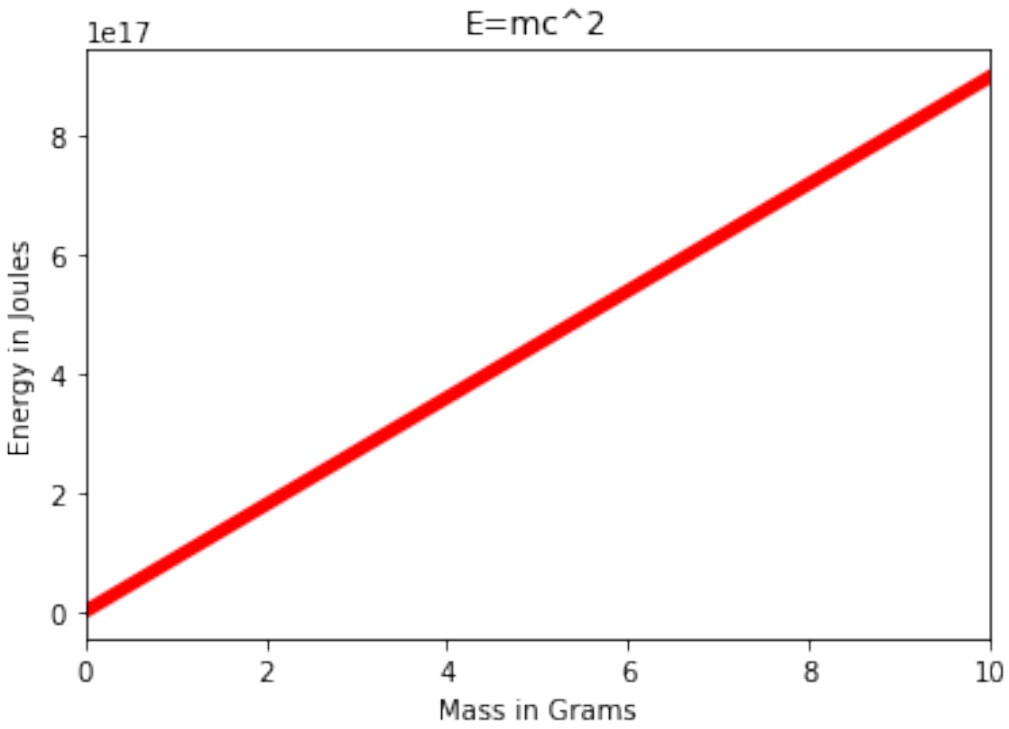
Now that we have the arrays E and m, we can plot this to see the relationship between Energy and Mass.

TASK: Import what you need from Matplotlib to plot out graphs:

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

TASK: Recreate the plot shown below which maps out E=mc^2 using the arrays we created in the previous task. Note the labels, titles, color, and axis limits. You don't need to match perfectly, but you should attempt to re-create each major component.

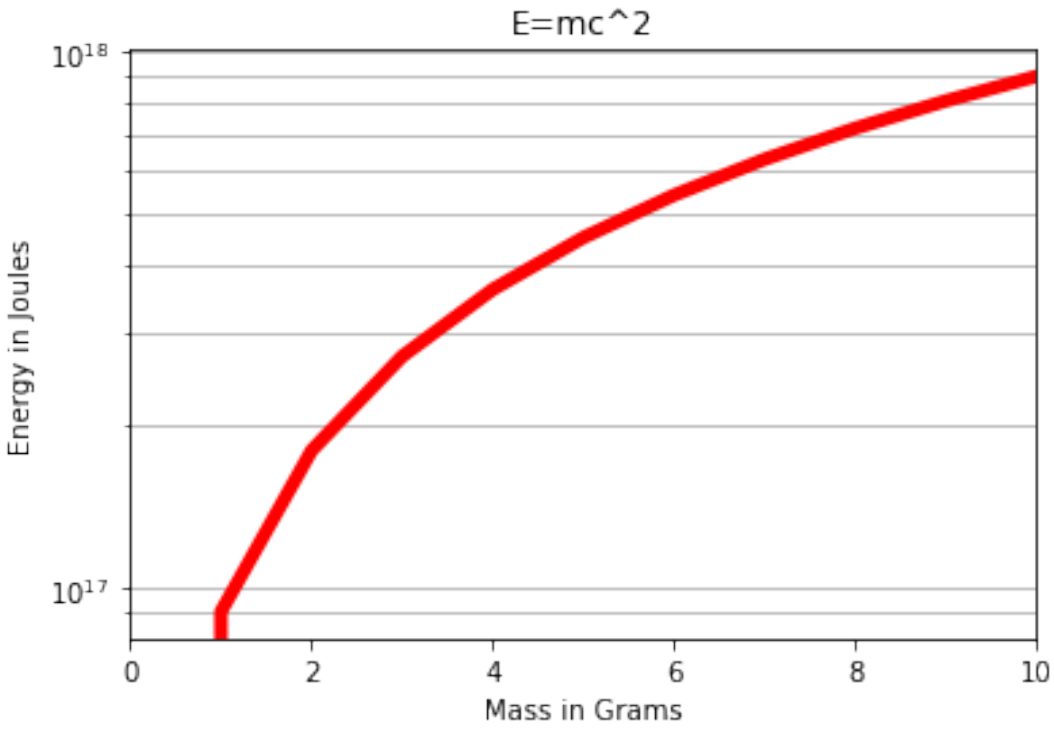
```
# CODE HERE

# DON"T RUN THE CELL BELOW< THAT WILL ERASE THE PLOT!

plt.plot(m,E,color='red',lw=5)
plt.title("E=mc^2")
plt.xlabel("Mass in Grams")
plt.ylabel("Energy in Joules")
plt.xlim(0,10)
plt.show()
```



Part Three (BONUS)

Can you figure out how to plot this on a logarthimic scale on the y axis? Place a grid along the y axis ticks as well. We didn't show this in the videos, but you should be able to figure this out by referencing Google, StackOverflow, Matplotlib Docs, or even our "Additional Matplotlib Commands" notebook. The plot we show here only required two more lines of code for the changes.

```
# CODE HERE

# DONT RUN THE CELL BELOW! THAT WILL ERASE THE PLOT!

plt.plot(m,E,color='red',lw=5)
plt.title("E=mc^2")
plt.xlabel("Mass in Grams")
plt.ylabel("Energy in Joules")
plt.xlim(0,10)

# LOG SCALE
plt.yscale("log")
plt.grid(which='both',axis='y')

plt.show()
```



Task Two: Creating plots from data points

In finance, the yield curve is a curve showing several yields to maturity or interest rates across different contract lengths (2 month, 2 year, 20 year, etc. ...) for a similar debt contract. The curve shows the relation between the (level of the) interest rate (or cost of borrowing) and the time to maturity, known as the "term", of the debt for a given borrower in a given currency.

The U.S. dollar interest rates paid on U.S. Treasury securities for various maturities are closely watched by many traders, and are commonly plotted on a graph such as the one on the right, which is informally called "the yield curve".

For this exercise, we will give you the data for the yield curves at two separate points in time. Then we will ask you to create some plots from this data.

Part One: Yield Curve Data

We've obtained some yeild curve data for you from the US Treasury Dept.. The data shows the interest paid for a US Treasury bond for a certain contract length. The labels list shows the corresponding contract length per index position.

