

```
In [1]: import matplotlib.pyplot as plt
```

```
In [2]: %matplotlib inline
```

```
In [3]: import numpy as np
x = np.linspace(0, 5, 11)
y = x ** 2
```

```
In [4]: x
```

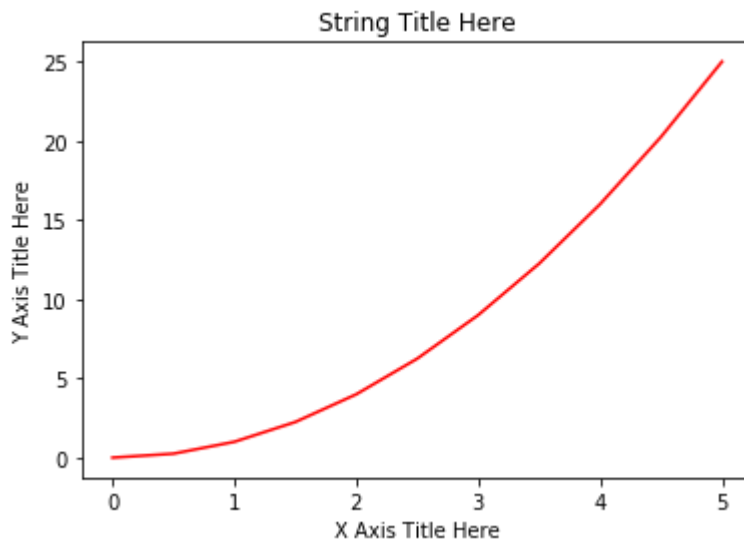
```
Out[4]: array([0. , 0.5, 1. , 1.5, 2. , 2.5, 3. , 3.5, 4. , 4.5, 5. ])
```

```
In [5]: y
```

```
Out[5]: array([ 0. ,  0.25,  1. ,  2.25,  4. ,  6.25,  9. , 12.25, 16. ,
        20.25, 25. ])
```

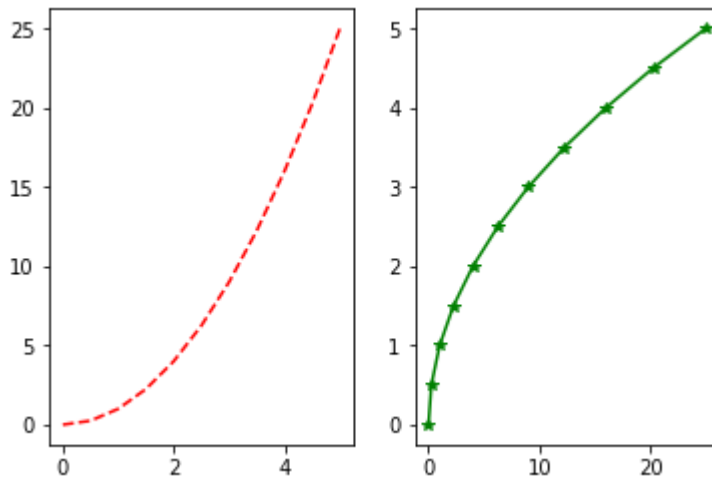
### Basic Matplotlib Command

```
In [6]: plt.plot(x, y, 'r') # 'r' is the color red
plt.xlabel('X Axis Title Here')
plt.ylabel('Y Axis Title Here')
plt.title('String Title Here')
plt.show()
```



### Creating Multiplots on Same Canva

```
In [10]: # plt.subplot(nrows, ncols, plot_number)
plt.subplot(1,2,1)
plt.plot(x, y, 'r--') # More on color options later
plt.subplot(1,2,2)
plt.plot(y, x, 'g*-');
```



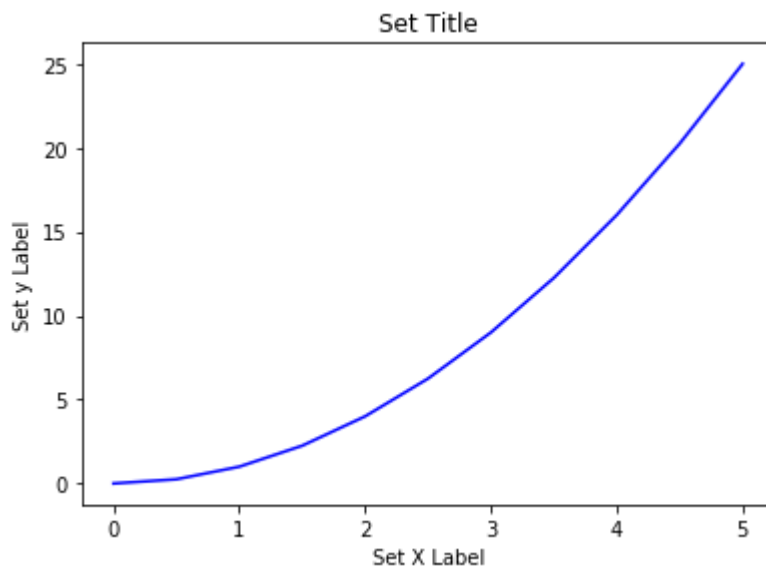
### Matplotlib Object Oriented Method

```
In [12]: # Create Figure (empty canvas)
fig = plt.figure()

# Add set of axes to figure
axes = fig.add_axes([0.1, 0.1, 0.8, 0.8]) # Left, bottom, width, height (range 0

# Plot on that set of axes
axes.plot(x, y, 'b')
axes.set_xlabel('Set X Label') # Notice the use of set_ to begin methods
axes.set_ylabel('Set y Label')
axes.set_title('Set Title')
```

