DATA VISUALIZATION IN MATPLOTLIB

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. 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

IMPORTING I am going to import the matplotlib.pyplot module under the name plt

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

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

EXAMPLE Here, I will be passing numpy arrays or pandas columns (which essentially also behave like arrays).

In [3]:
```
#The data I want to plot:
import numpy as np
x = np.linspace(0, 5, 11)
y = x ** 2
```

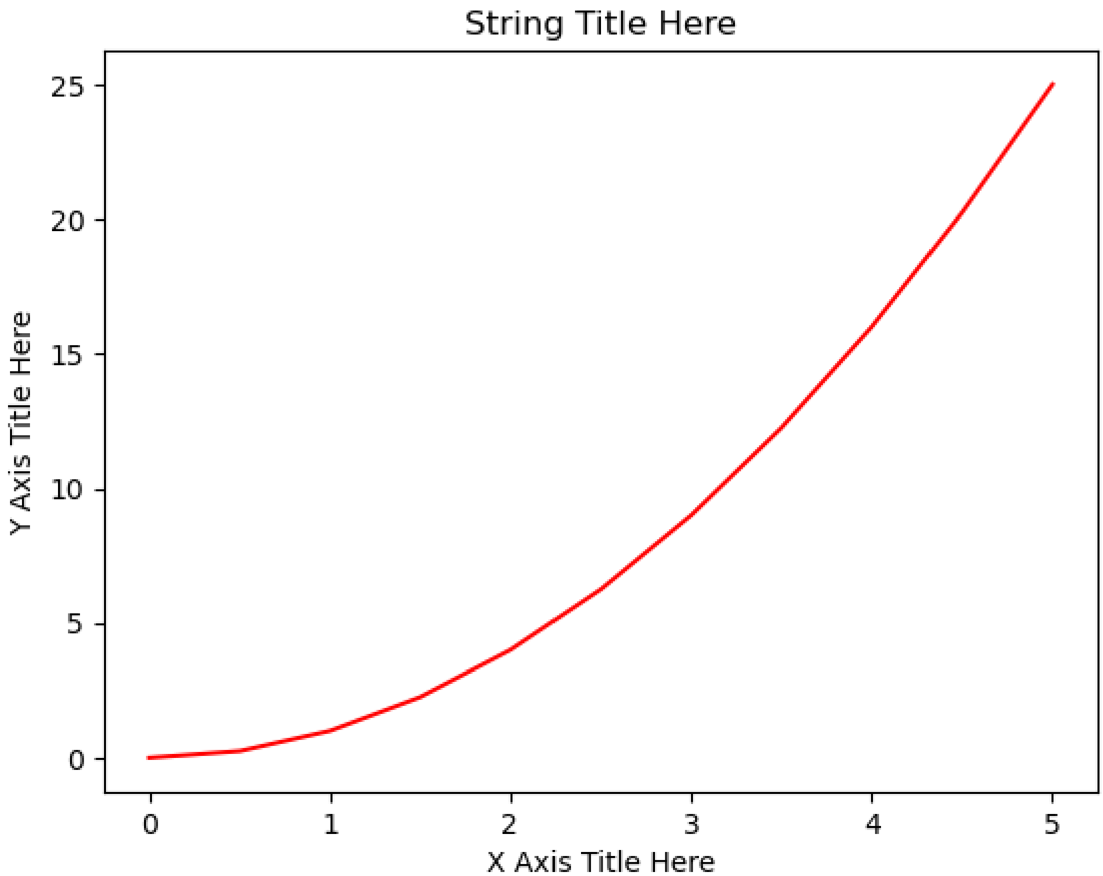
In [4]:
```
x
```

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

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

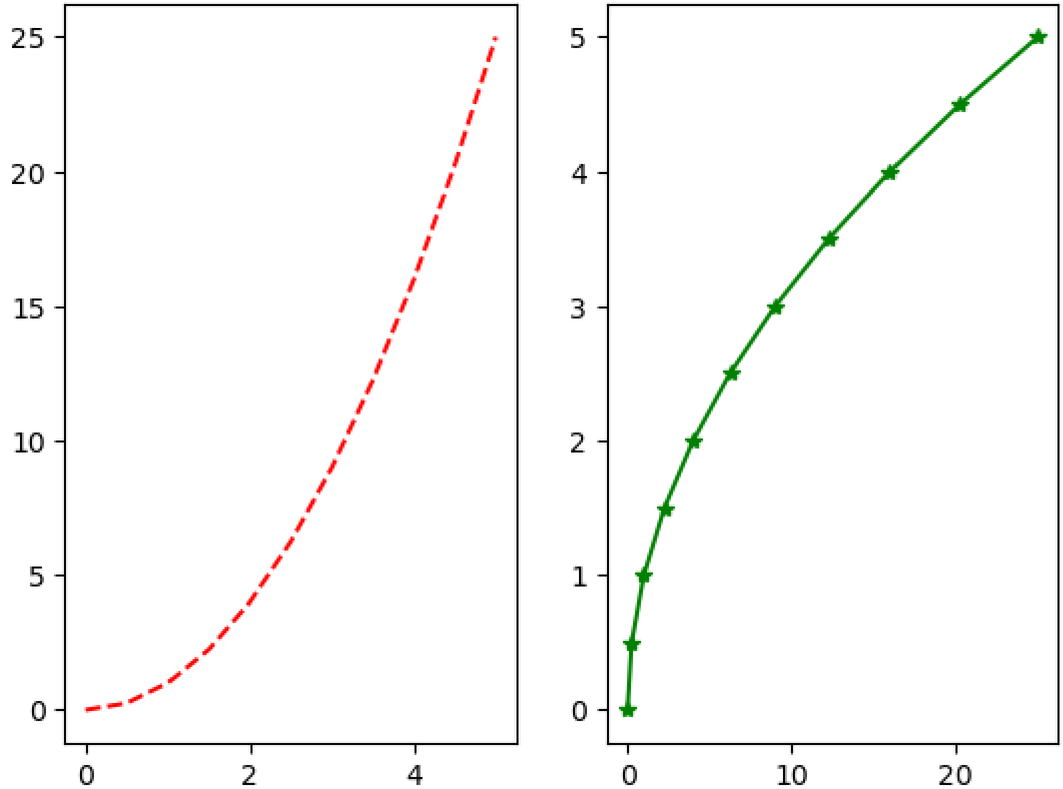
Out[5]:
```
array([ 0.  ,  0.25,  1.  ,  2.25,  4.  ,  6.25,  9.  , 12.25, 16.  ,
       20.25, 25.  ])
```