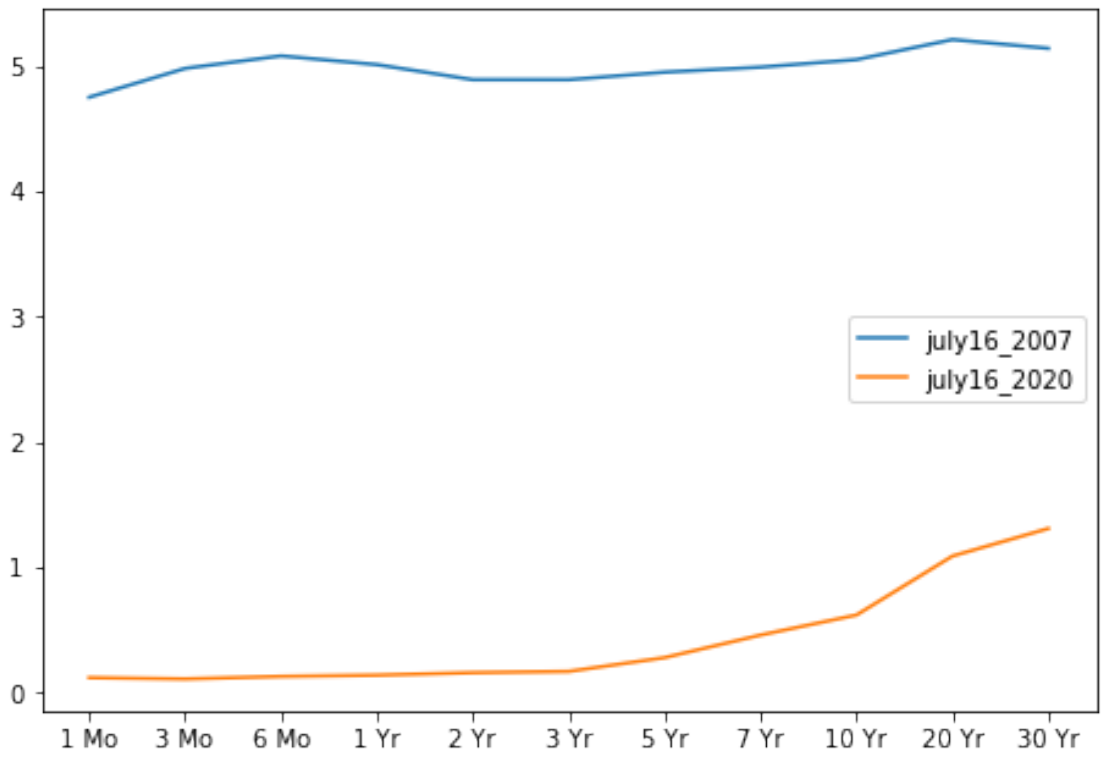
TASK: Run the cell below to create the lists for plotting.

```
labels = ['1 Mo','3 Mo','6 Mo','1 Yr','2 Yr','3 Yr','5 Yr','7 Yr','10 Yr','20 Yr','30 Yr']
```

```
july16_2007 =[4.75,4.98,5.08,5.01,4.89,4.89,4.95,4.99,5.05,5.21,5.14]
july16_2020 = [0.12,0.11,0.13,0.14,0.16,0.17,0.28,0.46,0.62,1.09,1.31]
```

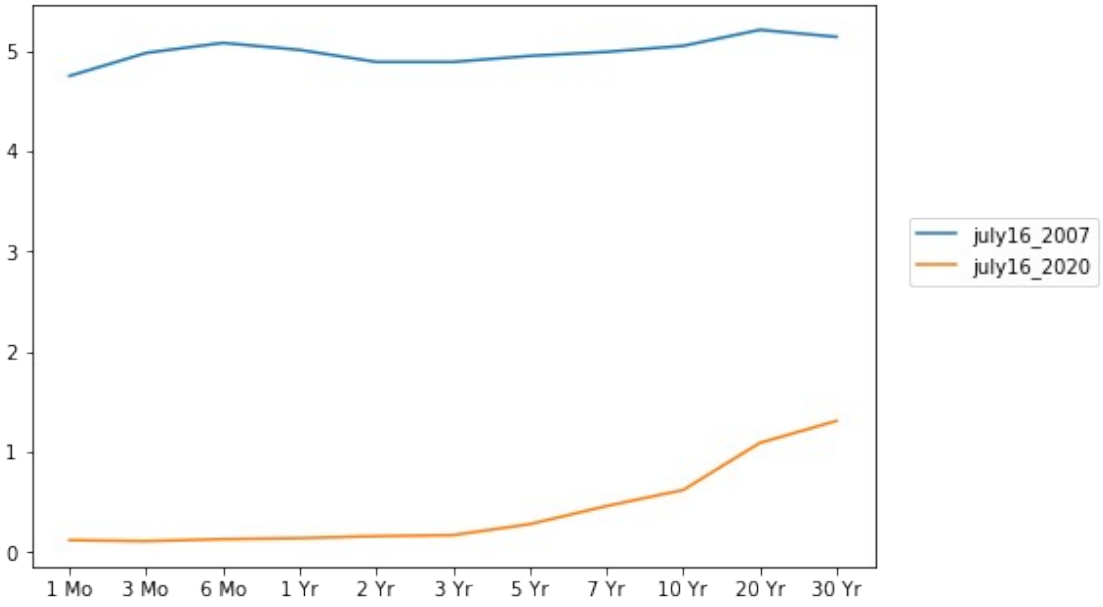
TASK: Figure out how to plot both curves on the same Figure. Add a legend to show which curve corresponds to a certain year.

```
# CODE HERE

# DONT RUN THE CELL BELOW! IT WILL ERASE THE PLOT!

# 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(labels, july16_2007,label='july16_2007')
axes.plot(labels,july16_2020,label='july16_2020')
plt.legend()

plt.show()
```



TASK: The legend in the plot above looks a little strange in the middle of the curves. While it is not blocking anything, it would be nicer if it were *outside* the plot. Figure out how to move the legend outside the main Figure plot.

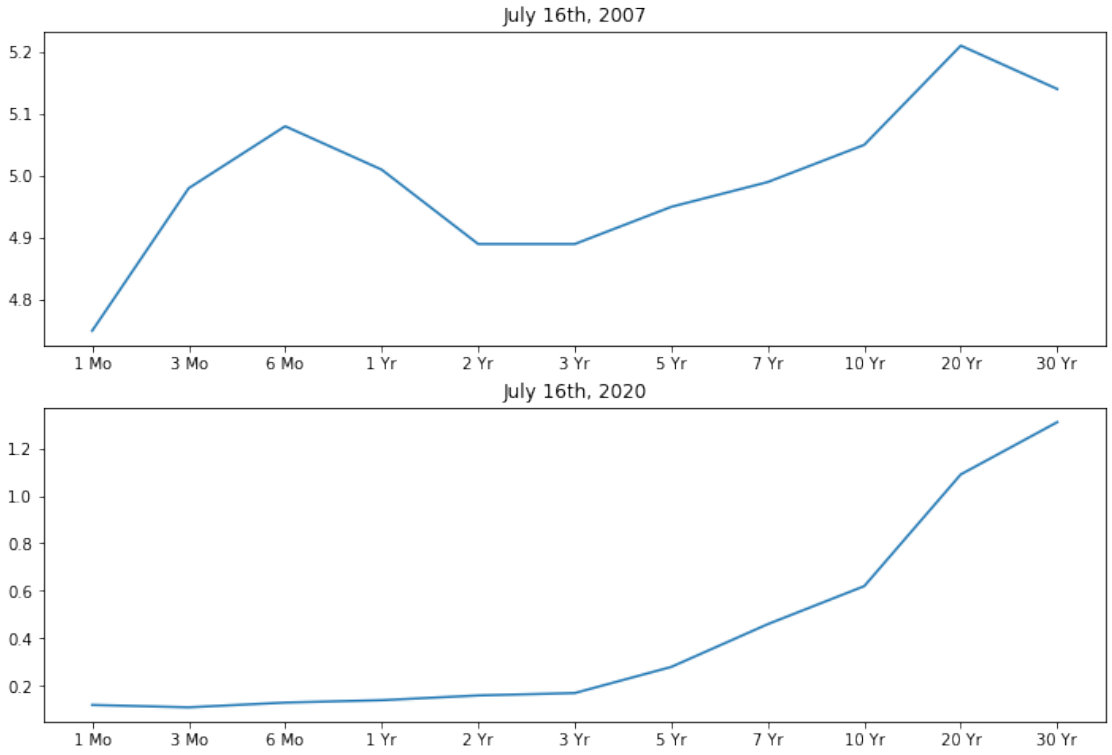
```
# CODE HERE

# DONT RUN THE CELL BELOW! IT WILL ERASE THE PLOT!

# 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(labels, july16_2007,label='july16_2007')
axes.plot(labels,july16_2020,label='july16_2020')
plt.legend(loc=(1.04,0.5))

plt.show()
```