Out[12]: Text(0.5,1,'Set Title')



```

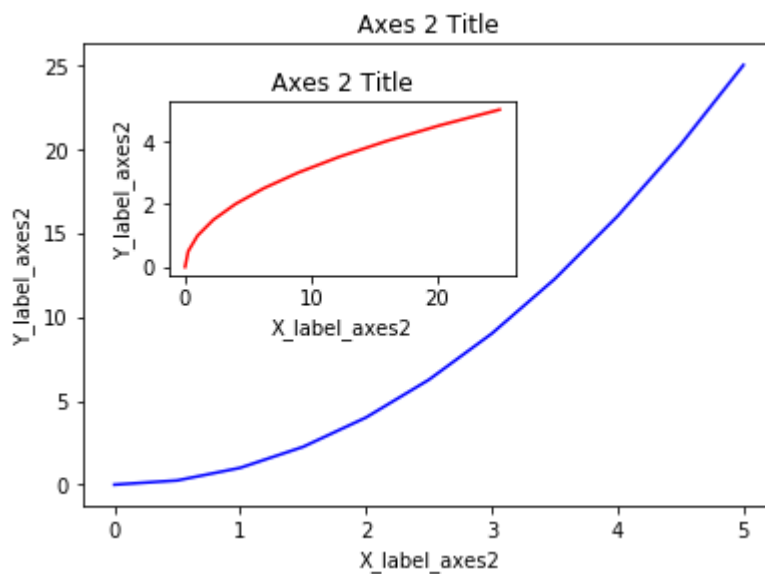
In [13]: # Creates blank canvas
fig = plt.figure()

axes1 = fig.add_axes([0.1, 0.1, 0.8, 0.8]) # main axes
axes2 = fig.add_axes([0.2, 0.5, 0.4, 0.3]) # inset axes

# Larger Figure Axes 1
axes1.plot(x, y, 'b')
axes1.set_xlabel('X_label_axes2')
axes1.set_ylabel('Y_label_axes2')
axes1.set_title('Axes 2 Title')

# Insert Figure Axes 2
axes2.plot(y, x, 'r')
axes2.set_xlabel('X_label_axes2')
axes2.set_ylabel('Y_label_axes2')
axes2.set_title('Axes 2 Title');

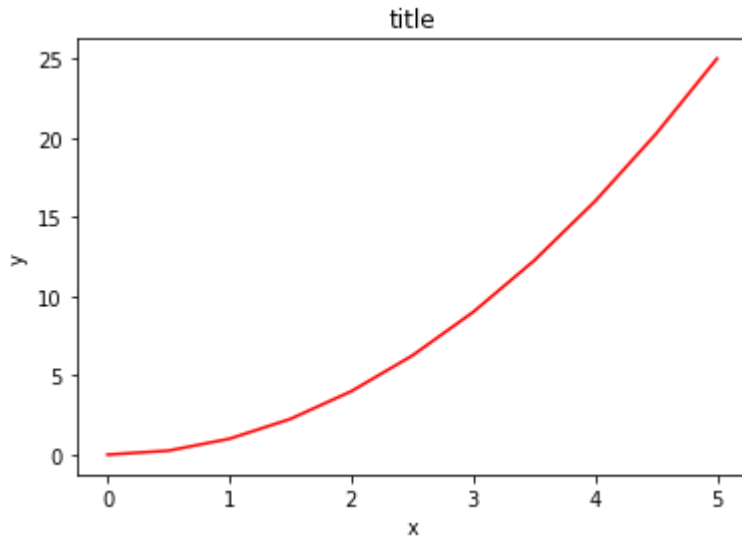
```



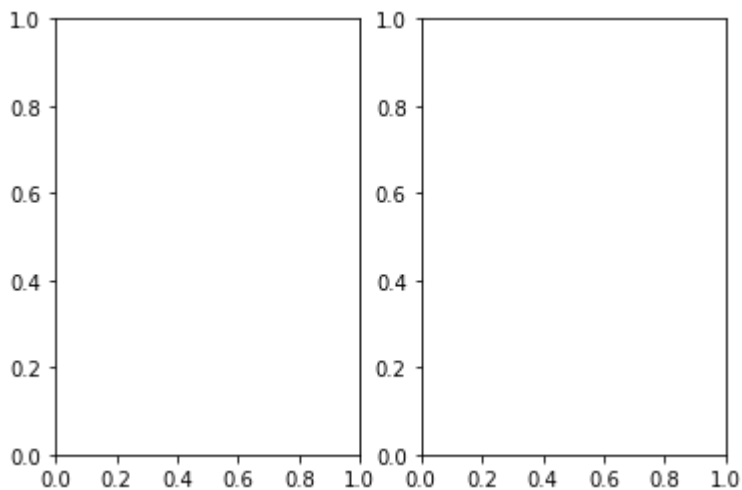
subplots()

```
In [15]: # 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');
```