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 CANVAS

In [7]:
```
# 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*-');
```



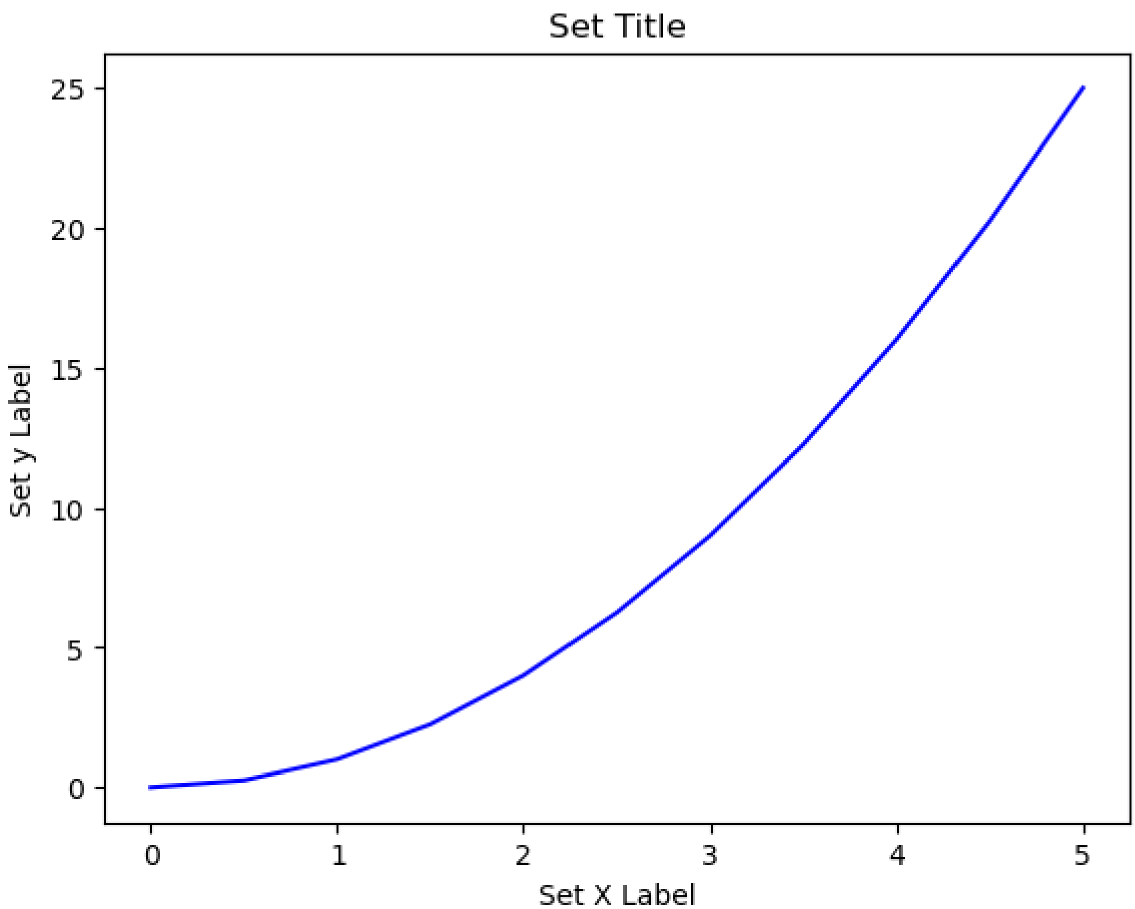
MATPLOTLIB OBJECT ORIENTED METHOD Now that we've seen the basics, I will break it all down with a more formal introduction of Matplotlib's Object Oriented API. This means I will instantiate figure objects and then call methods or attributes from that object.

INTRODUCTION TO THE OBJECT ORIENTED METHOD 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.

To begin, I will create a figure instance. Then, I can add axes to that figure:

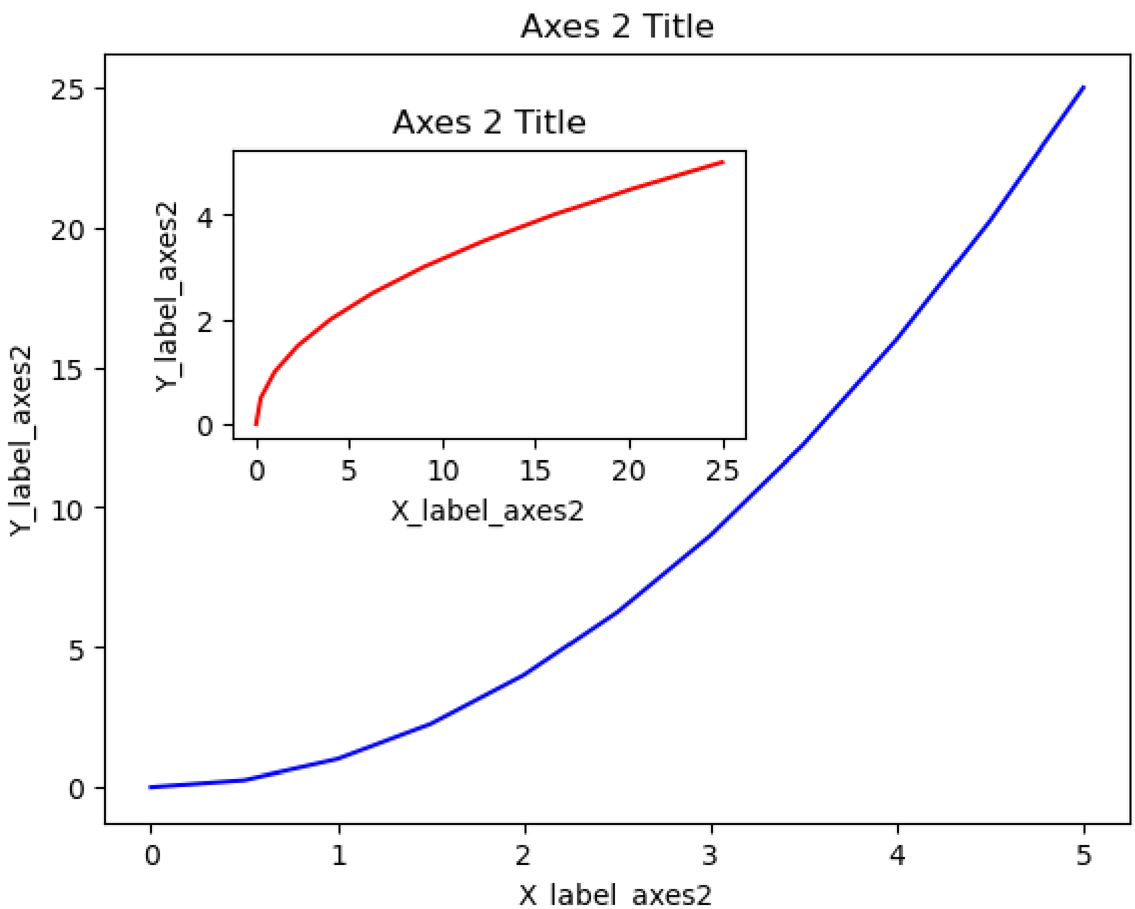
In [8]:
```
# Create Figure (empty canvas)
fig = plt.figure()

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

# 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[8]:
```
Text(0.5, 1.0, 'Set Title')
```



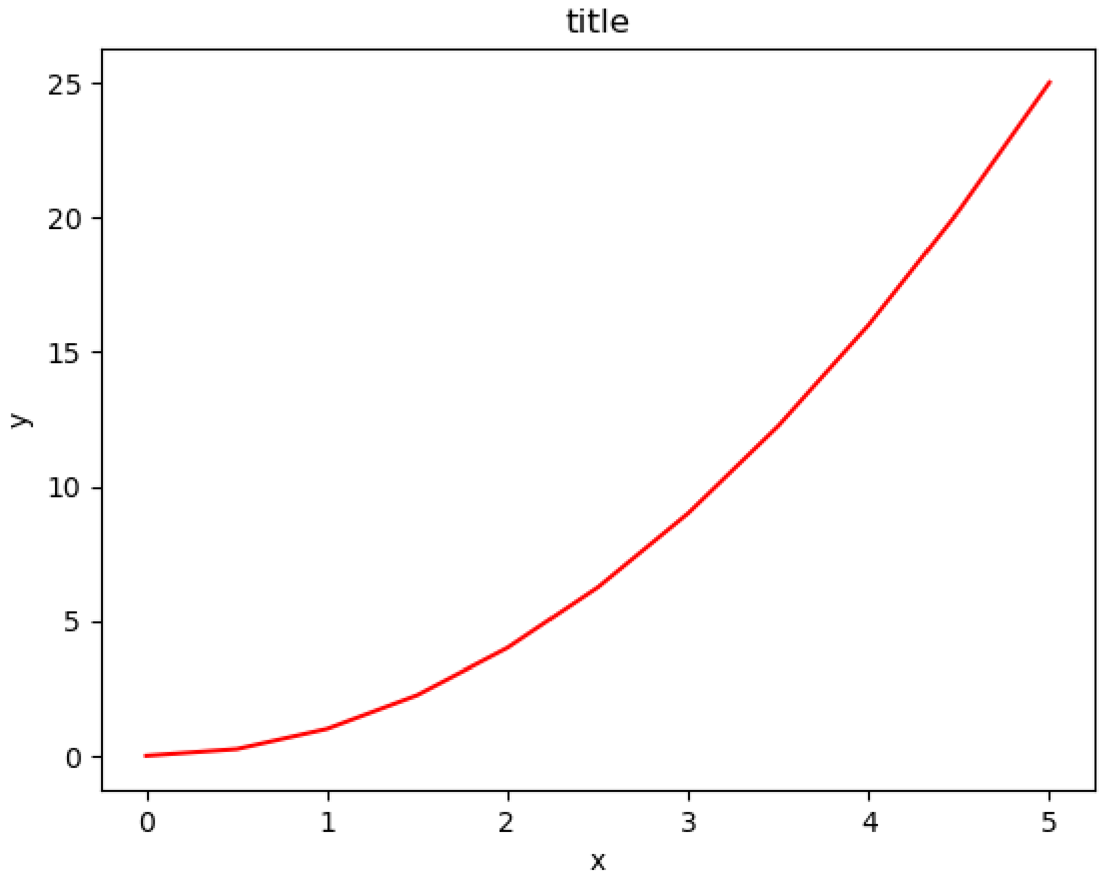
Code is a little more complicated, but the advantage is that I now have full control of where the plot axes are placed, and I can easily add more than one axis to the figure:

In [9]:
```
# 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');
```



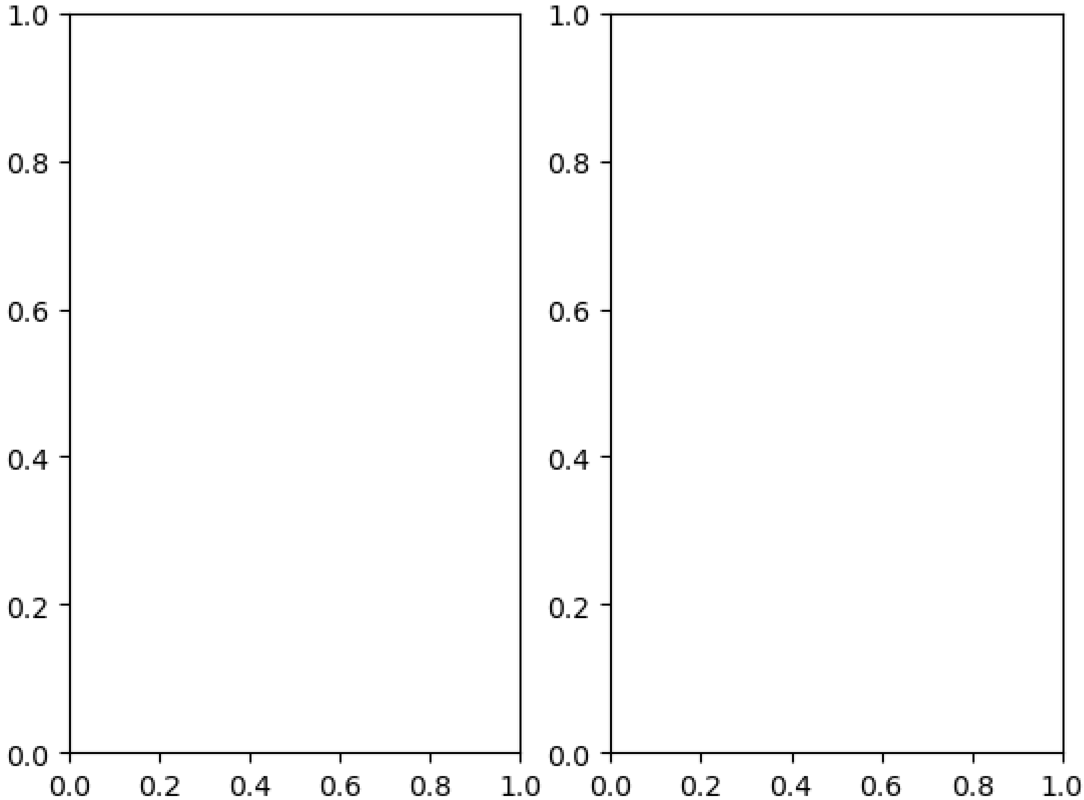
SUBPLOTS() The plt.subplots() object will act as a more automatic axis manager.

In [10]:
```
# 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');
```



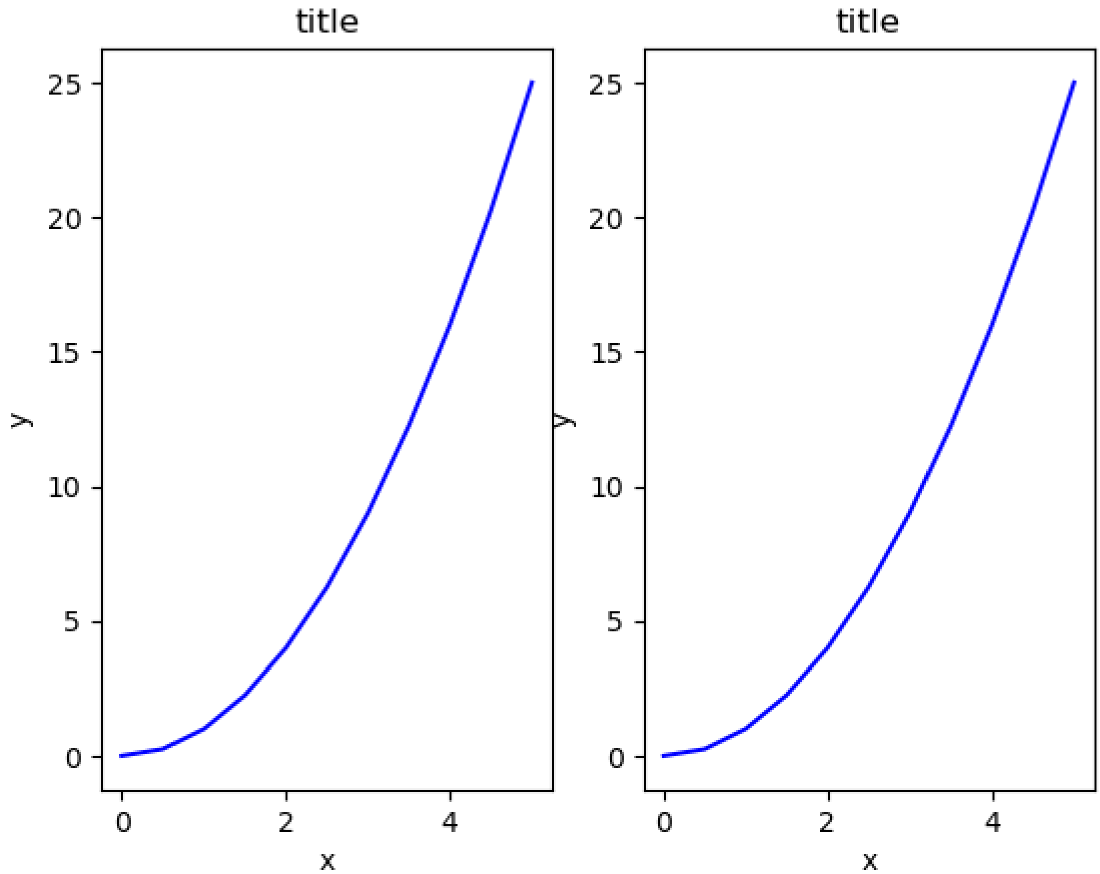
Then, I specified the number of rows and columns when creating the subplots() object:

In [11]:
```
# Empty canvas of 1 by 2 subplots
fig, axes = plt.subplots(nrows=1, ncols=2)
```



In [12]:
```
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
```

Out[12]:



In [13]:

```
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()
```

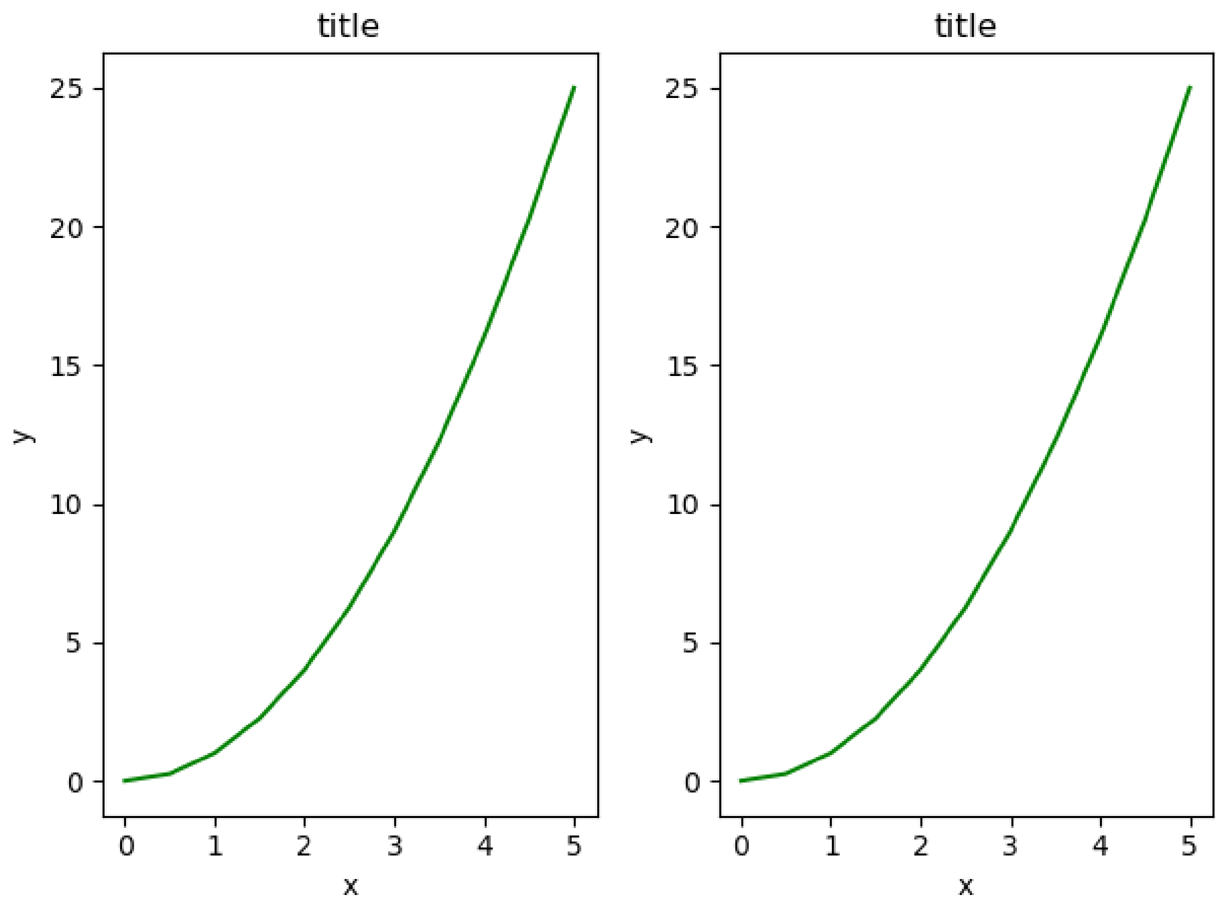
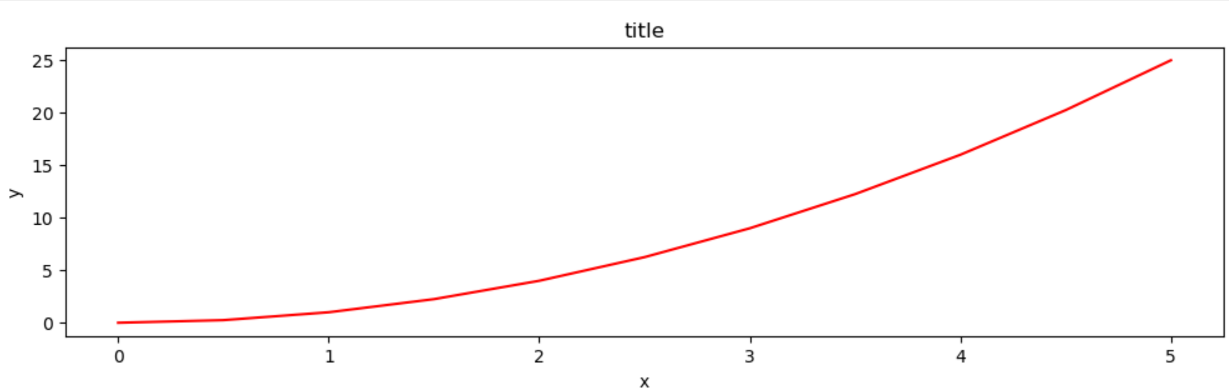


FIGURE SIZE, ASPECT RATIO AND DPI Matplotlib allows the aspect ratio, DPI and figure size to be specified when the Figure object is created. I can use the figsize and dpi keyword arguments.

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

In [14]:
```
fig, axes = plt.subplots(figsize=(12,3))

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

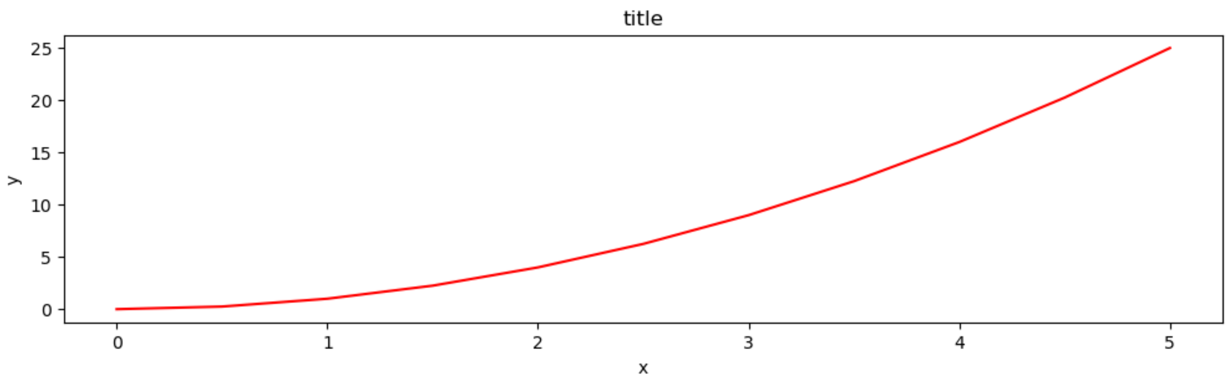


LEGENDS, LABELS AND TITLES

In [15]:
```python
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

# Most common to choose
ax.legend(loc=0) # let matplotlib decide the optimal location
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.
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.
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.
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.
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.
```

Out[15]:

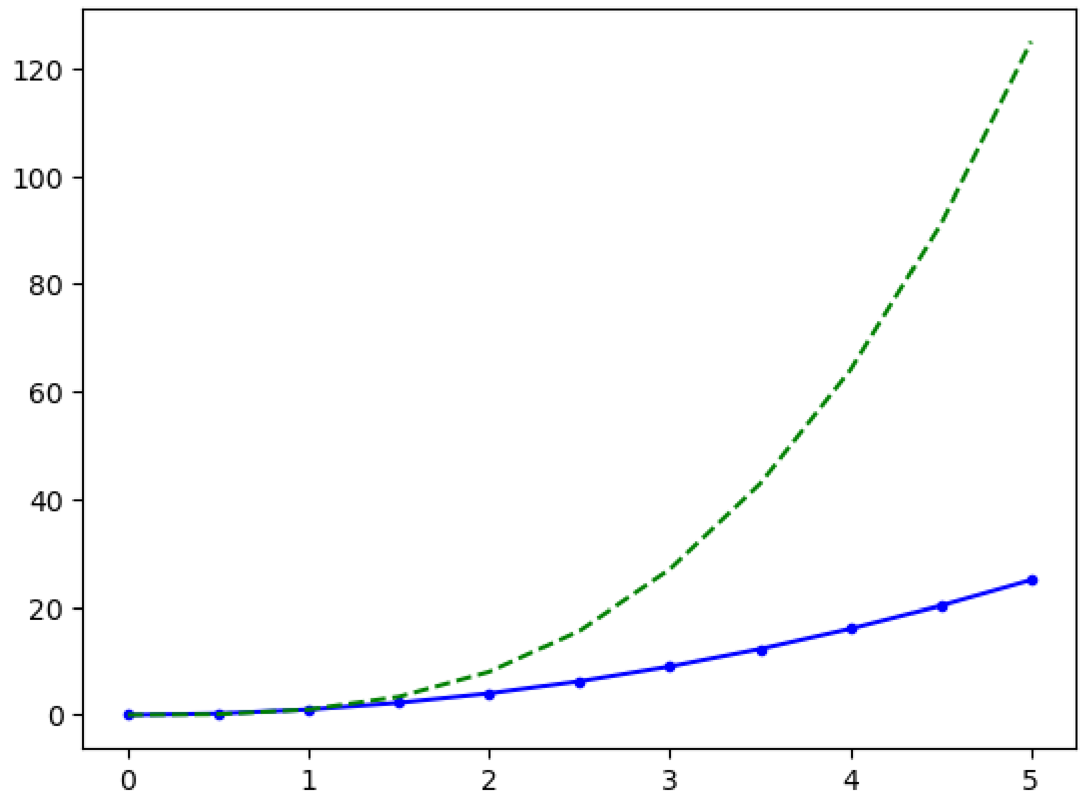


SETTING COLORS, LINEWIDTHS, LINETYPES Matplotlib gives me a lot of options for customizing colors, linewidths, and linetypes.

Colors with MatLab like syntax With matplotlib, I can define the colors of lines and other graphical elements in a number of ways. First of all, I 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:

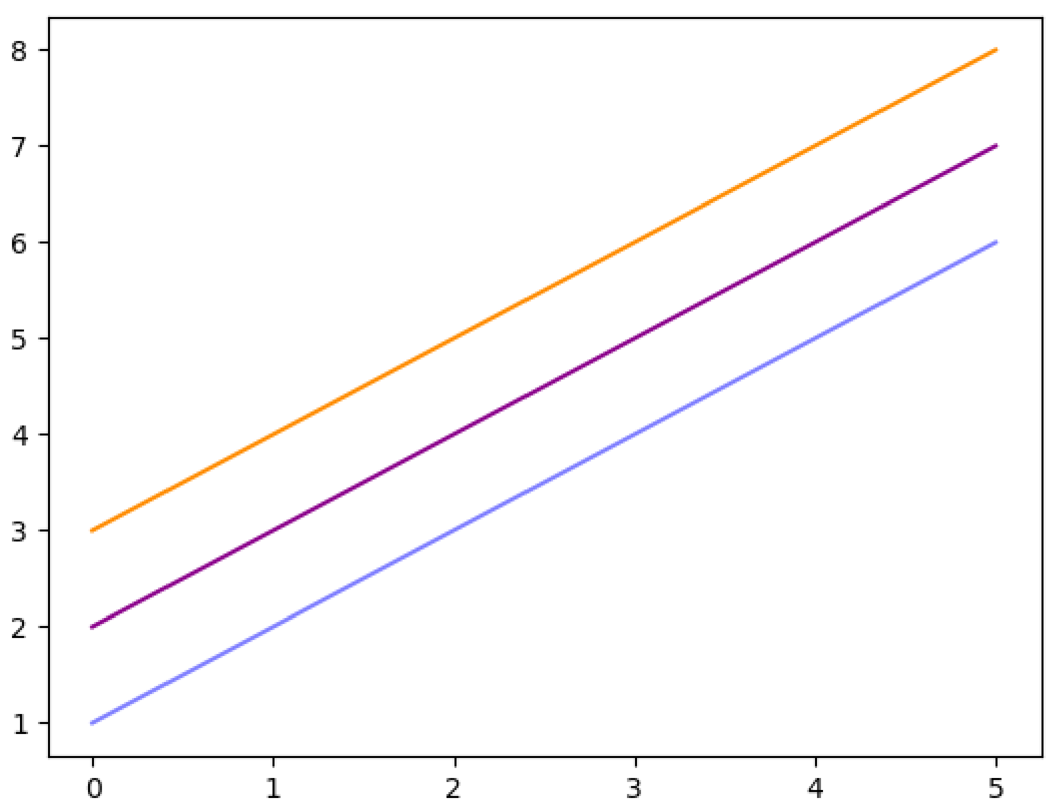
In [16]:
```python
# MATLAB style line color and style
fig, ax = plt.subplots()
ax.plot(x, x**2, 'b.-') # blue line with dots
ax.plot(x, x**3, 'g--') # green dashed line
```

Out[16]:
```
[<matplotlib.lines.Line2D at 0x117fb42b0>]
```



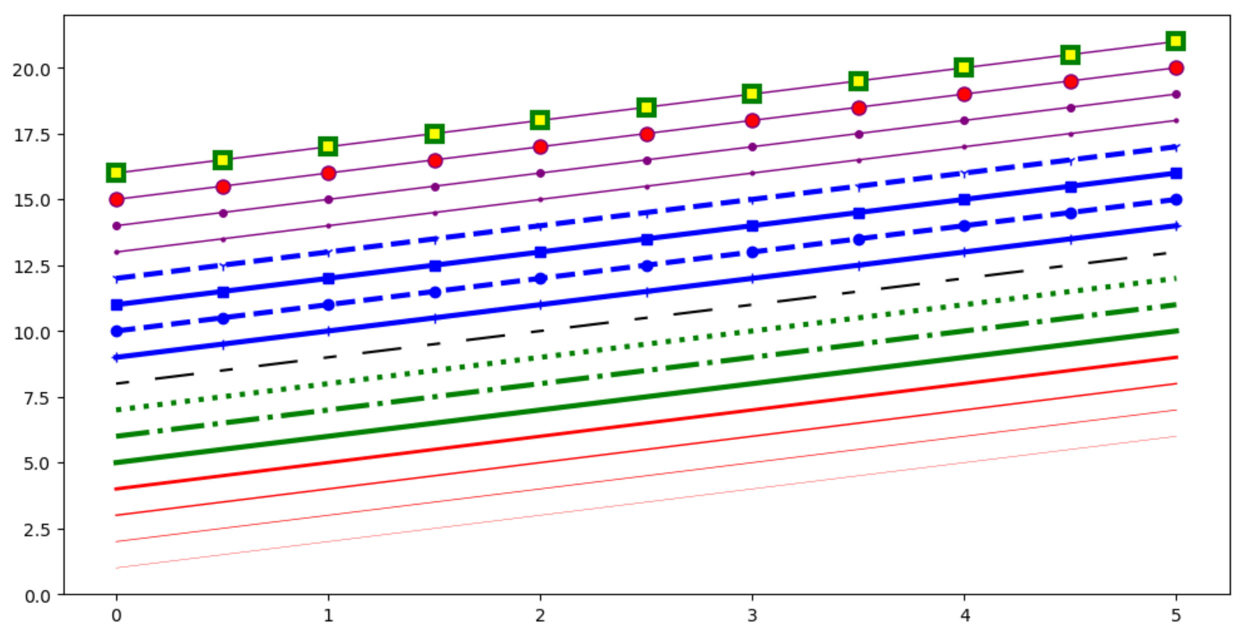
In [17]:
```
fig, ax = plt.subplots()

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
```