TASK: While the plot above clearly shows how rates fell from 2007 to 2020, putting these on the same plot makes it difficult to discern the rate differences within the same year. Use .suplots() to create the plot figure below, which shows each year's yield curve.

```
# CODE HERE

# DONT RUN THE CELL BELOW! IT WILL ERASE THE PLOT!
```

```
fig,axes = plt.subplots(nrows=2,ncols=1,figsize=(12,8))

axes[0].plot(labels, july16_2007,label='july16_2007')
axes[0].set_title("July 16th, 2007")
axes[1].plot(labels,july16_2020,label='july16_2020')
axes[1].set_title("July 16th, 2020")


plt.show()
```

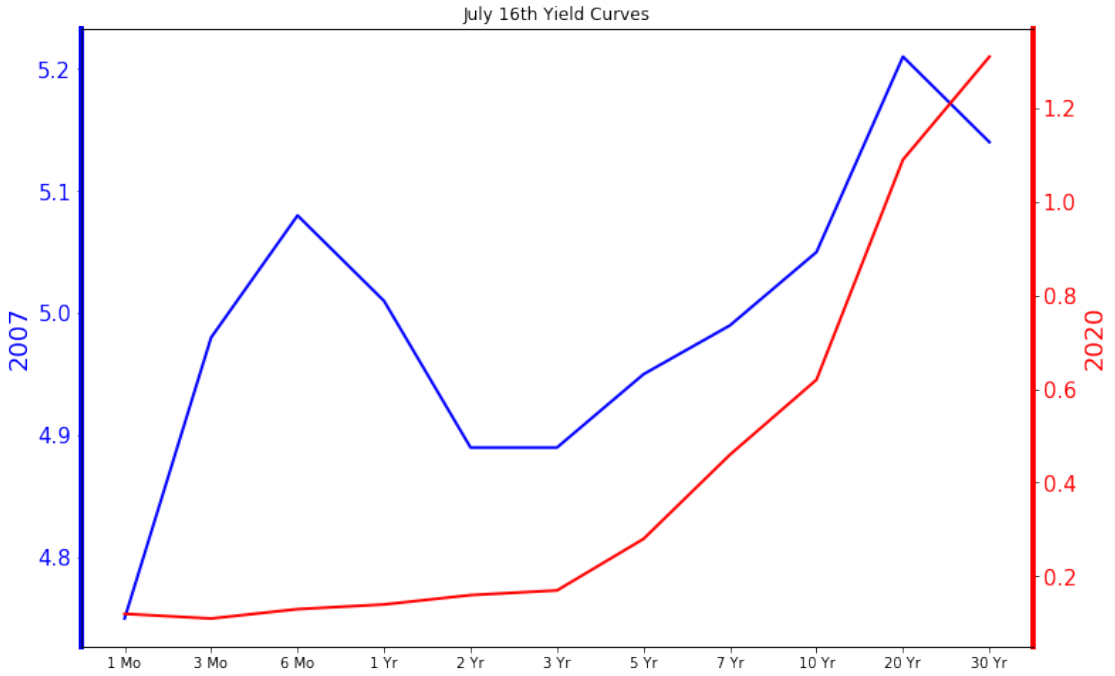


BONUS CHALLENGE TASK: Try to recreate the plot below that uses twin axes. While this plot may actually be more confusing than helpful, its a good exercise in Matplotlib control.

```
# CODE HERE

# DONT RUN THE CELL BELOW! IT ERASES THE PLOT!

fig, ax1 = plt.subplots(figsize=(12,8))

ax1.plot(labels,july16_2007, lw=2, color="blue")
ax1.set_ylabel("2007", fontsize=18, color="blue")

ax1.spines['left'].set_color('blue')
ax1.spines['left'].set_linewidth(4)

for label in ax1.get_yticklabels():
```

```
    label.set_color("blue")
plt.yticks(fontsize=15)

ax2 = ax1.twinx()
ax2.plot(labels,july16_2020, lw=2, color="red")
ax2.set_ylabel("2020", fontsize=18, color="red")

ax2.spines['right'].set_color('red')
ax2.spines['right'].set_linewidth(4)

for label in ax2.get_yticklabels():
    label.set_color("red")

ax1.set_title("July 16th Yield Curves");
plt.yticks(fontsize=15)
```

```
(array([0. , 0.2, 0.4, 0.6, 0.8, 1. , 1.2, 1.4]),
 <a list of 8 Text yticklabel objects>)
```



—

Advanced Matplotlib Commands Lecture

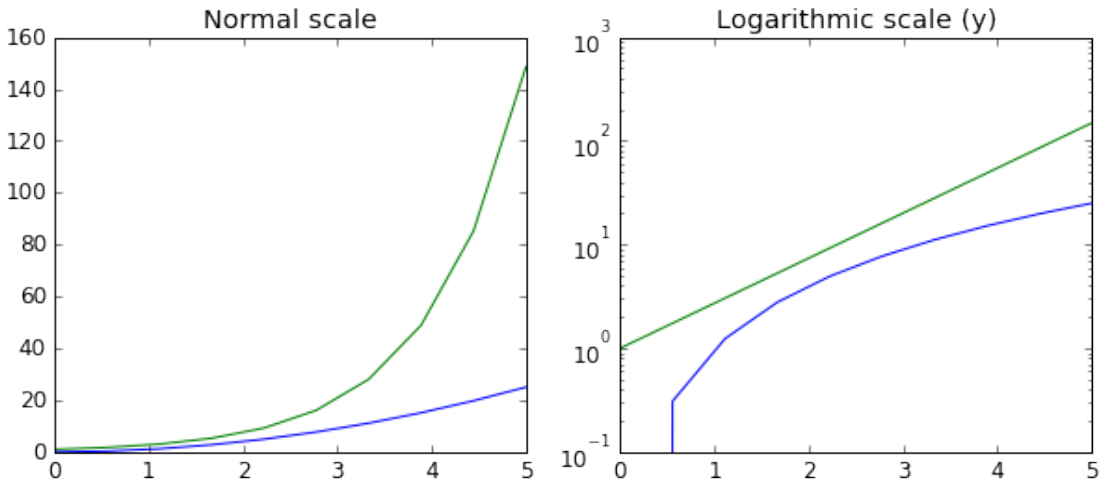
NOTE: There is no video for the notebook since its really just a reference for what method calls to look for. We also highly recommend doing a quick StackOverflow search if you're in need of a quick answer to what Matplotlib method to use for a particular case.