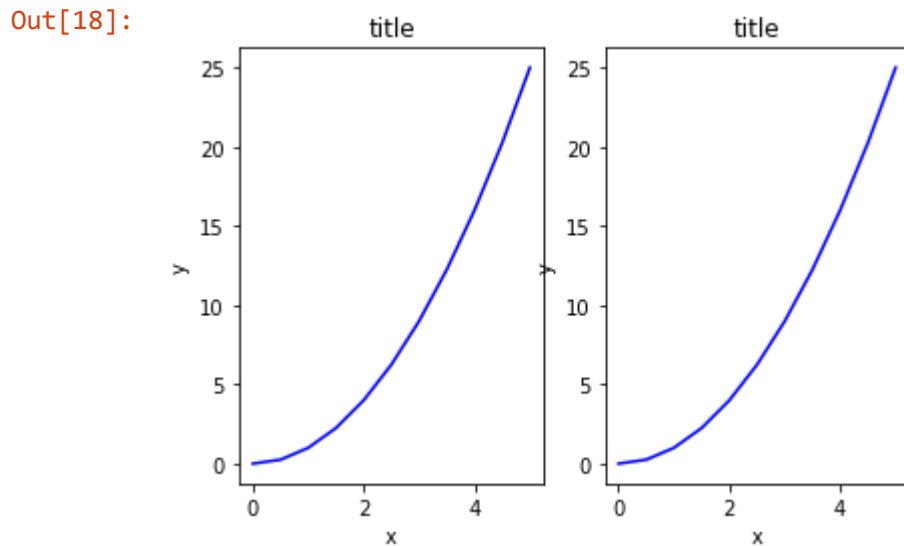
```
In [16]: # Empty canvas of 1 by 2 subplots
fig, axes = plt.subplots(nrows=1, ncols=2)
```



```
In [17]: # Axes is an array of axes to plot on
axes
```

```
Out[17]: array([<matplotlib.axes._subplots.AxesSubplot object at 0x0000000005325240>,
                <matplotlib.axes._subplots.AxesSubplot object at 0x000000000534C588>],
              dtype=object)
```

```
In [18]: for ax in axes:  
          ax.plot(x, y, 'b')  
          ax.set_xlabel('x')  
          ax.set_ylabel('y')  
          ax.set_title('title')  
  
          # Display the figure object  
fig
```

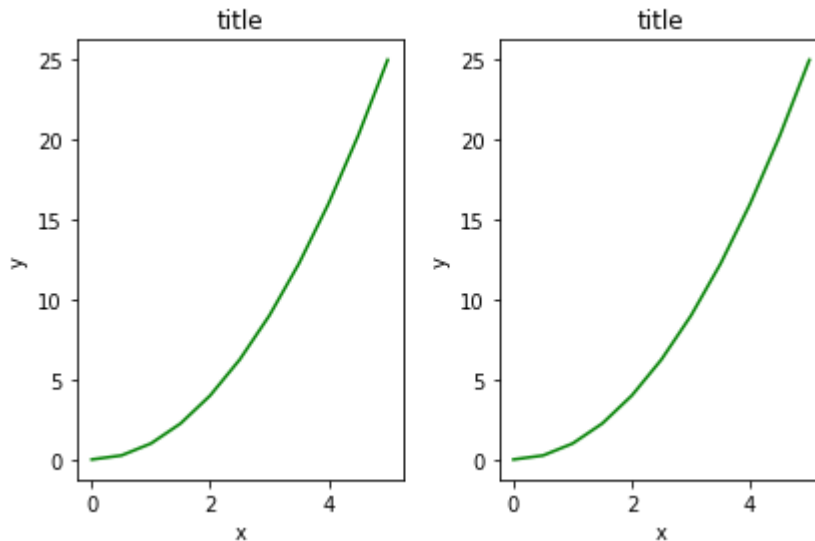


A common issue with matplotlib is overlapping subplots or figures. We can use `fig.tight_layout()` or `plt.tight_layout()` method, which automatically adjusts the positions of the axes on the figure canvas so that there is no overlapping content:

```
In [19]: fig, axes = plt.subplots(nrows=1, ncols=2)

for ax in axes:
    ax.plot(x, y, 'g')
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_title('title')

fig
plt.tight_layout()
```



## Figure size, aspect ratio and DPI

Matplotlib allows the aspect ratio, DPI and figure size to be specified when the Figure object is created. You can use the `figsize` and `dpi` keyword arguments.