Out[17]:
```
[<matplotlib.lines.Line2D at 0x122015840>]
```



LINE AND MARKER STYLES To change the line width, I can use the linewidth or lw keyword argument. The line style can be selected using the linestyle or ls keyword arguments:

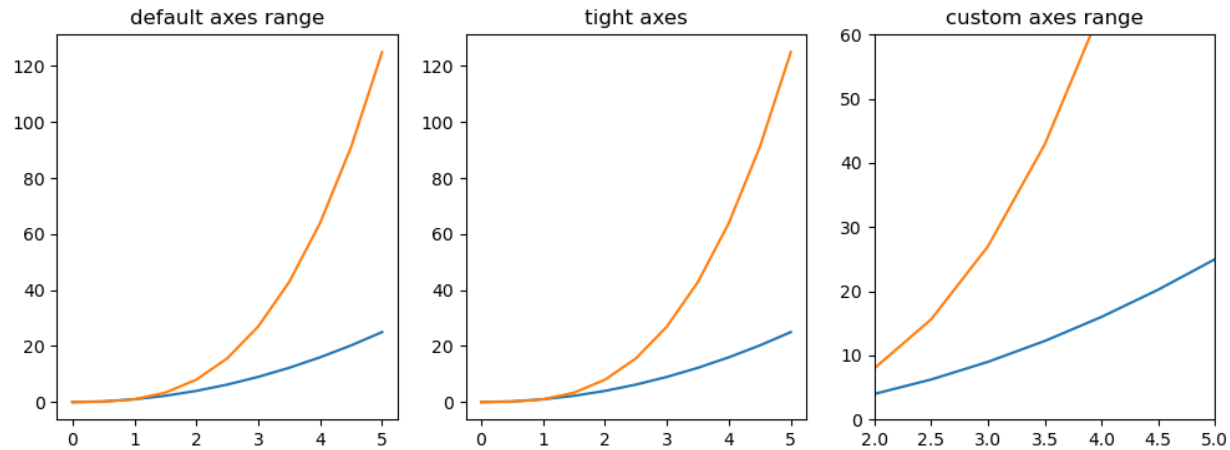
In [18]:
```
fig, ax = plt.subplots(figsize=(12,6))

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 linestype 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, marker
ax.plot(x, x+16, color="purple", lw=1, ls='-', marker='s', markersize=8,
        markerfacecolor="yellow", markeredgewidth=3, markeredgecolor="green");
```



PLOT RANGE I 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 [19]:
```
fig, axes = plt.subplots(1,3, figsize = (12,4))
axes[0].plot(x, x**2, x , x**3)
axes[0].set_title('default axes range')

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")
```

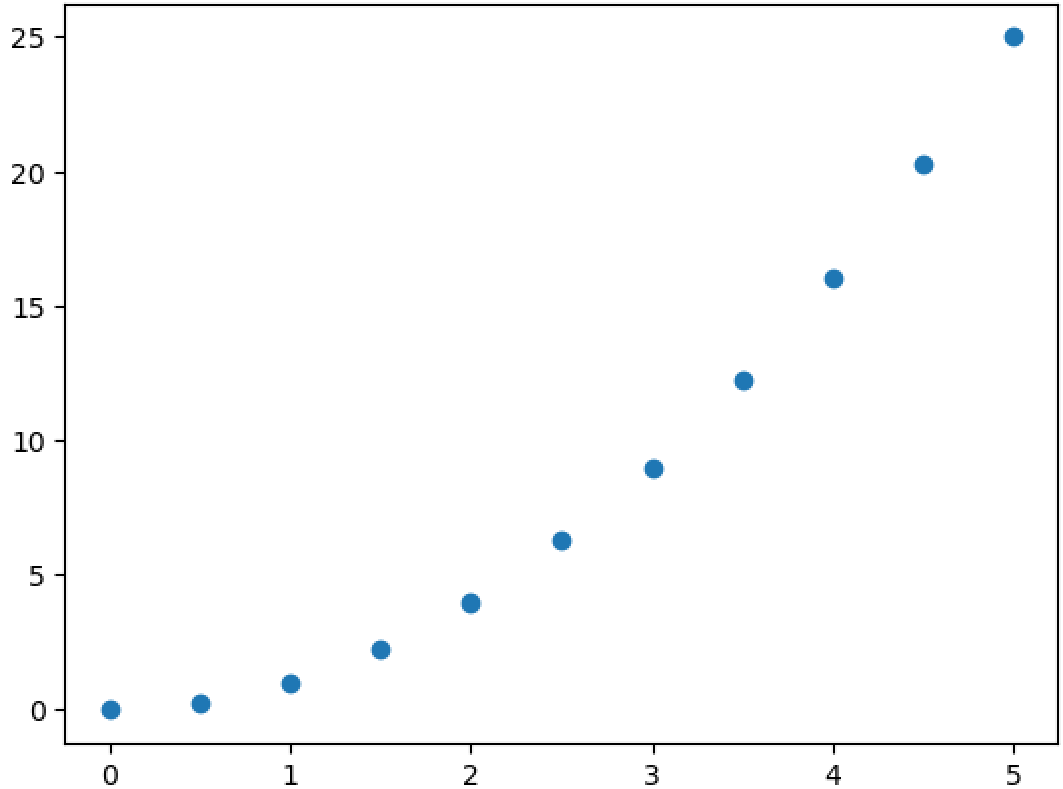
Out[19]:
```
Text(0.5, 1.0, 'custom axes range')
```



SPECIAL PLOT TYPES There are many specialized plots I can create, such as barplots, histograms, scatter plots, and much more.

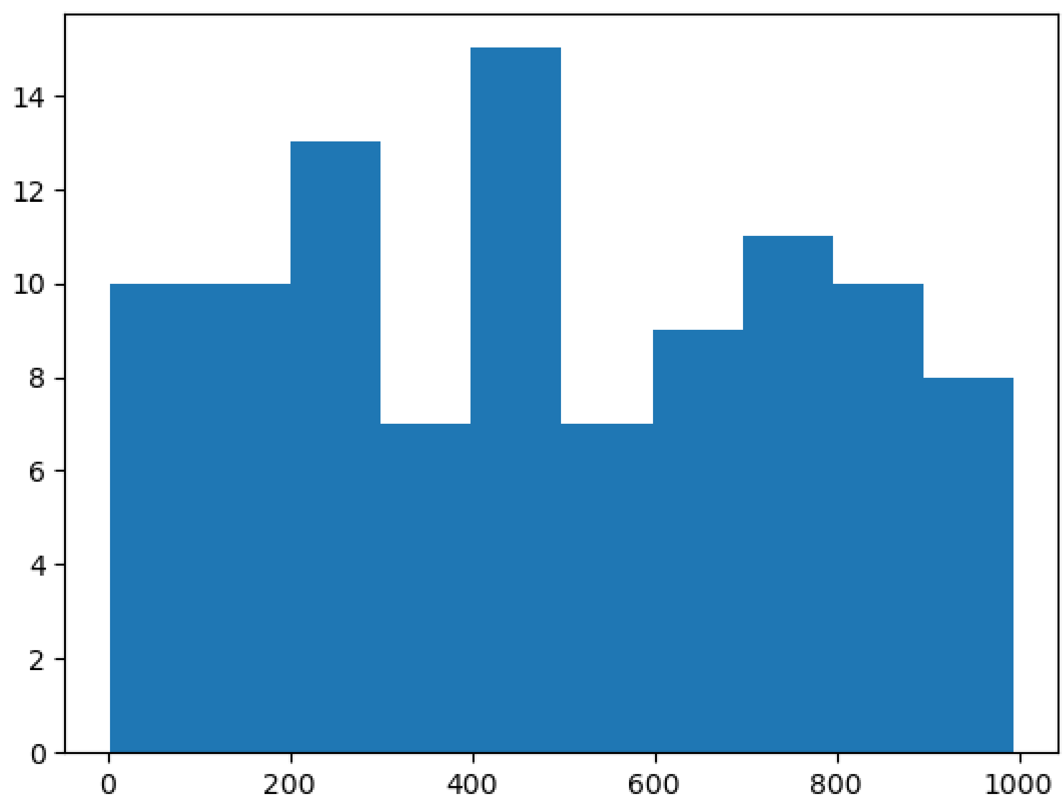
In [20]:
```
plt.scatter(x,y)
```

Out[20]:
```
<matplotlib.collections.PathCollection at 0x1222a9360>
```



In [21]:
```
from random import sample
data = sample(range(1, 1000), 100)
plt.hist(data)
```

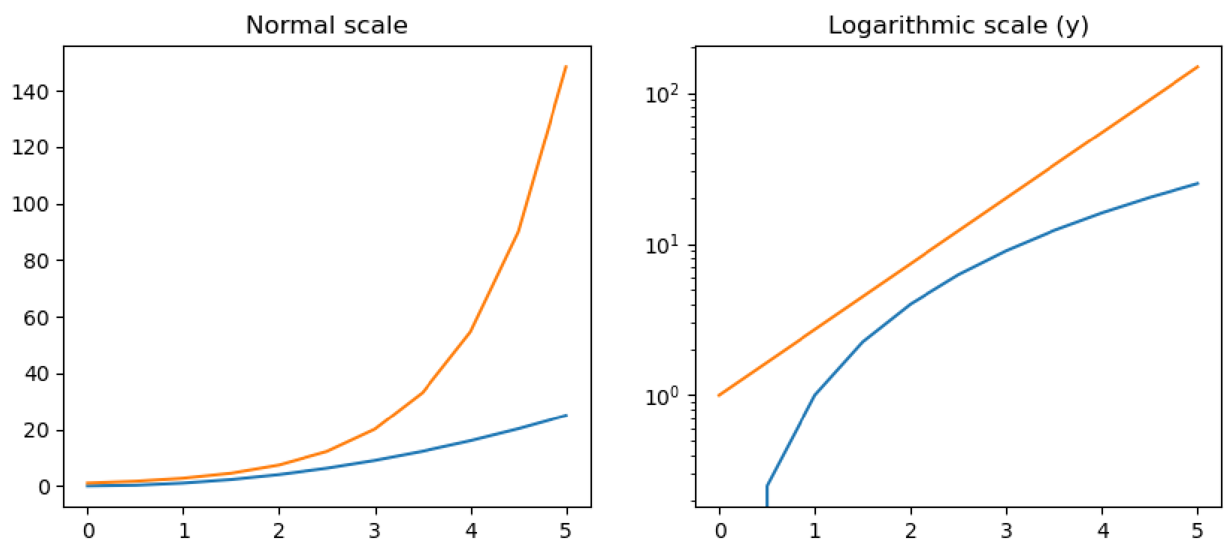
Out[21]:
```
(array([10., 10., 13.,  7., 15.,  7.,  9., 11., 10.,  8.]),
 array([  3. , 102.1, 201.2, 300.3, 399.4, 498.5, 597.6, 696.7, 795.8,
        894.9, 994. ]),
 <BarContainer object of 10 artists>)
```



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

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

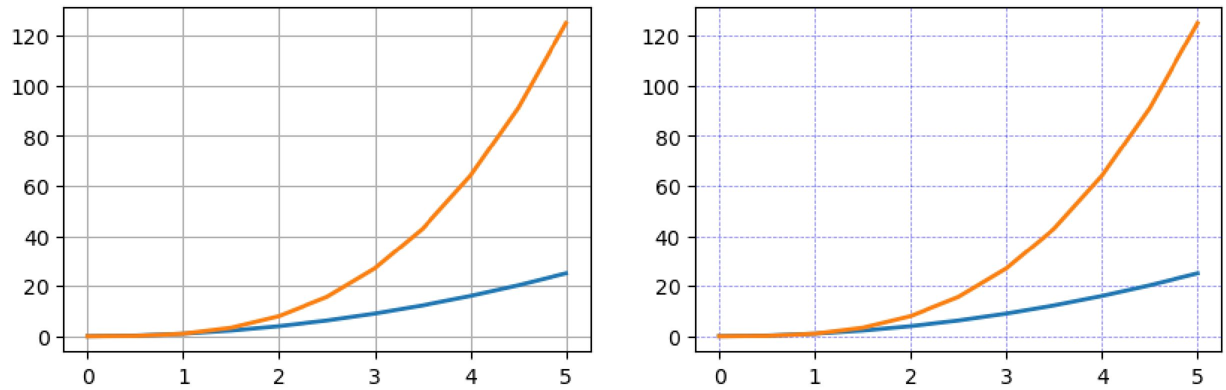
LOGARITHMIC SCALE It is also possible to set a logarithmic scale for one or both axes. This functionality is in fact only one application of a more general transformation system in Matplotlib. Each of the axes' scales are set seperately using set_xscale and set_yscale methods which accept one parameter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

In [22]:
```
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)");
```



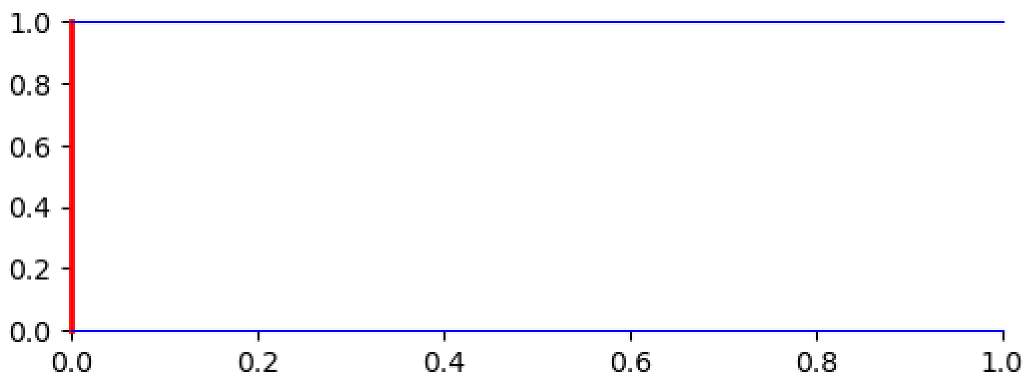
AXIS GRID With the grid method in the axis object, I can turn on and off grid lines. I can also customize the appearance of the grid lines using the same keyword arguments as the plot function:

In [23]:
```
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)
```



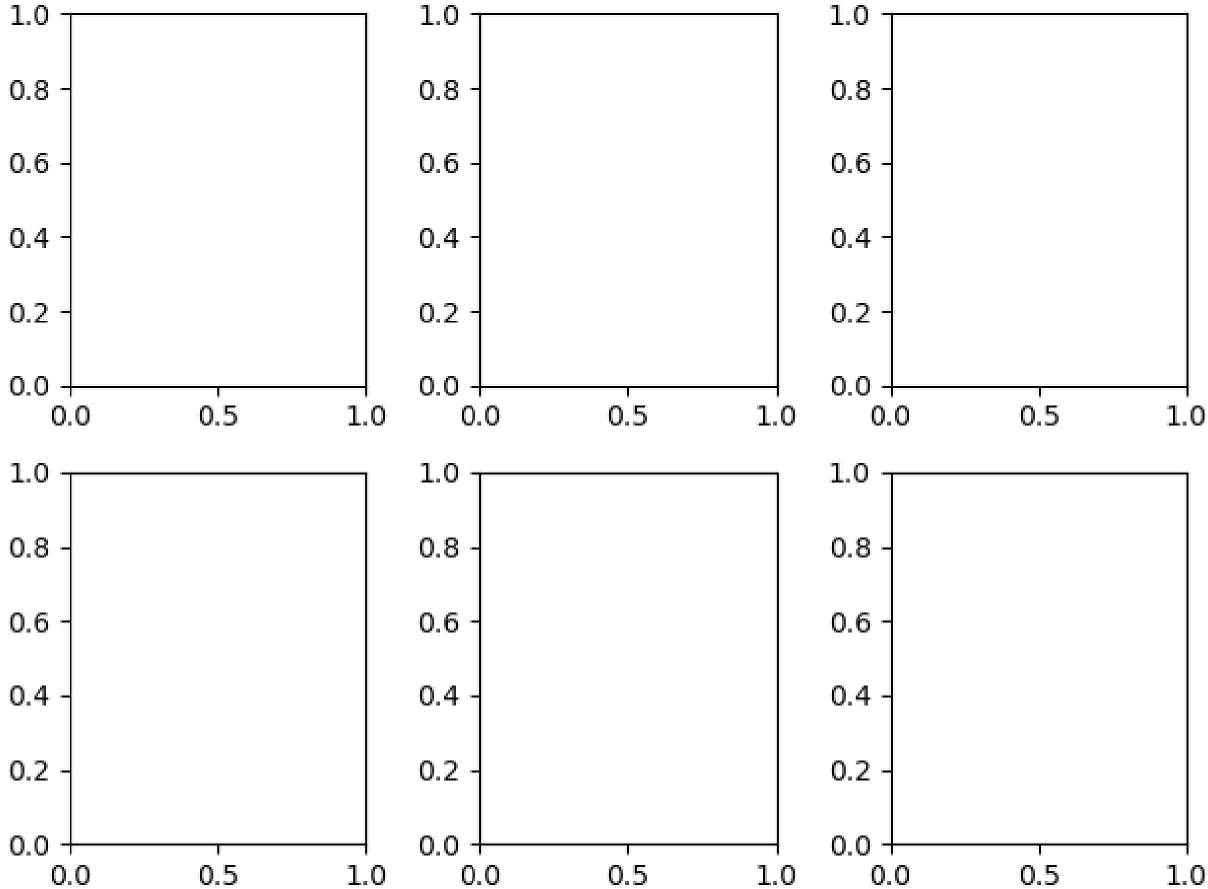
AXIS SPINES

In [24]:
```
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
```

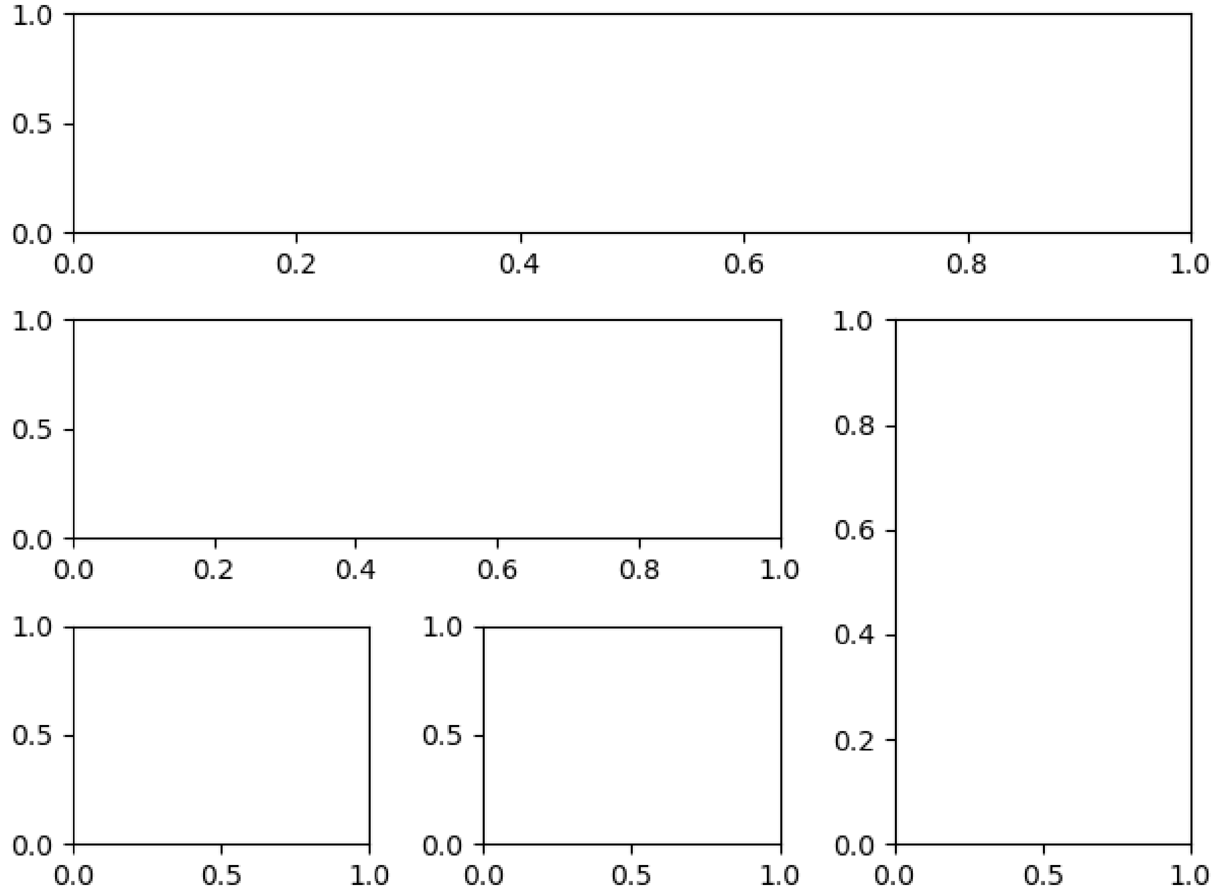


FIGURES WITH MULTIPLE SUBPLOTS AND INSETS Axes can be added to a matplotlib Figure canvas manually using fig.add_axes or using a sub-figure layout manager such as subplots, subplot2grid, or gridspec:

In [29]:
```
fig, ax = plt.subplots(2, 3)
fig.tight_layout()
```



In [30]:
```
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()
```


1.0
0.5
0.0
0.0 0.2 0.4 0.6 0.8 1.0
1.0
0.5
0.0
0.0 0.2 0.4 0.6 0.8 1.0
1.0
0.8
0.6
0.4
0.2
0.0
0.0 0.5 1.0
1.0
0.5
0.0
0.0 0.5 1.0
1.0
0.5
0.0
0.0 0.5 1.0

THANK YOU