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)");
```

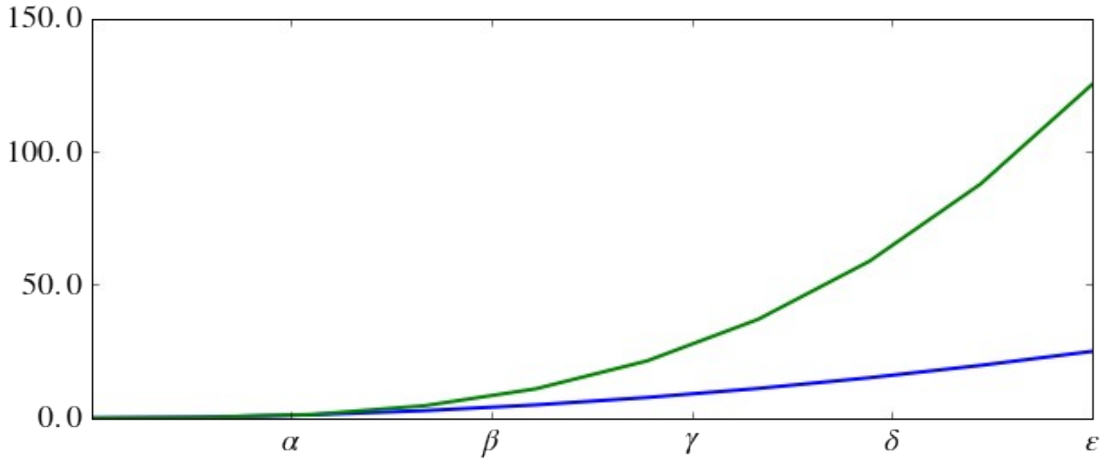


Placement of ticks and custom tick labels

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



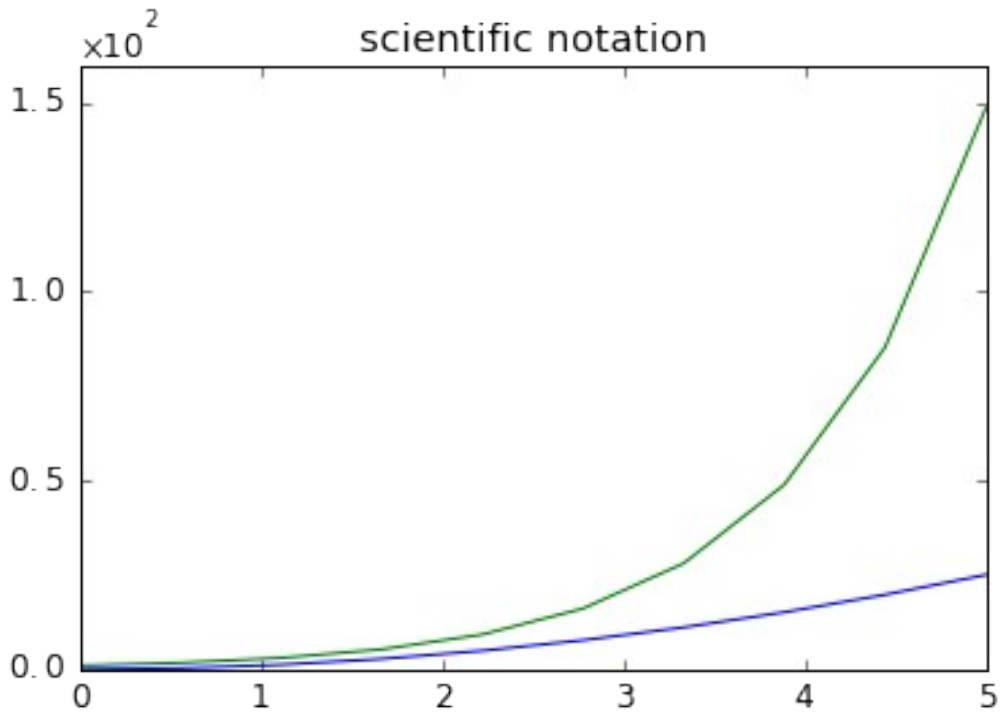
There are a number of more advanced methods for controlling major and minor tick placement in matplotlib figures, such as automatic placement according to different policies. See http://matplotlib.org/api/ticker_api.html for details.

Scientific notation

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

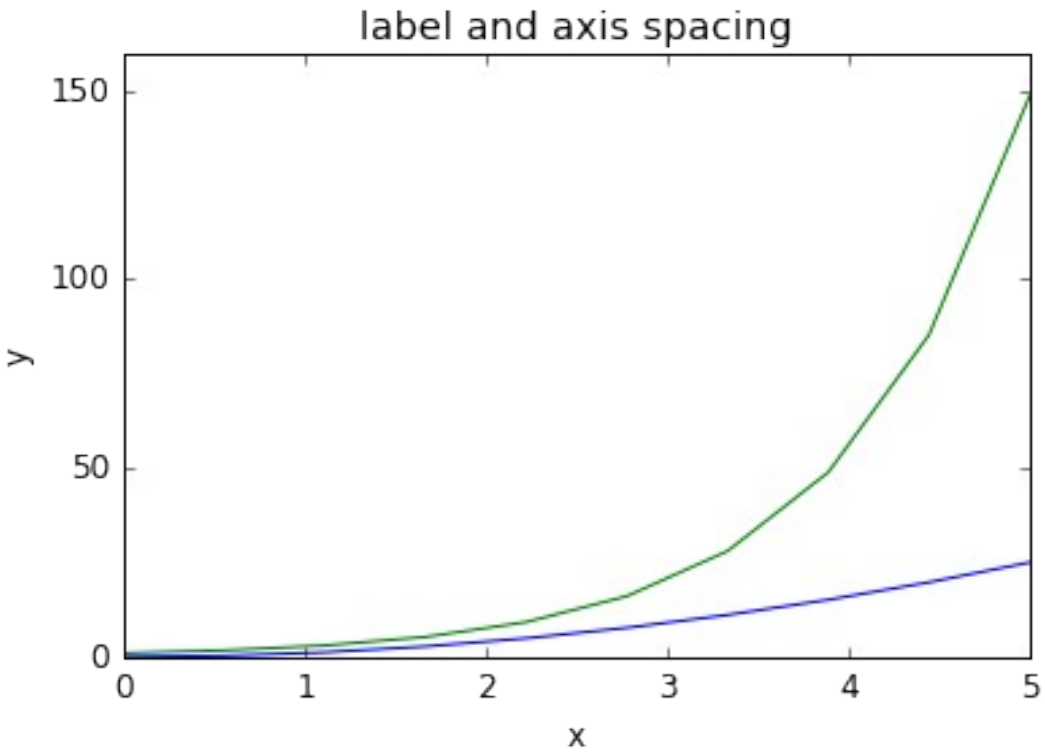
```
fig, ax = plt.subplots(1, 1)

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

```
# 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");
```

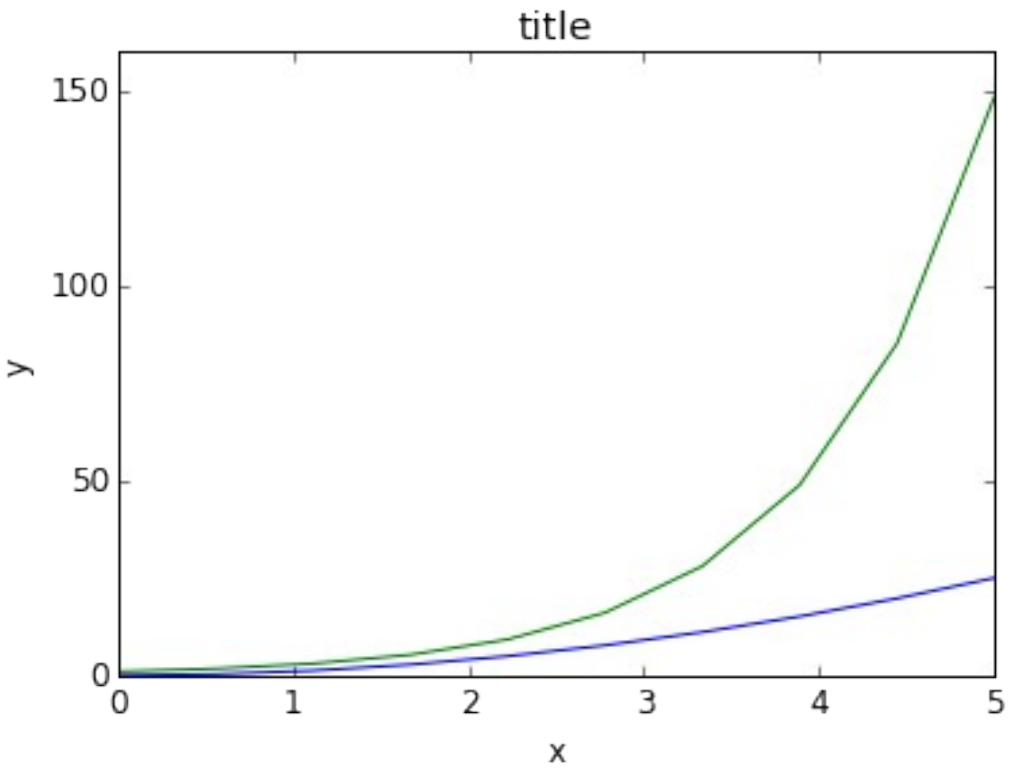



```
# restore defaults
matplotlib.rcParams['xtick.major.pad'] = 3
matplotlib.rcParams['ytick.major.pad'] = 3
```

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`:

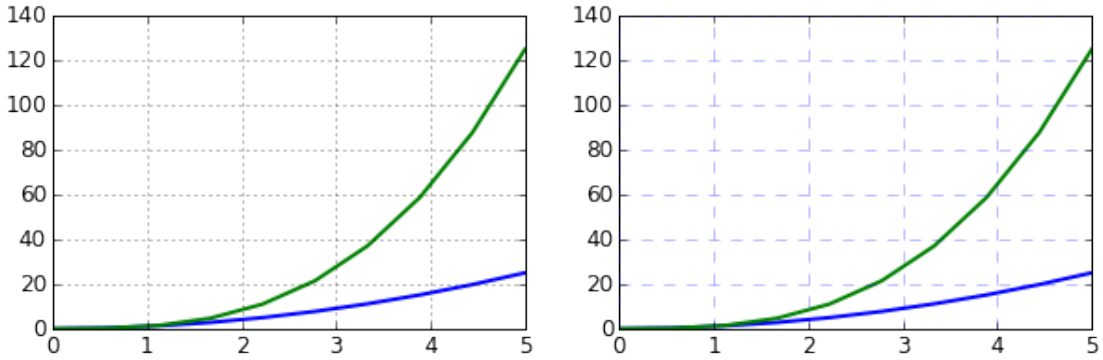
```
fig, ax = plt.subplots(1, 1)

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:

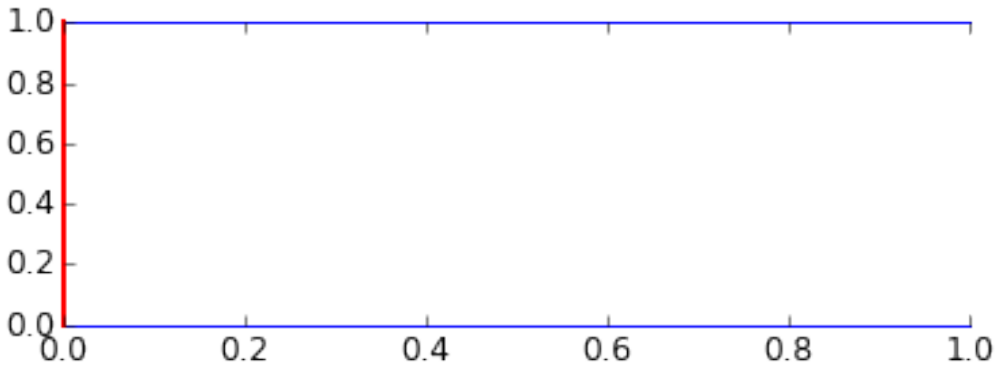
```
fig, axes = plt.subplots(1, 2, figsize=(10,3))

# 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
```

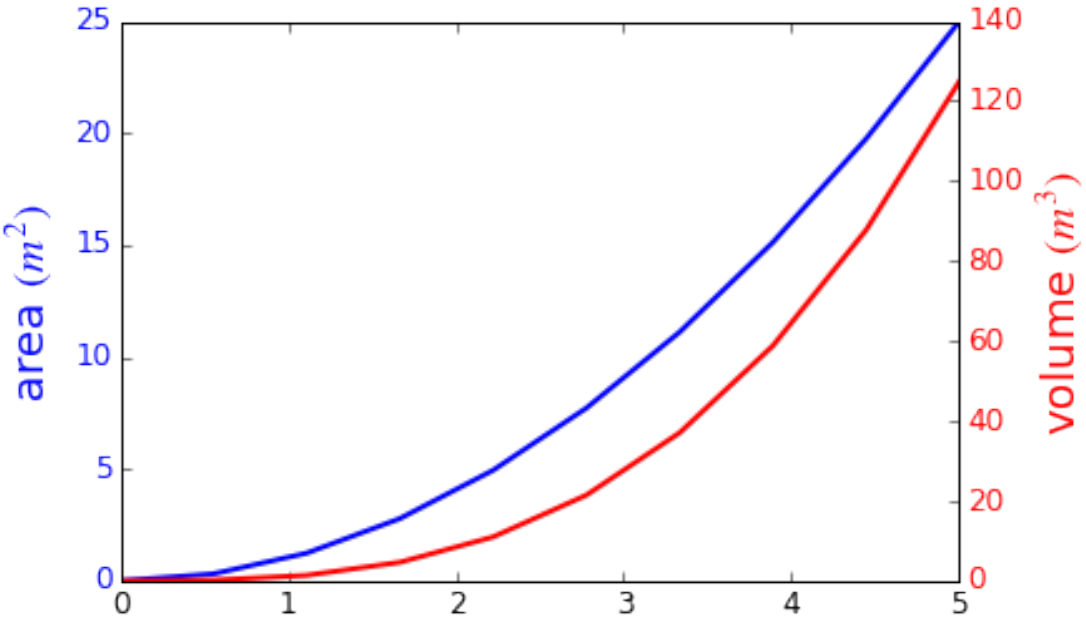


Twin axes

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:

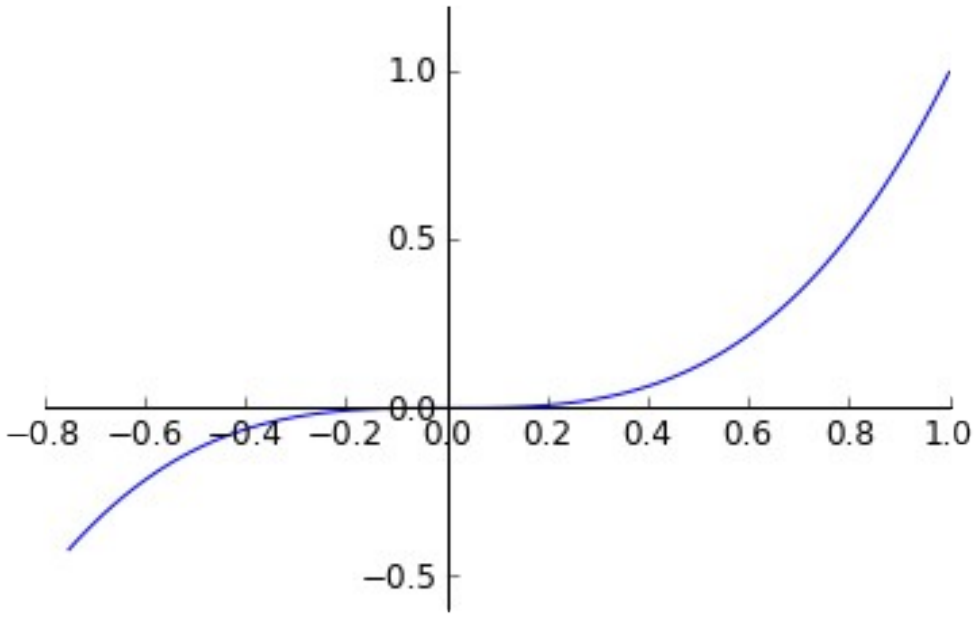
```
fig, ax1 = plt.subplots()