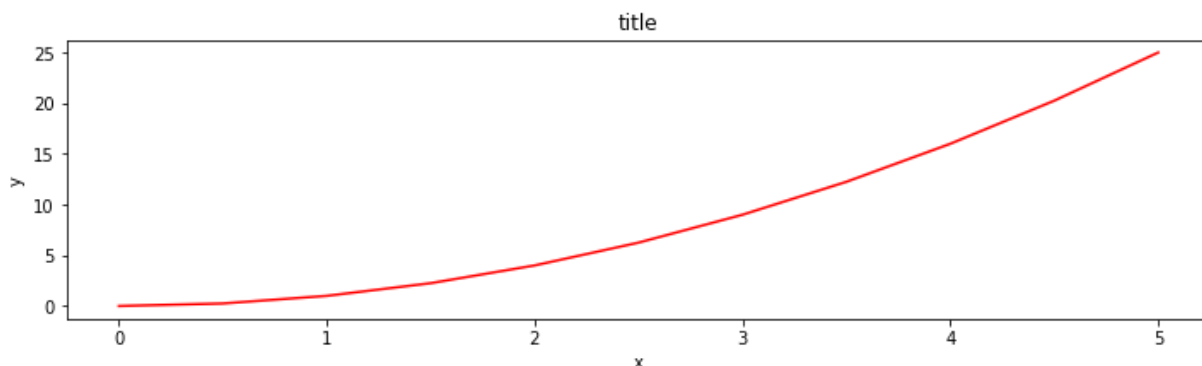
- `figsize` is a tuple of the width and height of the figure in inches
- `dpi` is the dots-per-inch (pixel per inch).

```
In [20]: fig = plt.figure(figsize=(8,4), dpi=100)

<Figure size 800x400 with 0 Axes>
```

```
In [21]: fig, axes = plt.subplots(figsize=(12,3))

axes.plot(x, y, 'r')
axes.set_xlabel('x')
axes.set_ylabel('y')
axes.set_title('title');
```



## Saving figures

Matplotlib can generate high-quality output in a number formats, including PNG, JPG, EPS, SVG, PGF and PDF. To save a figure to a file we can use the `savefig` method in the `Figure` class:

```
In [22]: fig.savefig("filename.png")
```

```
In [23]: fig.savefig("filename.png", dpi=200)
```

## Legends, labels and titles

Now that we have covered the basics of how to create a figure canvas and add axes instances to the canvas, let's look at how decorate a figure with titles, axis labels, and legends.

### Figure titles

A title can be added to each axis instance in a figure. To set the title, use the `set_title` method in the axes instance:

```
In [25]: ax.set_title("title");
```

### Axis labels

Similarly, with the methods `set_xlabel` and `set_ylabel`, we can set the labels of the X and Y axes:

```
In [26]: ax.set_xlabel("x")
ax.set_ylabel("y");
```

## Legends

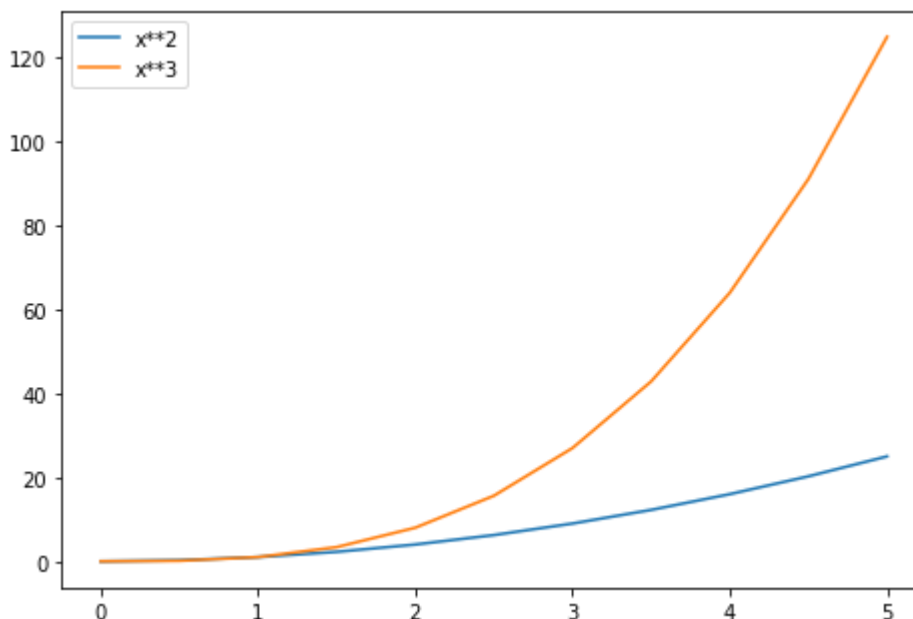
You can use the **label="label text"** keyword argument when plots or other objects are added to the figure, and then using the **legend** method without arguments to add the legend to the figure:

```
In [27]: 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()
```

Out[27]: <matplotlib.legend.Legend at 0x83d7d30>



Notice how are legend overlaps 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](http://matplotlib.org/users/legend_guide.html#legend-location) ([http://matplotlib.org/users/legend\\_guide.html#legend-location](http://matplotlib.org/users/legend_guide.html#legend-location)) for details. Some of the most common **loc** values are:



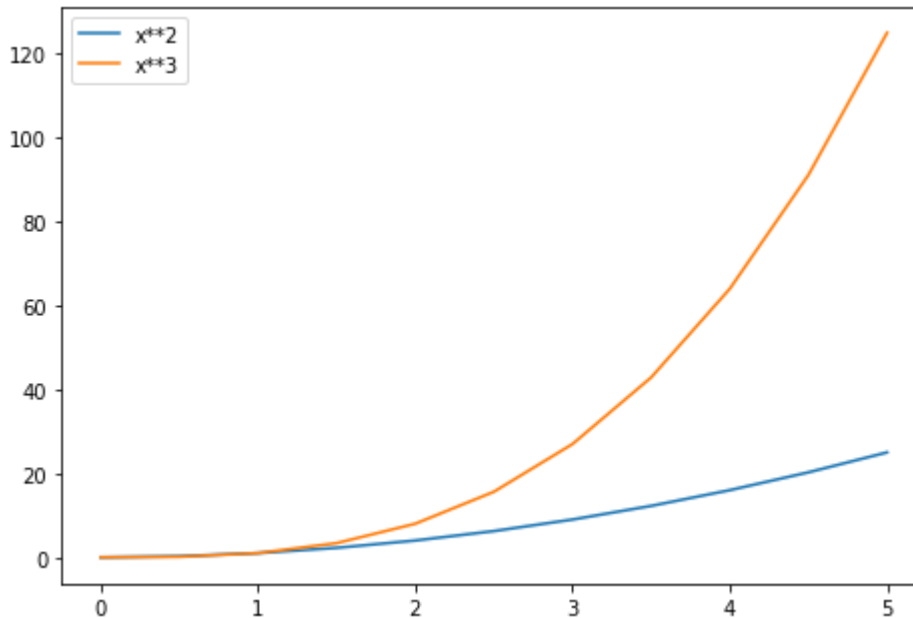
```
In [28]: # Lots of options....

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
```

Out[28]:



## Setting colors, linewidths, linetypes

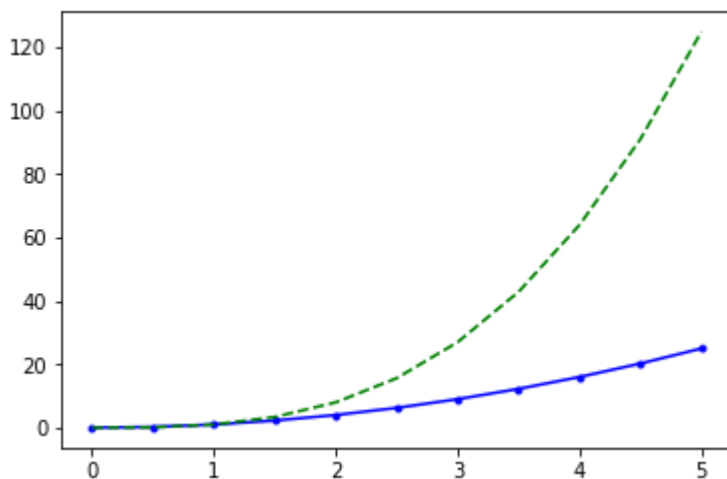
Matplotlib gives you a *lot* of options for customizing colors, linewidths, and linetypes.

There is the basic MATLAB like syntax (which I would suggest you avoid using for more clarity sake:

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:

```
In [30]: # 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
```

```
Out[30]: [<matplotlib.lines.Line2D at 0x678ef60>]
```



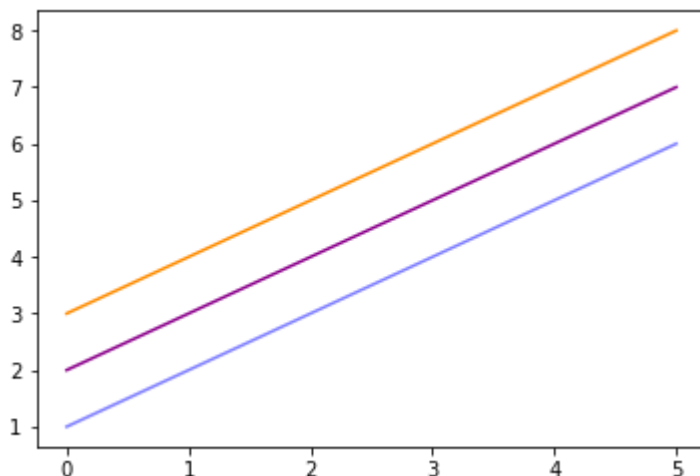
## Colors with the color= parameter

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.

```
In [31]: fig, ax = plt.subplots()

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

```
Out[31]: [<matplotlib.lines.Line2D at 0x67dd780>]
```



## Line and marker styles

```
In [35]: fig, ax = plt.subplots(figsize=(12,6))