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

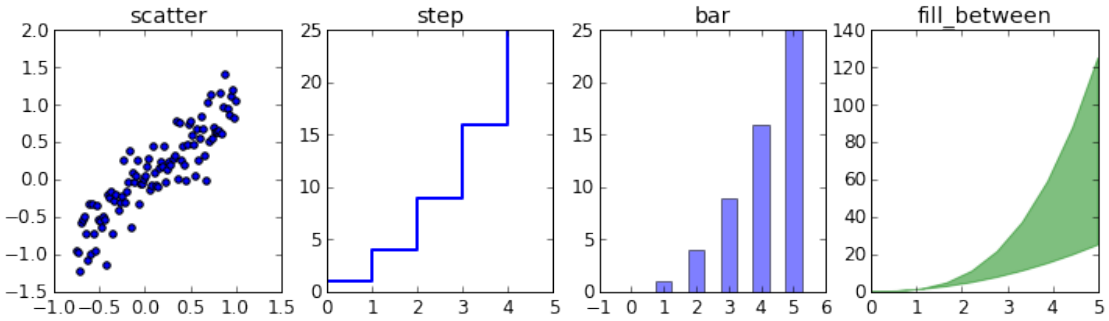
```
fig, ax = plt.subplots()

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:

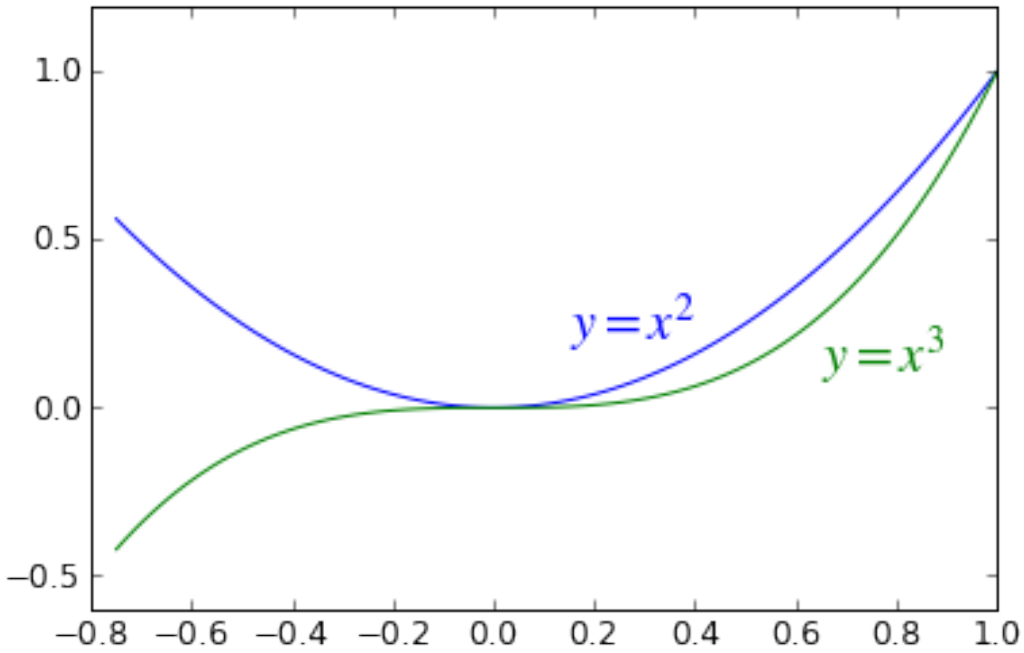
```
n = np.array([0,1,2,3,4,5])

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");
```

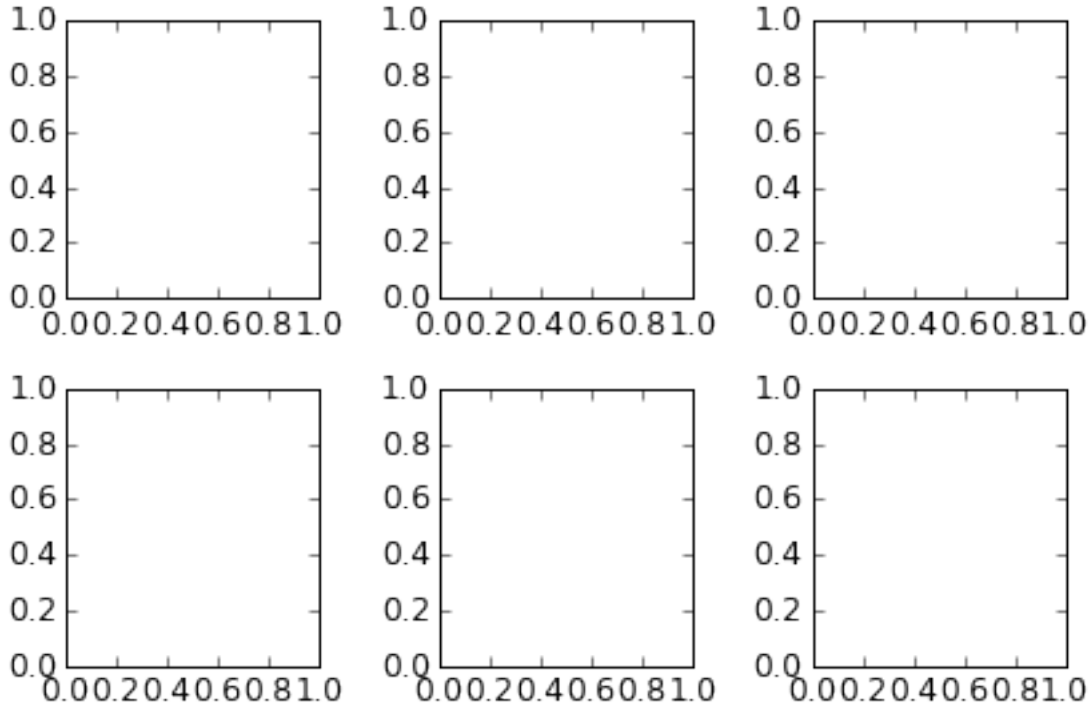


Figures with multiple subplots and insets

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`:

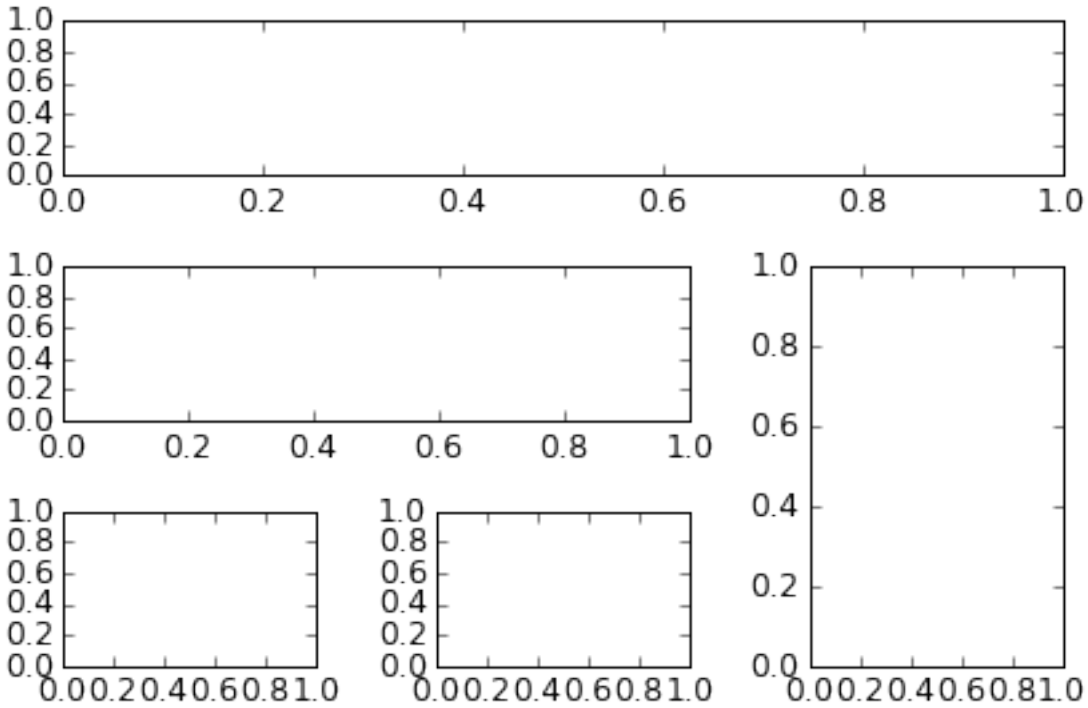
subplots

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



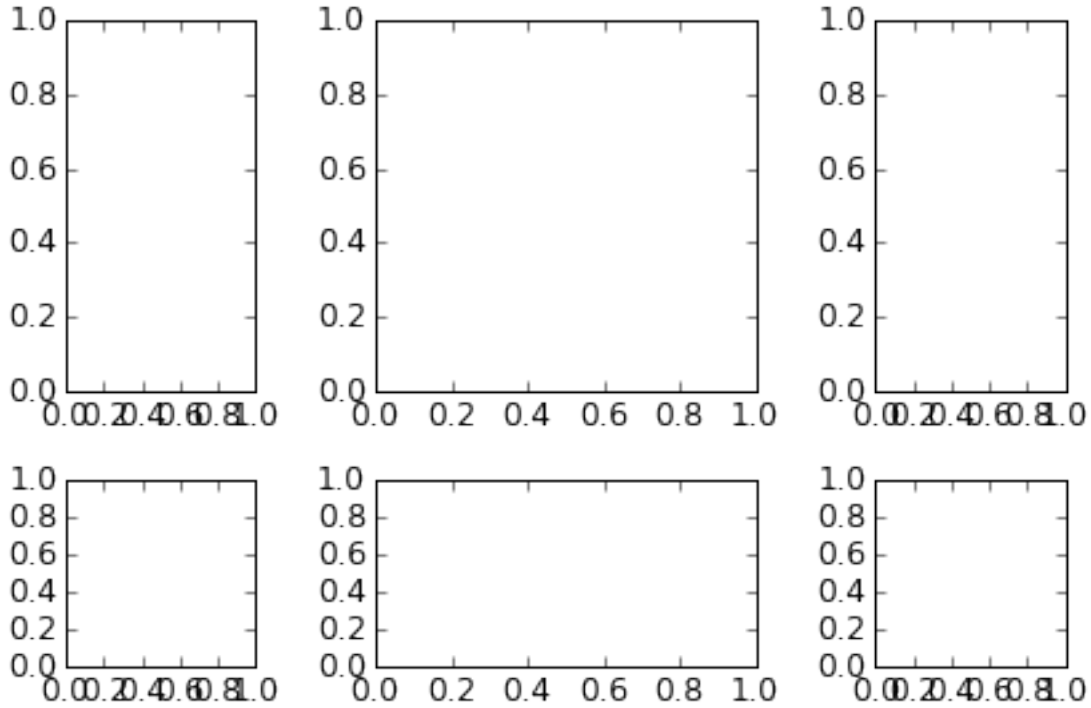
subplot2grid