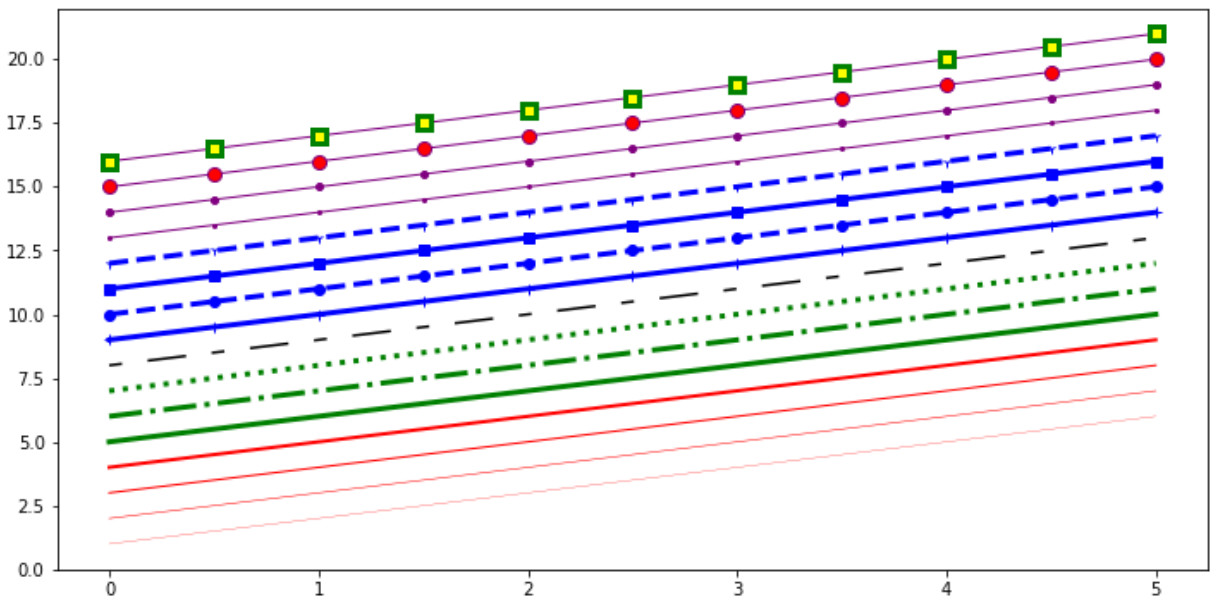
ax.plot(x, x+1, color="red", linewidth=0.25)
ax.plot(x, x+2, color="red", linewidth=0.50)
ax.plot(x, x+3, color="red", linewidth=1.00)
ax.plot(x, x+4, color="red", linewidth=2.00)

# possible linestyle options '-', '-.', ':', 'steps'
ax.plot(x, x+5, color="green", lw=3, linestyle='-')
ax.plot(x, x+6, color="green", lw=3, ls='-.')
ax.plot(x, x+7, color="green", lw=3, ls=':')

# custom dash
line, = ax.plot(x, x+8, color="black", lw=1.50)
line.set_dashes([5, 10, 15, 10]) # format: line length, space length, ...

# possible marker symbols: marker = '+', 'o', '*', 's', ',', '.', '1', '2', '3',
ax.plot(x, x+ 9, color="blue", lw=3, ls='-', marker='+')
ax.plot(x, x+10, color="blue", lw=3, ls='--', marker='o')
ax.plot(x, x+11, color="blue", lw=3, ls='-', marker='s')
ax.plot(x, x+12, color="blue", lw=3, ls='--', marker='1')

# marker size and color
ax.plot(x, x+13, color="purple", lw=1, ls='-', marker='o', markersize=2)
ax.plot(x, x+14, color="purple", lw=1, ls='-', marker='o', markersize=4)
ax.plot(x, x+15, color="purple", lw=1, ls='-', marker='o', markersize=8, markerfacecolor="yellow")
ax.plot(x, x+16, color="purple", lw=1, ls='-', marker='s', markersize=8,
        markerfacecolor="yellow", markeredgewidth=3, markeredgewidth="green");
```



## Plot range

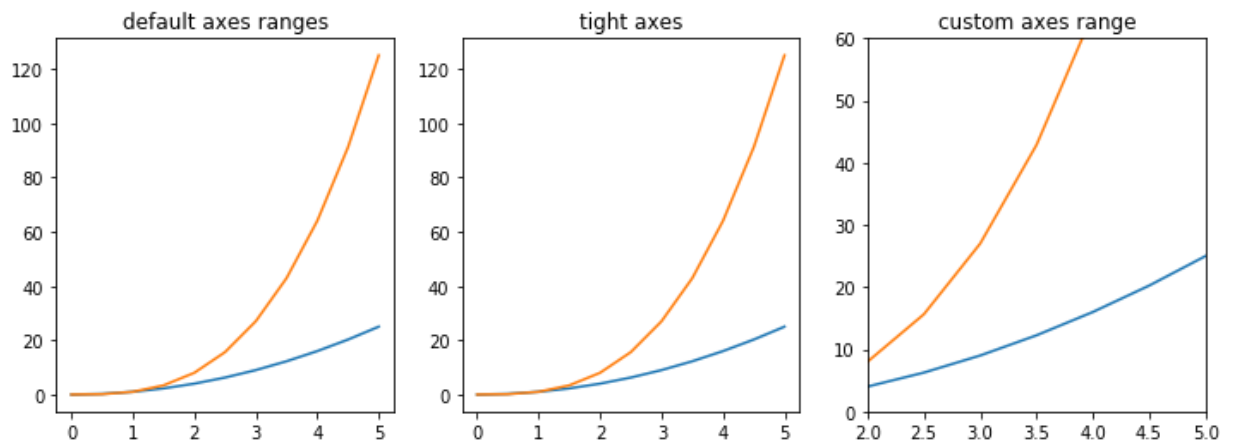
We can configure the ranges of the axes using the `set_ylim` and `set_xlim` methods in the axis object, or `axis('tight')` for automatically getting "tightly fitted" axes ranges:

```
In [36]: fig, axes = plt.subplots(1, 3, figsize=(12, 4))

axes[0].plot(x, x**2, x, x**3)
axes[0].set_title("default axes ranges")