```
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

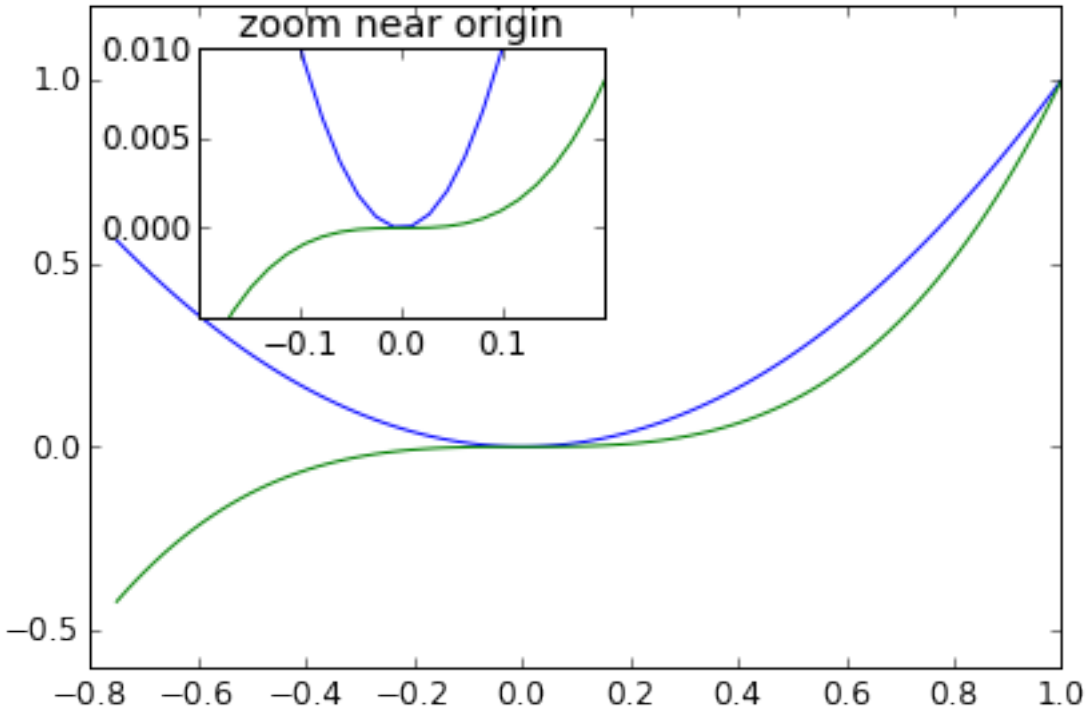
```
import matplotlib.gridspec as gridspec

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]);
```



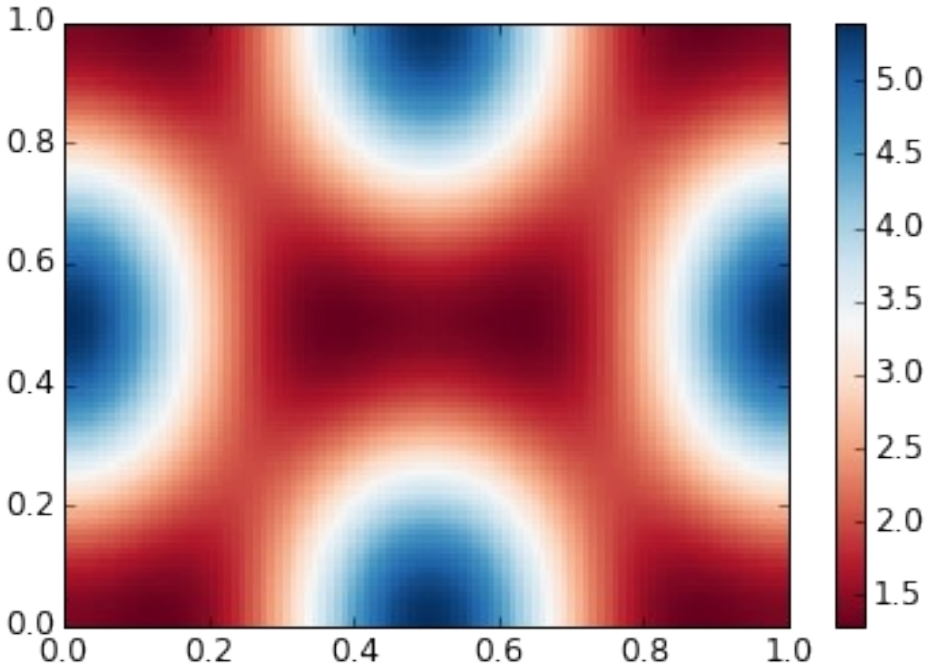
Colormap and contour figures

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
```

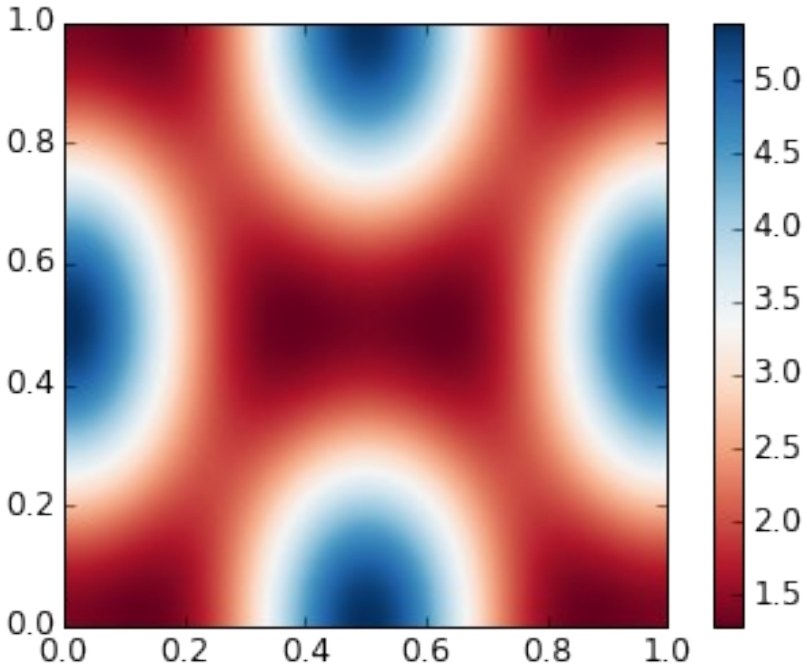
pcolor

```
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)
```



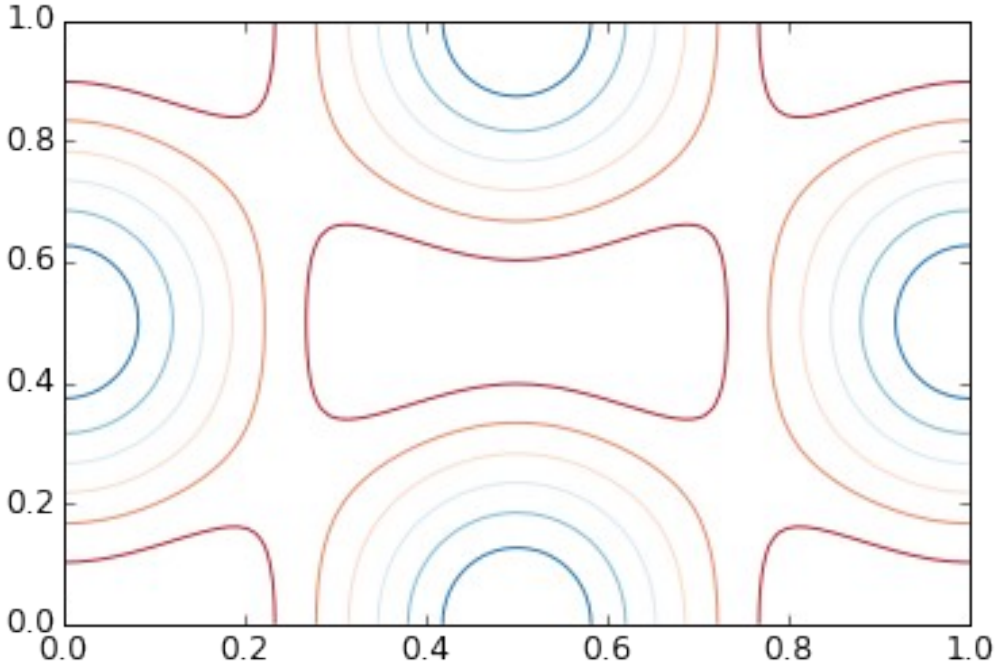
imshow

```
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])
```



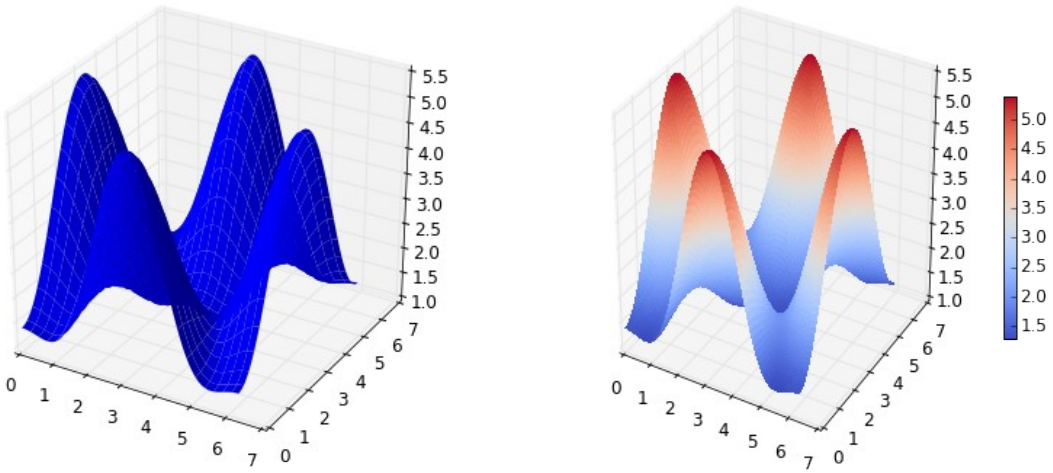
3D figures

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
```

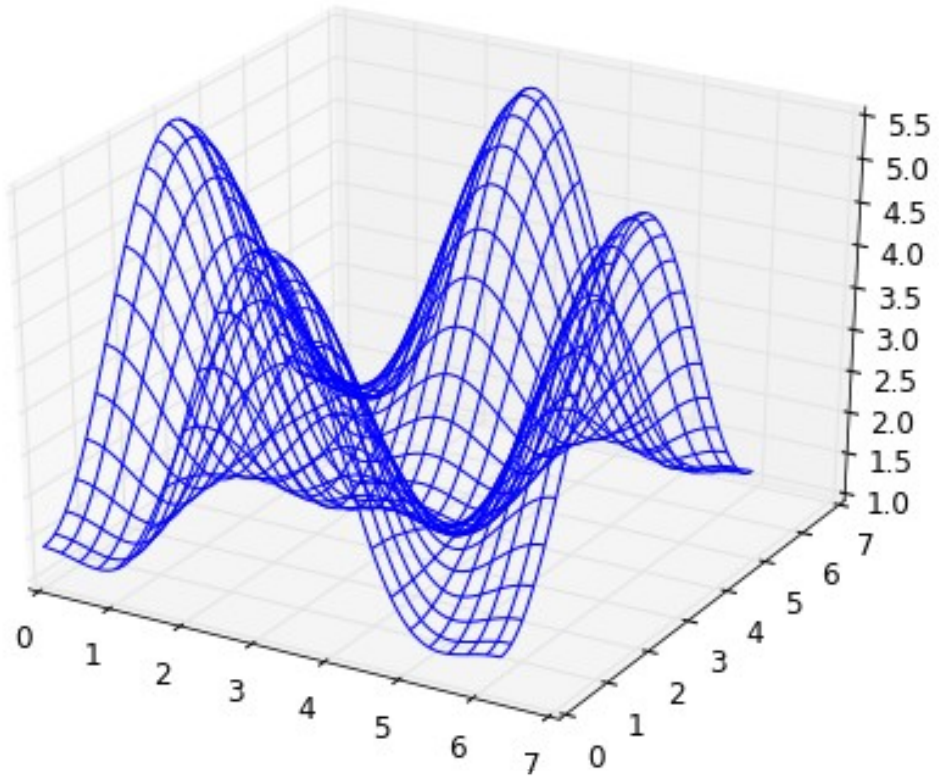
Surface plots

```
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)
```



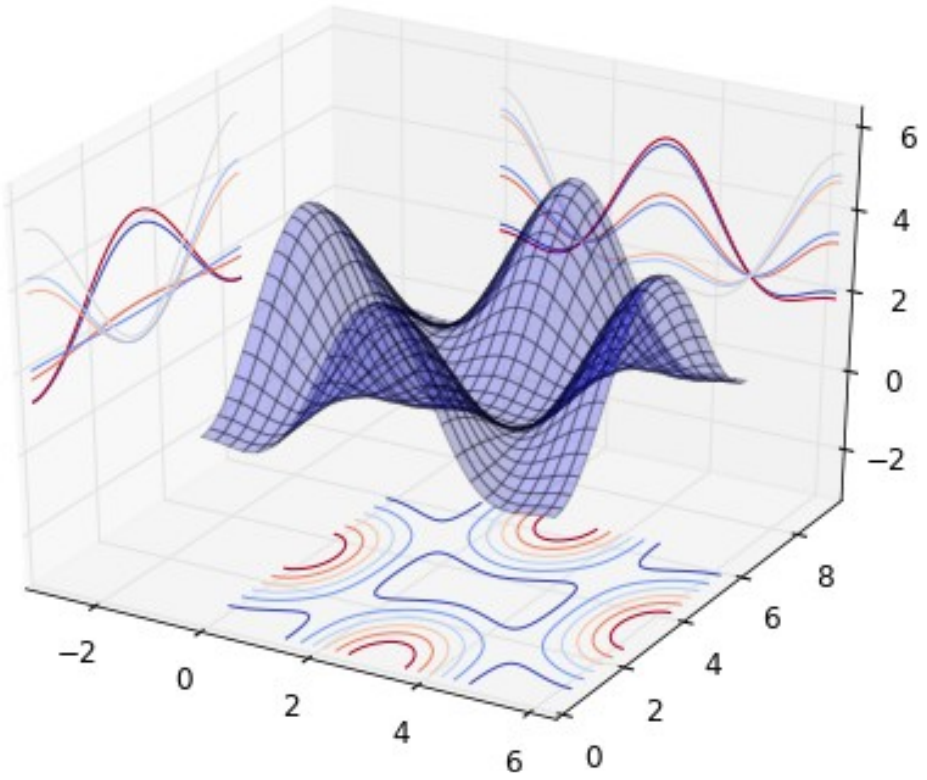
Wire-frame plot

```
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!