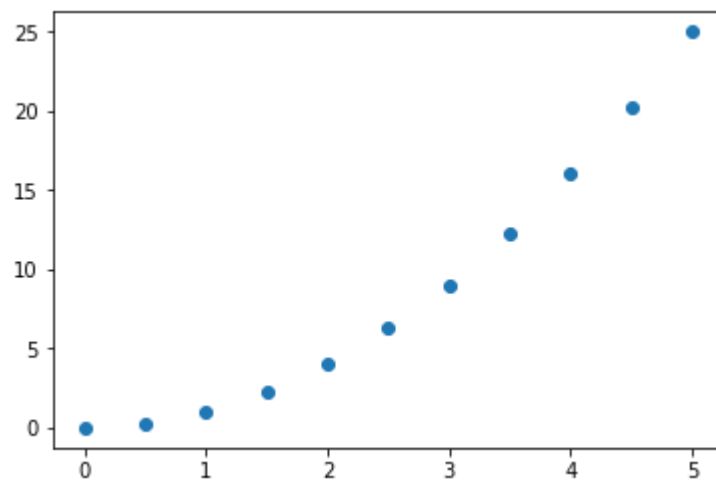
axes[1].plot(x, x**2, x, x**3)
axes[1].axis('tight')
axes[1].set_title("tight axes")

axes[2].plot(x, x**2, x, x**3)
axes[2].set_ylim([0, 60])
axes[2].set_xlim([2, 5])
axes[2].set_title("custom axes range");
```



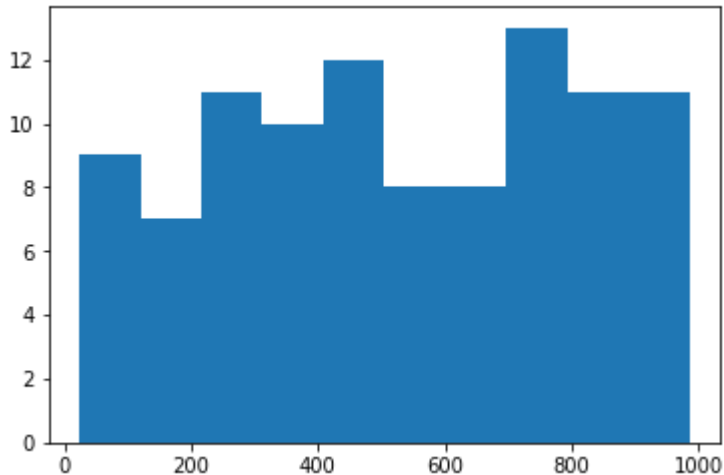
```
In [37]: plt.scatter(x,y)
```

```
Out[37]: <matplotlib.collections.PathCollection at 0x9de0518>
```



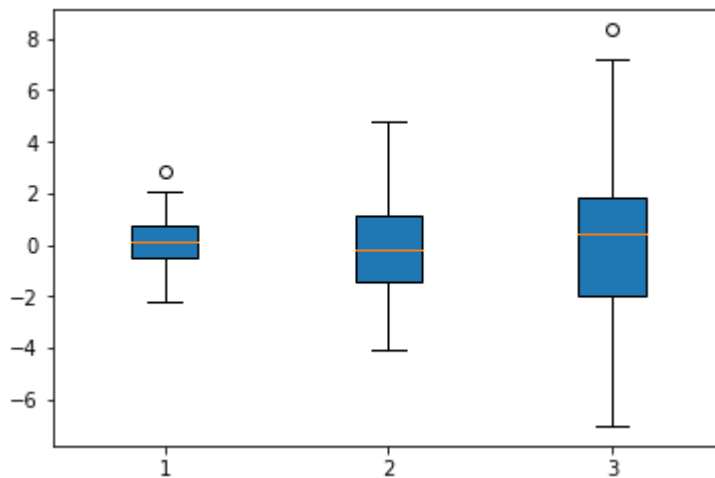
```
In [38]: from random import sample
data = sample(range(1, 1000), 100)
plt.hist(data)
```

```
Out[38]: (array([ 9.,  7., 11., 10., 12.,  8.,  8., 13., 11., 11.]),
array([ 24. , 120.1, 216.2, 312.3, 408.4, 504.5, 600.6, 696.7, 792.8,
      888.9, 985. ]),
<a list of 10 Patch objects>)
```



```
In [39]: data = [np.random.normal(0, std, 100) for std in range(1, 4)]

# rectangular box plot
plt.boxplot(data, vert=True, patch_artist=True);
```



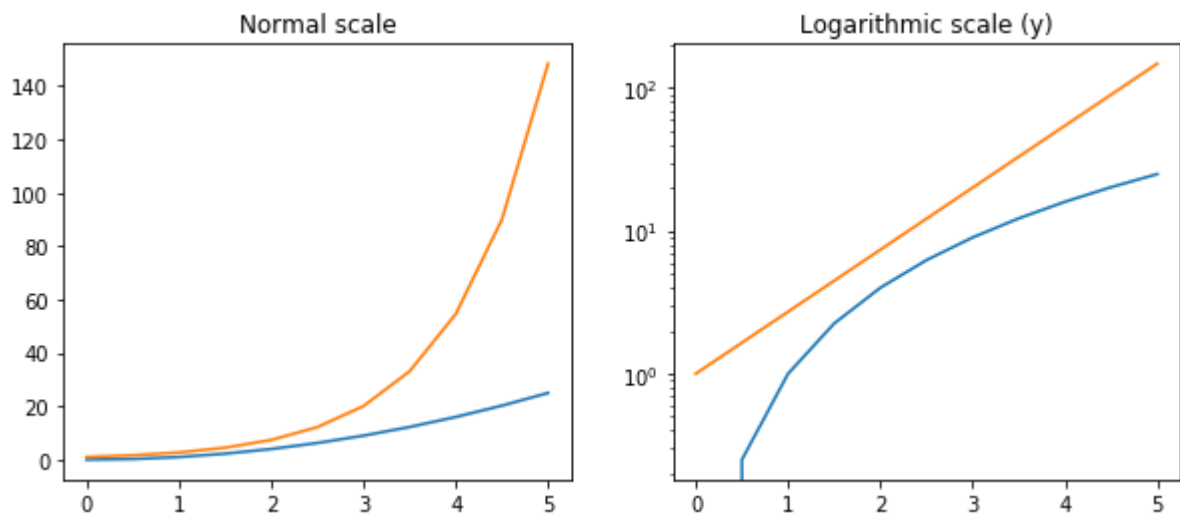
### Logarithmic scale

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 separately using `set_xscale` and `set_yscale` methods which accept one parameter (with the value "log" in this case):

```
In [40]: 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

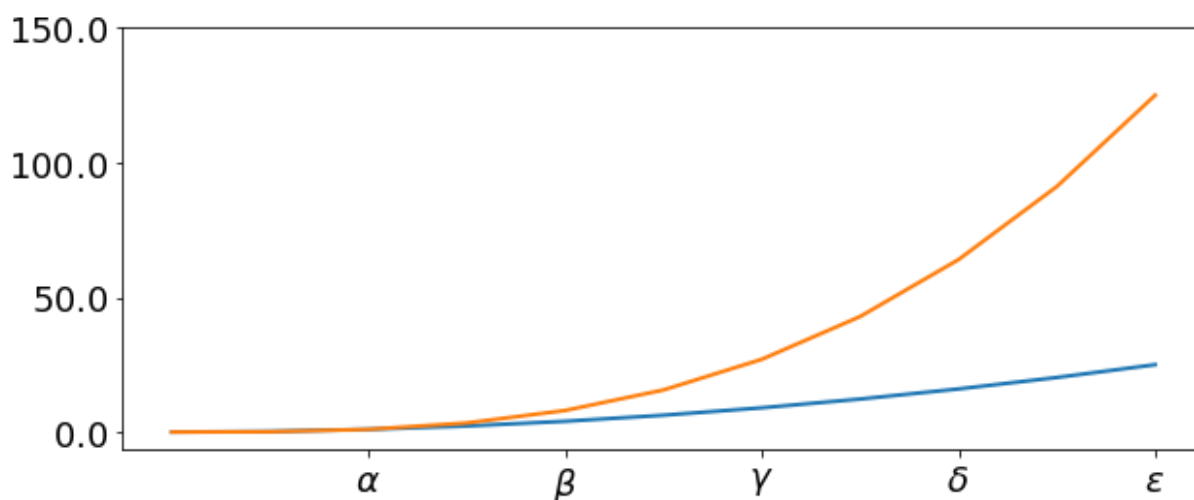
We can explicitly determine where we want the axis ticks with `set_xticks` and `set_yticks`, which both take a list of values for where on the axis the ticks are to be placed. We can also use the `set_xticklabels` and `set_yticklabels` methods to provide a list of custom text labels for each tick location:

```
In [41]: 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'$\epsilon$'])

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

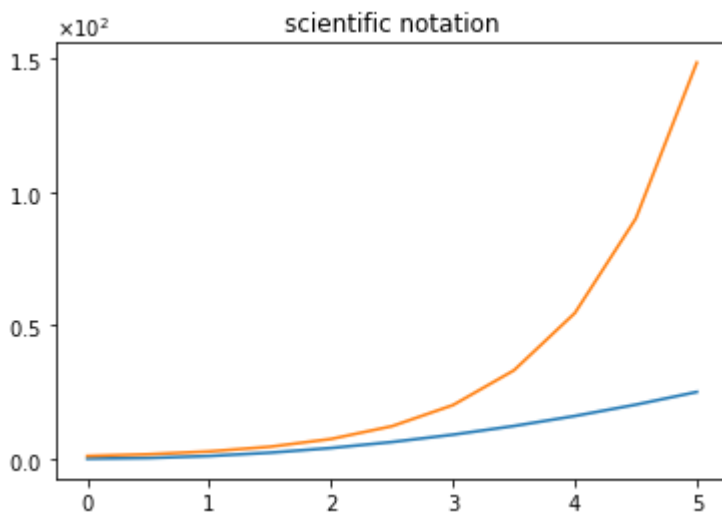


```
In [42]: fig, ax = plt.subplots(1, 1)

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)
```



Type *Markdown* and LaTeX:  $\alpha^2$

In [ ]:

In [58]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]: