Assignments10

matplotlib.pdf

Import Matplotlib

Before, we need to actually start using Matplotlib, we need to import it. We can import Matplotlib as follows:-
`import matplotlib`
Most of the time, we have to work with **pyplot** interface of Matplotlib. So, I will import **pyplot** interface of Matplotlib as follows:
`import matplotlib.pyplot`
To make things even simpler, we will use standard shorthand for Matplotlib imports as follows:-
`import matplotlib.pyplot as plt`

In [202…
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as dp
```

Displaying Plots in Matplotlib

Viewing the Matplotlib plot is context based. The best usage of Matplotlib differs depending on how we are using it. There are three applicable contexts for viewing the plots. The three applicable contexts are using plotting from a script, plotting from an IPython shell or plotting from a Jupyter notebook.

Plotting from a script

If we are using Matplotlib from within a script, then the `plt.show()` command is of great use. It starts an event loop, looks for all currently active figure objects, and opens one or more interactive windows that display the figure or figures.
The `plt.show()` command should be used only once per Python session. It should be used only at the end of the script. Multiple `plt.show()` commands can lead to unpredictable results and should mostly be avoided.

Plotting from a Jupyter notebook

The Jupyter Notebook (formerly known as the IPython Notebook) is a data analysis and visualization tool that provides multiple tools under one roof. It provides code execution, graphical plots, rich text and media display, mathematics formula and much more

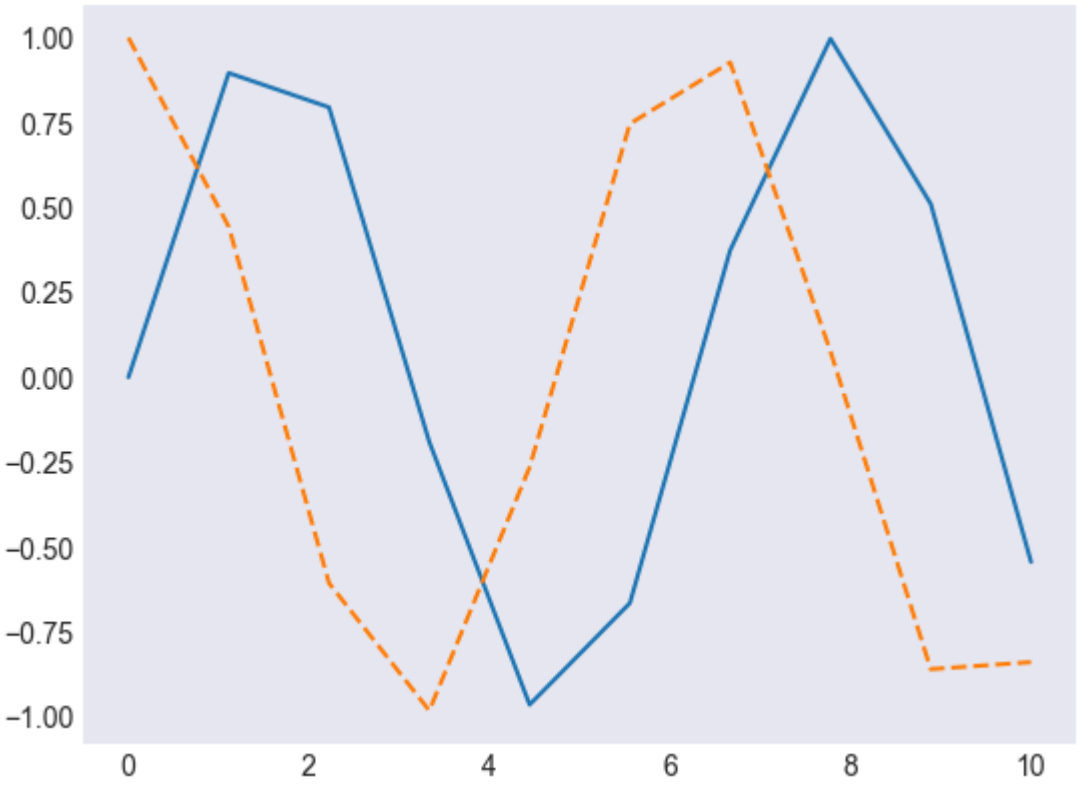
facilities into a single executable document.
Interactive plotting within a Jupyter Notebook can be done with the %matplotlib command. There are two possible options to work with graphics in Jupyter Notebook. These are as follows:-

- `%matplotlib notebook` – This command will produce interactive plots embedded within the notebook.
- `%matplotlib inline` – It will output static images of the plot embedded in the notebook.

After this command (it needs to be done only once per kernel per session), any cell within the notebook that creates a plot will embed a PNG image of the graphic

In [203…
```
%matplotlib inline
#%matplotlib inline is used to display the plot in the notebook itself
x1=np.linspace(0,10,10)
figure1=plt.figure()
plt.plot(x1, np.sin(x1),'-')
plt.plot(x1, np.cos(x1),'--')
```

Out[203…
```
[<matplotlib.lines.Line2D at 0x183c95d36d0>]
```



Matplotlib Object Hierarchy

There is an Object Hierarchy within Matplotlib. In Matplotlib, a plot is a hierarchy of nested Python objects. A **hierarch** means that there is a tree-like structure of Matplotlib objects underlying each plot.

A **Figure** object is the outermost container for a Matplotlib plot. The **Figure** object contain multiple **Axes** objects. So, the Figure is the final graphic that may contain one or more **Axes**. The **Axes** represent an individual plot.

So, we can think of the **Figure** object as a box-like container containing one or more **Axes**. The **Axes** object contain smaller objects such as tick marks, lines, legends, title and text-boxes.

Matplotlib API Overview

Matplotlib has two APIs to work with. A MATLAB-style state-based interface and a more powerful object-oriented (OO) interface. The former MATLAB-style state-based interface is called **pyplot interface** and the latter is called **Object-Oriented** interface.

There is a third interface also called **pylab** interface. It merges pyplot (for plotting) and NumPy (for mathematical functions) together in an environment closer to MATLAB. This is considered bad practice nowadays. So, the use of **pylab** is strongly discouraged and hence, I will not discuss it any further.

Pyplot API

Matplotlib.pyplot provides a MATLAB-style, procedural, state-machine interface to the underlying object-oriented library in Matplotlib. **Pyplot** is a collection of command style functions that make Matplotlib work like MATLAB. Each pyplot function makes some change to a figure - e.g., creates a figure, creates a plotting area in a figure etc.

Matplotlib.pyplot is stateful because the underlying engine keeps track of the current figure and plotting area information and plotting functions change that information. To make it clearer, we did not use any object references during our plotting we just issued a pyplot command, and the changes appeared in the figure.
We can get a reference to the current figure and axes using the following commands-

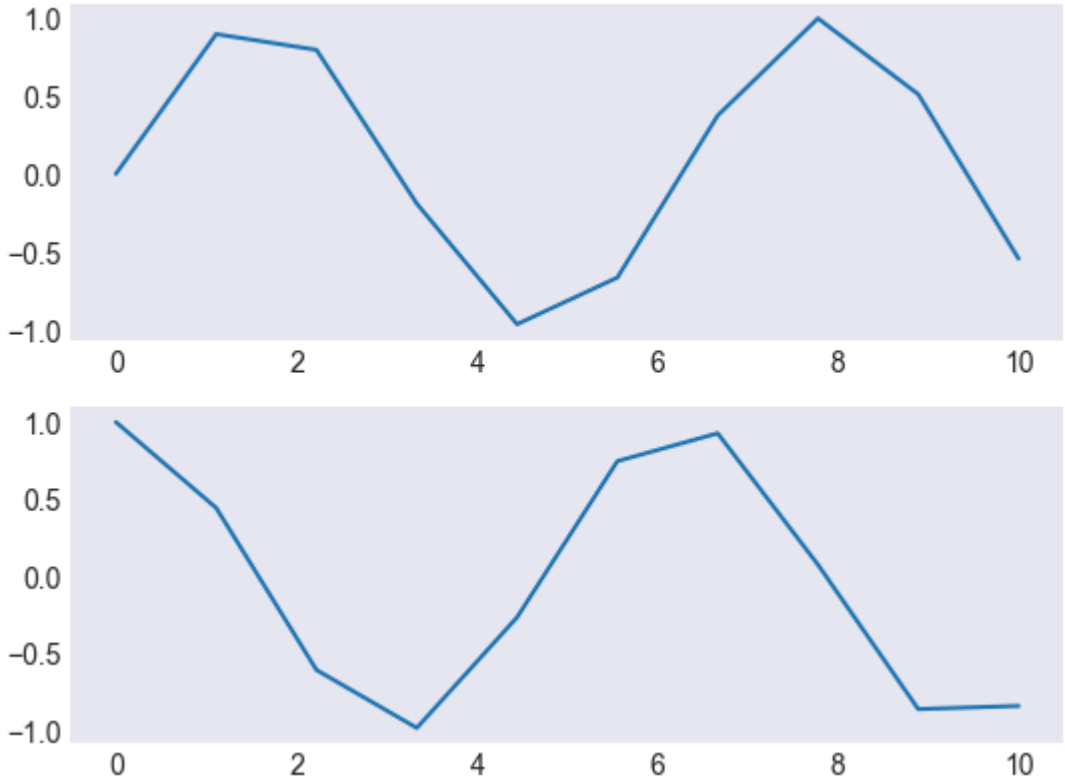
`plt.gcf ( )` # get current figure
`plt.gca ( )` # get current axes
Matplotlib.pyplot is a collection of commands and functions that make Matplotlib behave like MATLAB (for plotting). The MATLAB-style tools are contained in the pyplot (plt) interface.
This is really helpful for interactive plotting, because we can issue a command and see the result immediately. But, it is not suitable for more complicated cases. For these cases, we have another interface called **Object-Oriented interface**, described later.

In [244…

```
plt.figure()
plt.subplot(2,1,1)
plt.plot(x1,np.sin(x1),'-')
```

```
plt.subplot(2,1,2)
plt.plot(x1,np.cos(x1),'-')
```

Out[244… [<matplotlib.lines.Line2D at 0x183c9c7a3d0>]

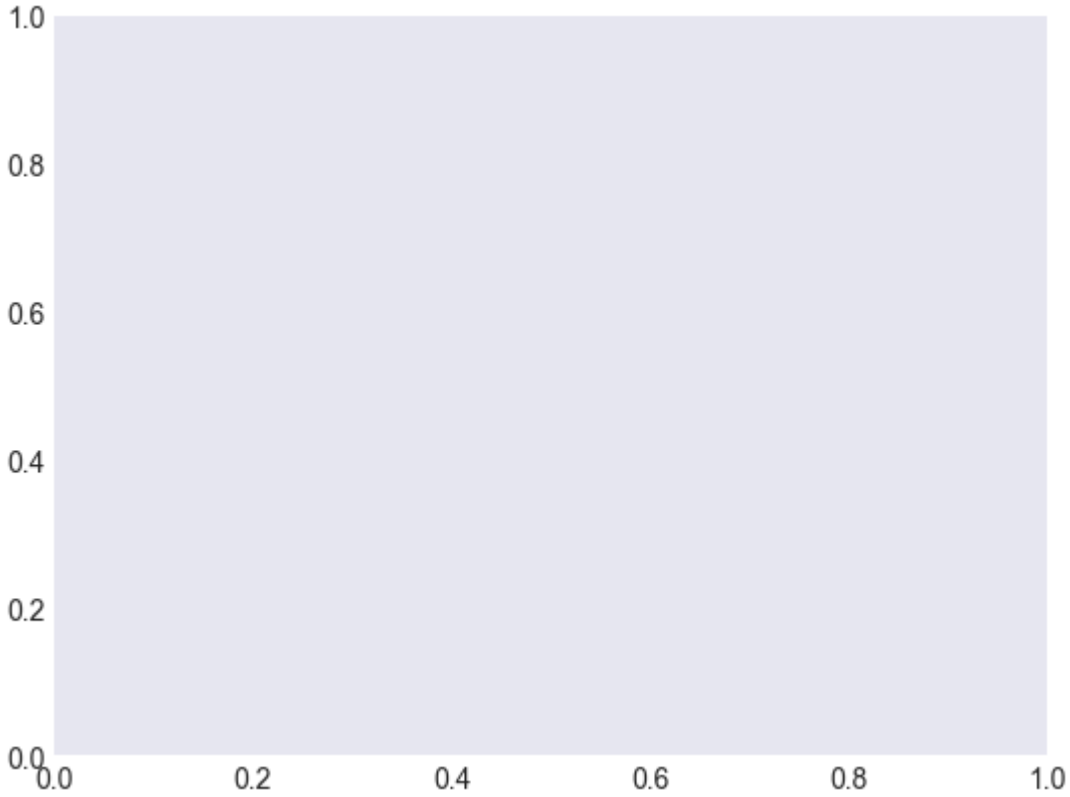


In [246…
```
print(plt.gcf())#cd() is used to get current figure
```

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

In [206…
```
print(plt.gca())#gca() is used to get the current axis
```

```
Axes(0.125,0.11;0.775x0.77)
```

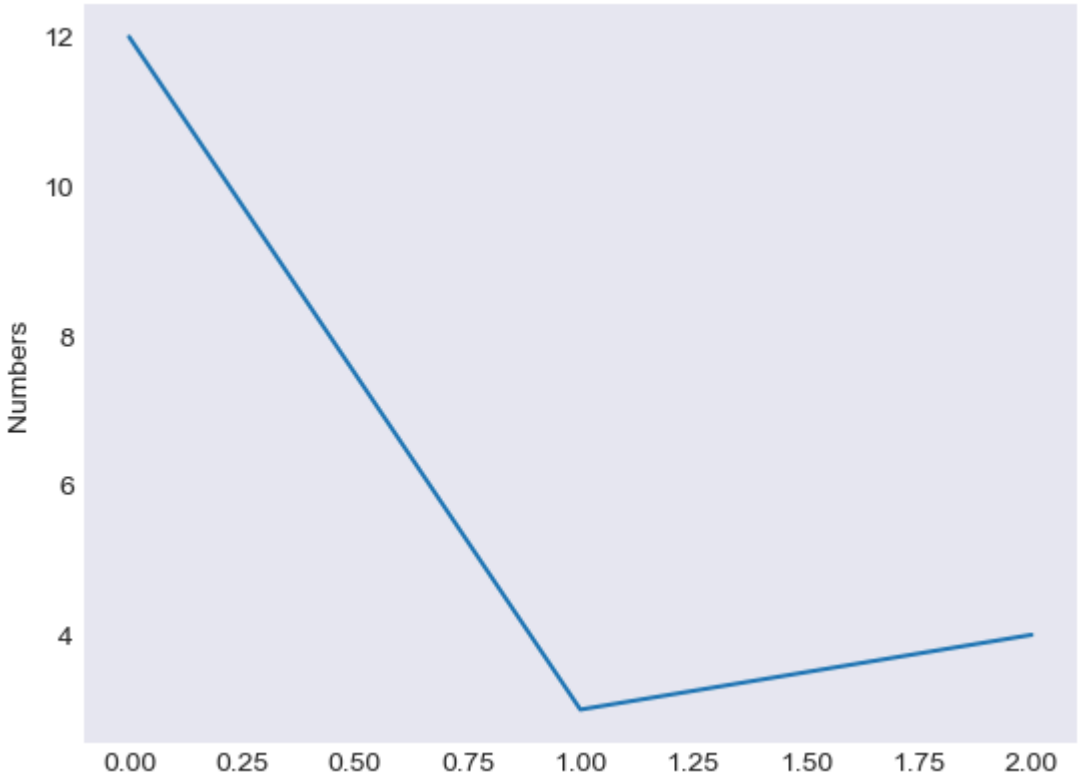


Visualization with Pyplot

Generating visualization with Pyplot is very easy. The x-axis values ranges from 0-3 and the y-axis from 1-4. If we provide a single list or array to the plot() command, matplotlib assumes it is a sequence of y values, and automatically generates the x values. Since python ranges start with 0, the default x vector has the same length as y but starts with 0. Hence the x data are [0,1,2,3] and y data are [1,2,3,4].

In [136…

```
plt.plot([12,3,4])
plt.ylabel('Numbers')
plt.show()
```

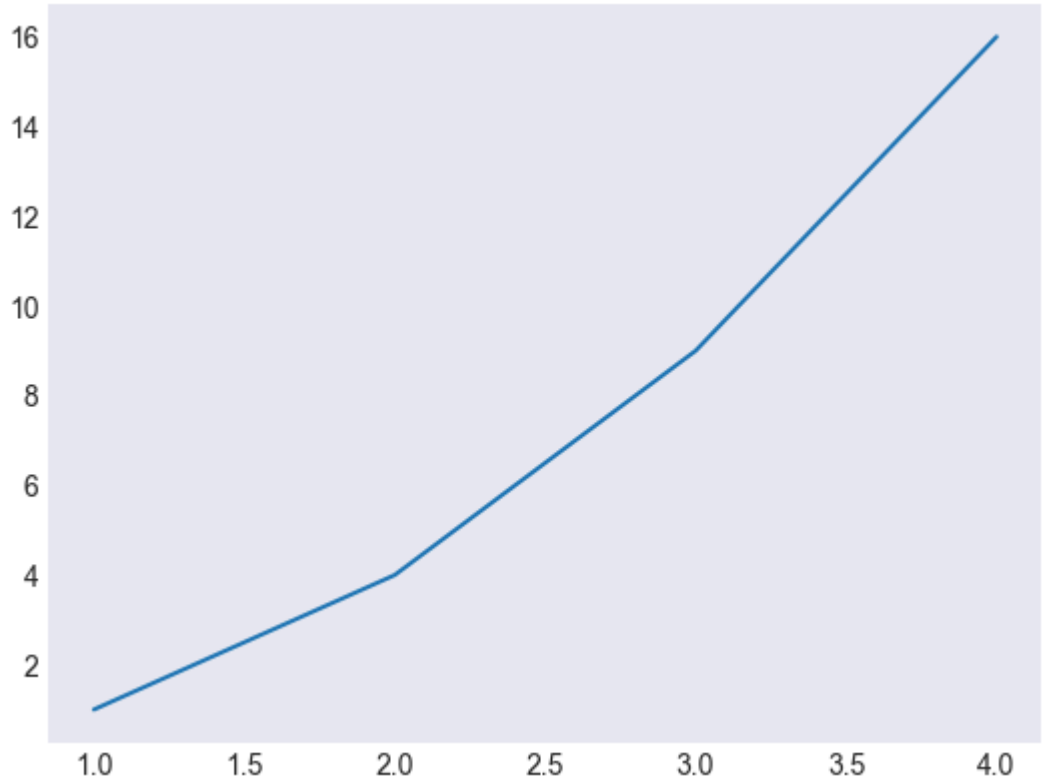


plot() - A versatile command

plot() is a versatile command. It will take an arbitrary number of arguments. For example, to plot x versus y, we can issue the following command:-

In [208…

```
plt.plot([1,2,3,4],[1,4,9,16])
plt.show()
```

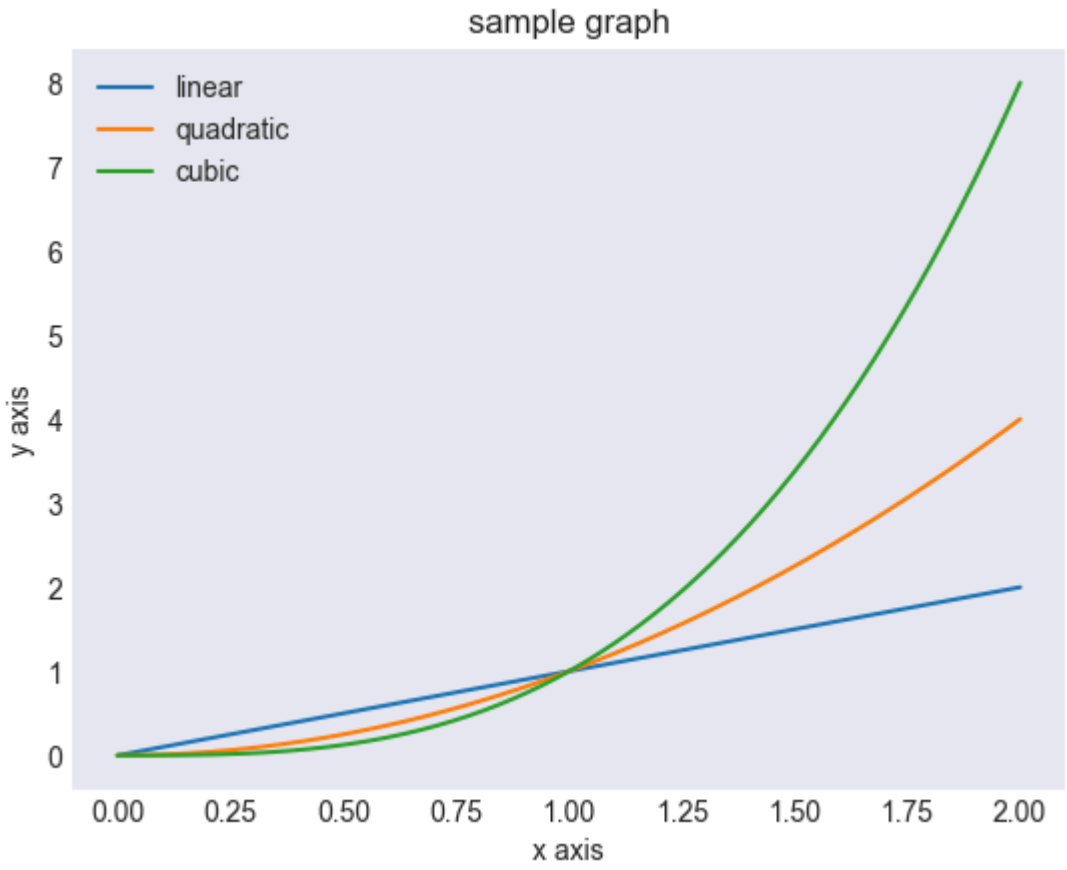


State-machine interface

Pyplot provides the state-machine interface to the underlying object-oriented plotting library. The state-machine implicitly and automatically creates figures and axes to achieve the desired plot. For example:

In [209…

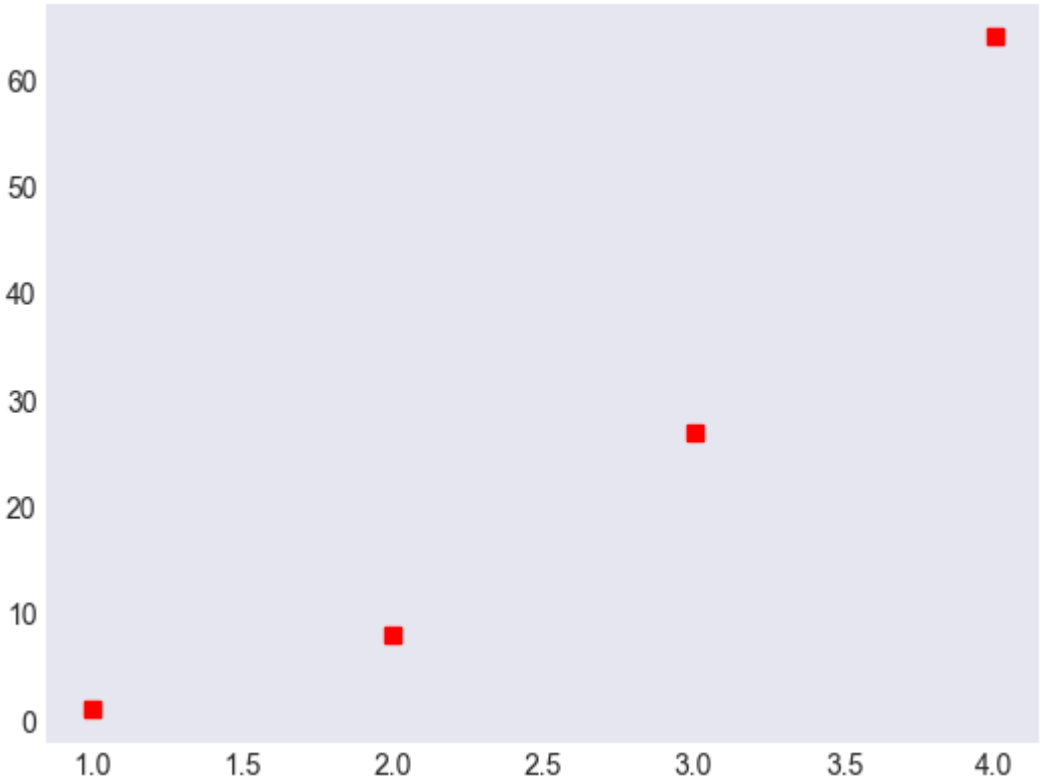
```
x = np.linspace(0,2,1000)
plt.plot(x, x, label = 'linear')
plt.plot(x, x**2, label = 'quadratic')
plt.plot(x,x**3, label = 'cubic')
plt.xlabel('x axis')
plt.ylabel('y axis')
plt.title('sample graph')
plt.legend()
plt.show()
```



Formatting the style of plot

For every x, y pair of arguments, there is an optional third argument which is the format string that indicates the color and line type of the plot. The letters and symbols of the format string are from MATLAB. We can concatenate a color string with a line style string. The default format string is 'b-', which is a solid blue line. For example, to plot the above line with red circles, we would issue the following command:-

In [210…
```
plt.plot([1,2,3,4],[1,8,27,64], 'rs')
plt.show()
```

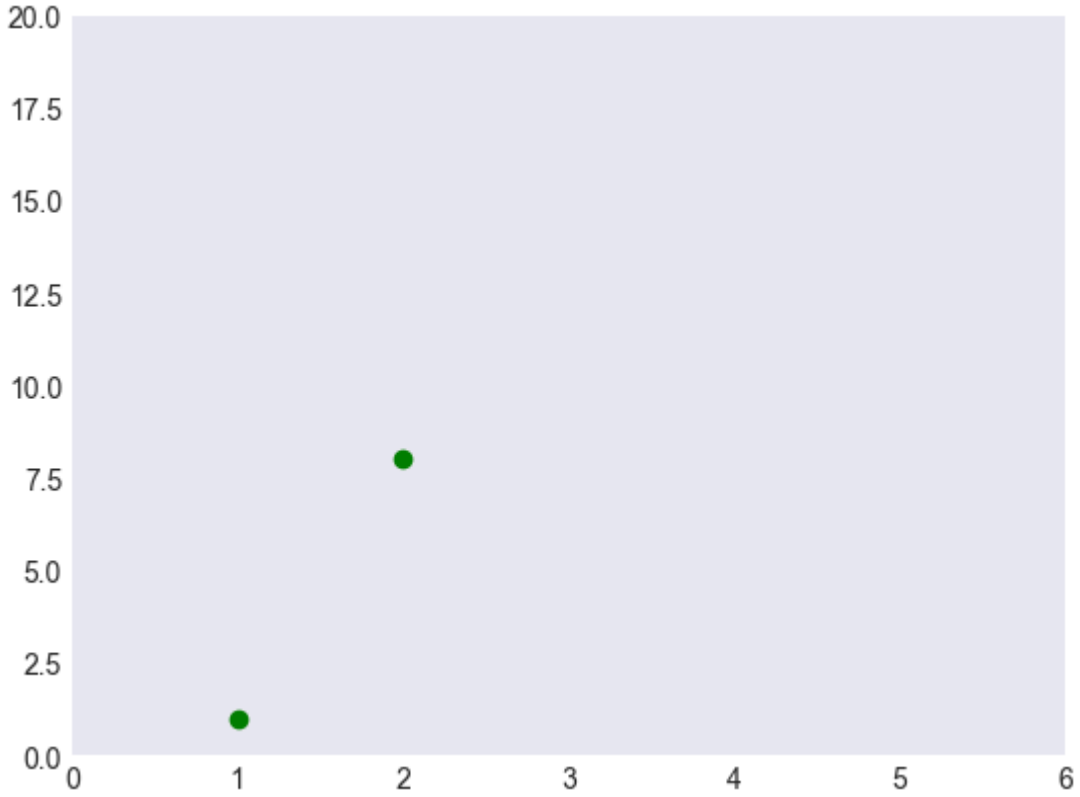


The **axis()** command in the example above takes a list of [xmin, xmax, ymin, ymax] and specifies the viewport of the axes.

Working with NumPy arrays

Generally, we have to work with NumPy arrays. All sequences are converted to numpy arrays internally. The below example illustrates plotting several lines with different format styles in one command using arrays.

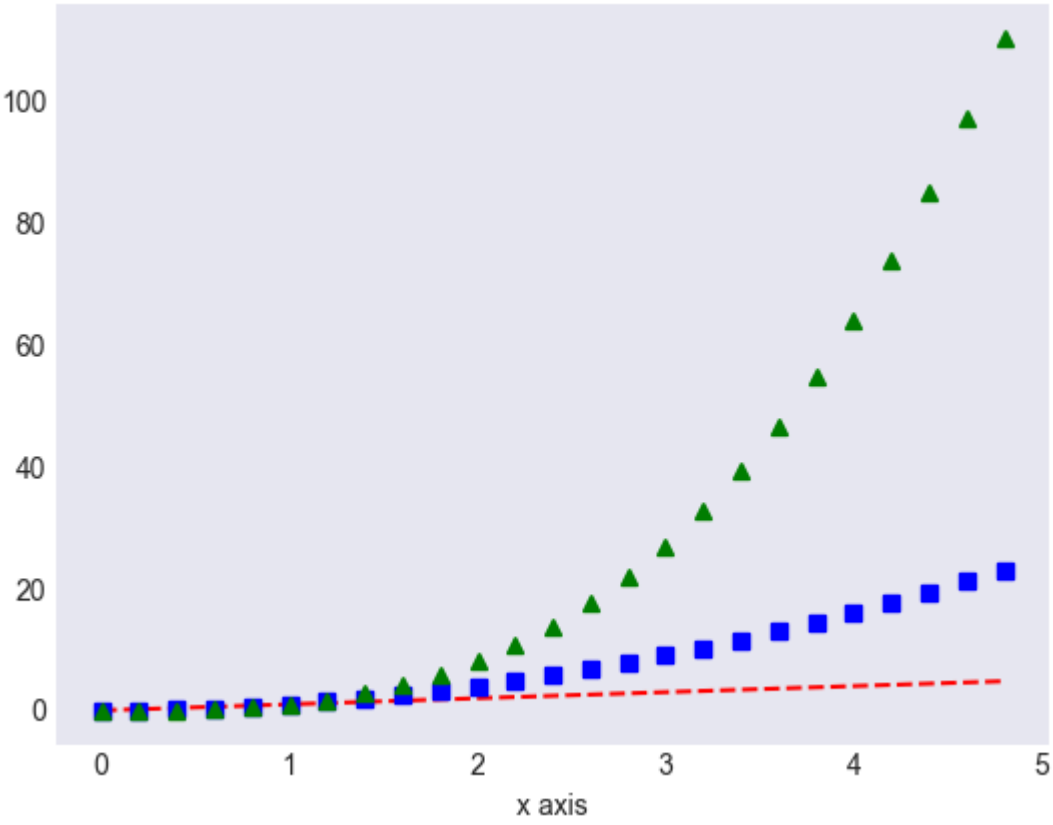
In [211…
```
plt.plot([1,2,3,4],[1,8,27,64], 'go')
plt.axis([0, 6, 0, 20])
plt.show()
```



In [212…

```
t2 = np.arange(0.,5.,0.2)
plt.plot(t2,t2,'r--',t2,t2**2,'bs',t2,t2**3,'g^')
plt.xlabel('x axis')
plt.legend()
plt.show()
```

```
C:\Users\User\AppData\Local\Temp\ipykernel_43620\713951040.py:4: UserWarning: 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.
  plt.legend()
```

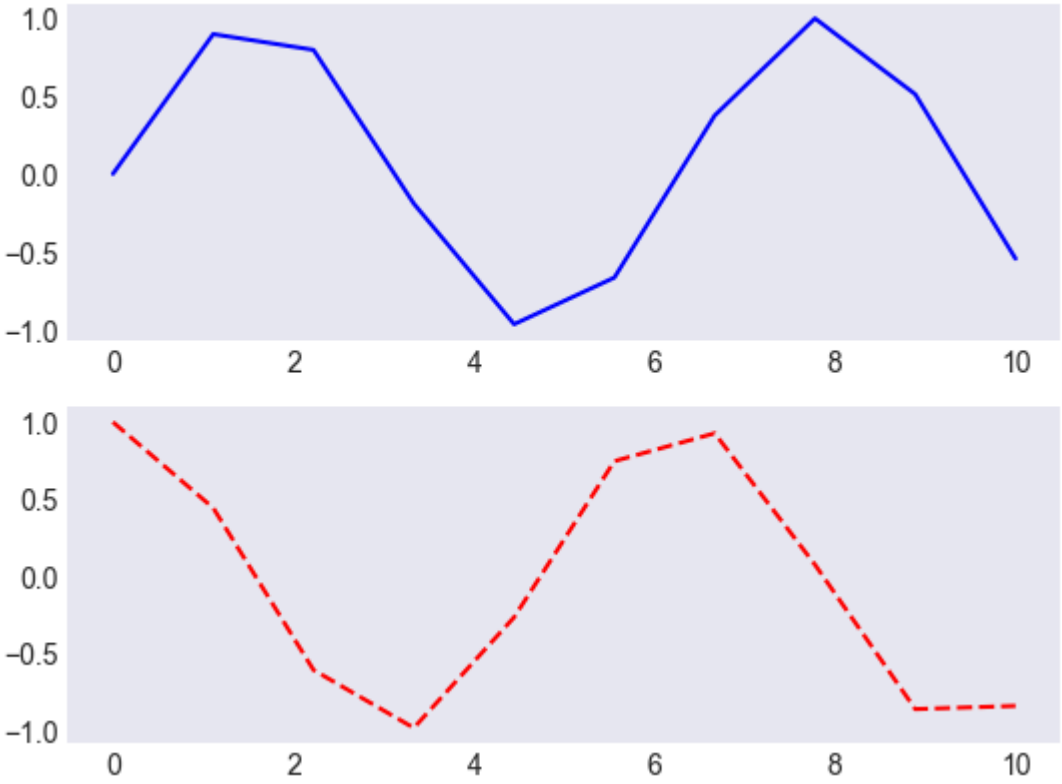


Object-Oriented API

The **Object-Oriented API** is available for more complex plotting situations. It allows us to exercise more control over the figure. In Pyplot API, we depend on some notion of an "active" figure or axes. But, in the **Object-Oriented API** the plotting functions are methods of explicit Figure and Axes objects.
Figure is the top level container for all the plot elements. We can think of the **Figure** object as a box-like container containing one or more **Axes**.
The **Axes** represent an individual plot. The **Axes** object contain smaller objects such as axis, tick marks, lines, legends, title and text-boxes.
The following code produces sine and cosine curves using Object-Oriented API.

In [213…
```
fig, ax = plt.subplots(2)# when 1 number is passed, it refers to no of rows
ax[0].plot(x1, np.sin(x1), 'b-')
ax[1].plot(x1, np.cos(x1),'r--')
```

Out[213…
```
[<matplotlib.lines.Line2D at 0x183c9843150>]
```



objects and Reference

The main idea with the **Object Oriented API** is to have objects that one can apply functions and actions on. The real advantage of this approach becomes apparent when more than one figure is created or when a figure contains more than one subplot.
We create a reference to the figure instance in the **fig** variable. Then, we ceate a new axis instance **axes** using the **add_axes** method in the Figure class instance fig as follows:-

In [214…
```
fig=plt.figure()
x2 = np.linspace(0,5,50)
y2=x2**2
ax=fig.add_axes([0.1,0.1,0.8,0.8])
ax.plot(x2,y2,'m')
ax.set_xlabel('x2')
ax.set_ylabel('y2')
ax.set_title('title')
```

Out[214…
```
Text(0.5, 1.0, 'title')
```

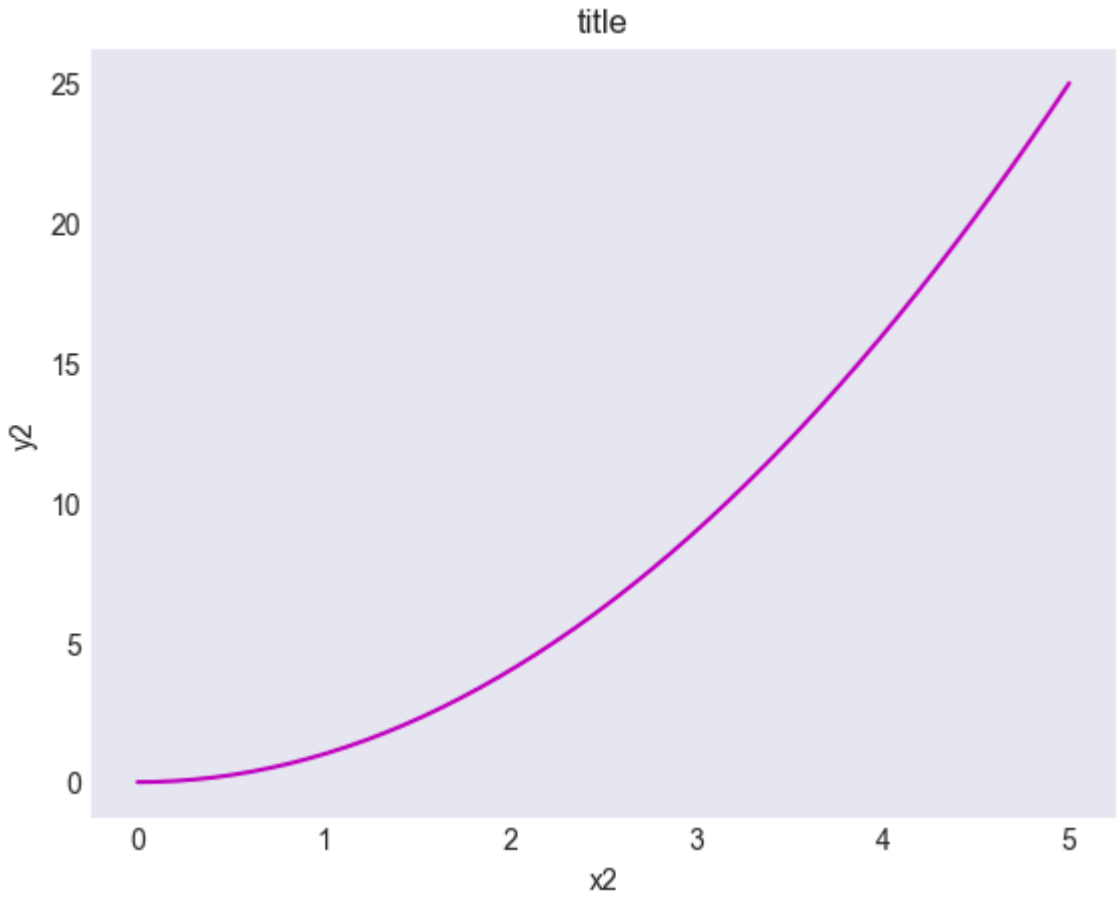
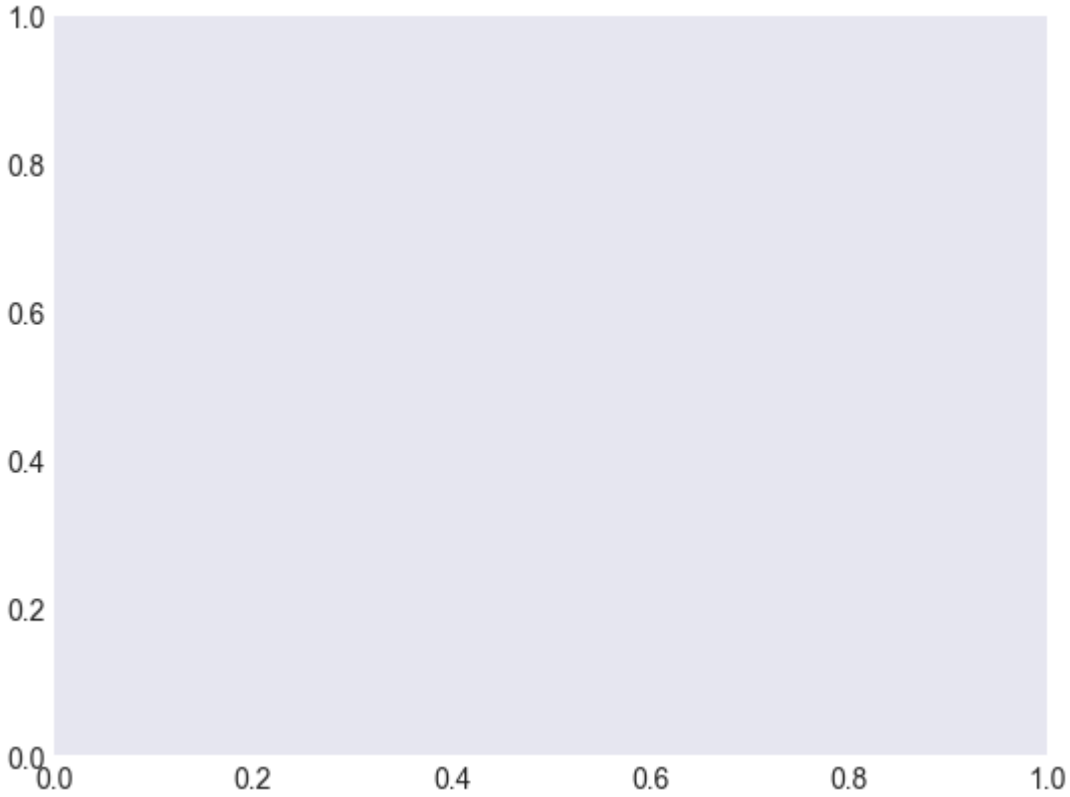


Figure and Axes

I start by creating a figure and an axes. A figure and axes can be created as follows:
`fig = plt.figure()`
`ax = plt.axes()`
In Matplotlib, the **figure** (an instance of the class plt.Figure) is a single container that contains all the objects representing axes, graphics, text and labels. The **axes** (an instance of the class plt.Axes) is a bounding box with ticks and labels. It will contain the plot elements that make up the visualization. I have used the variable name fig to refer to a figure instance, and ax to refer to an axes instance or group of axes instances.

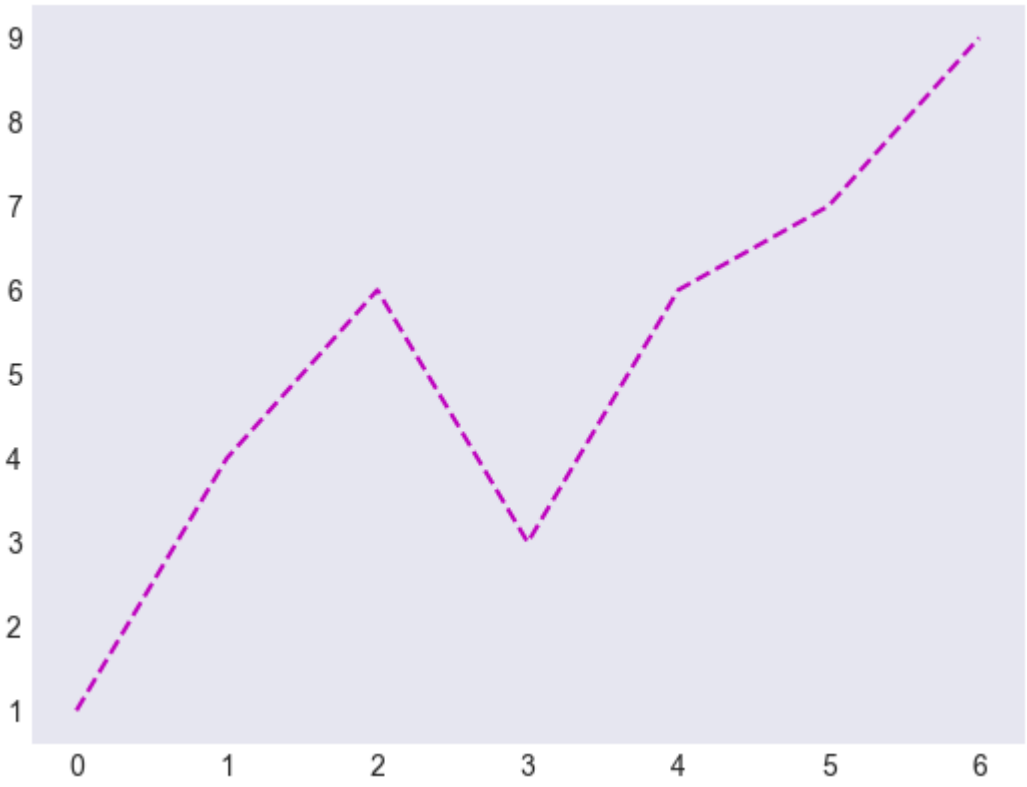
In [215…
```
fig = plt.figure() #will plot the figure
ax = plt.axes() #will plot the axes
plt.show()
```



First plot with Matplotlib

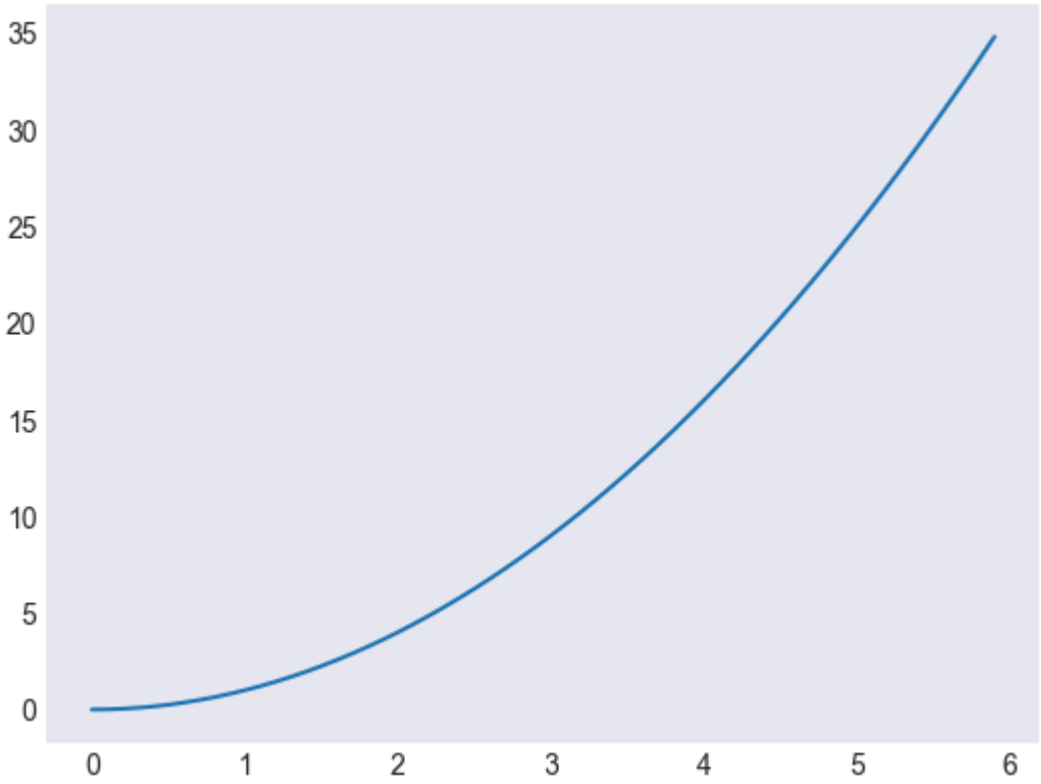
In [216…

```
plt.plot([1,4,6,3,6,7,9],'m--')
plt.show()
```



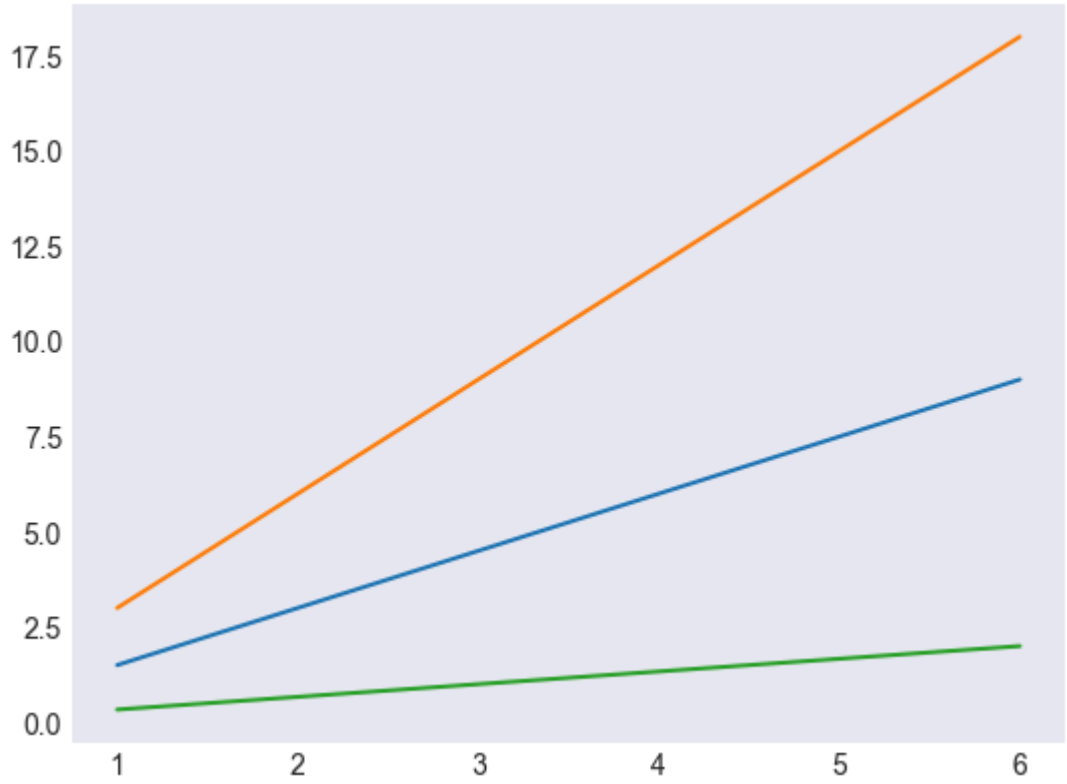
specifying both the lists

In [217…
```
x3=np.arange(0,6,0.1)
plt.plot(x3,[xi**2 for xi in x3])
plt.show()
```



Multiline Plots

In [218…
```
x4 = range(1,7)
plt.plot(x4,[xi*1.5 for xi in x4])
plt.plot(x4,[xi*3 for xi in x4])
plt.plot(x4,[xi/3 for xi in x4])
plt.show()
```

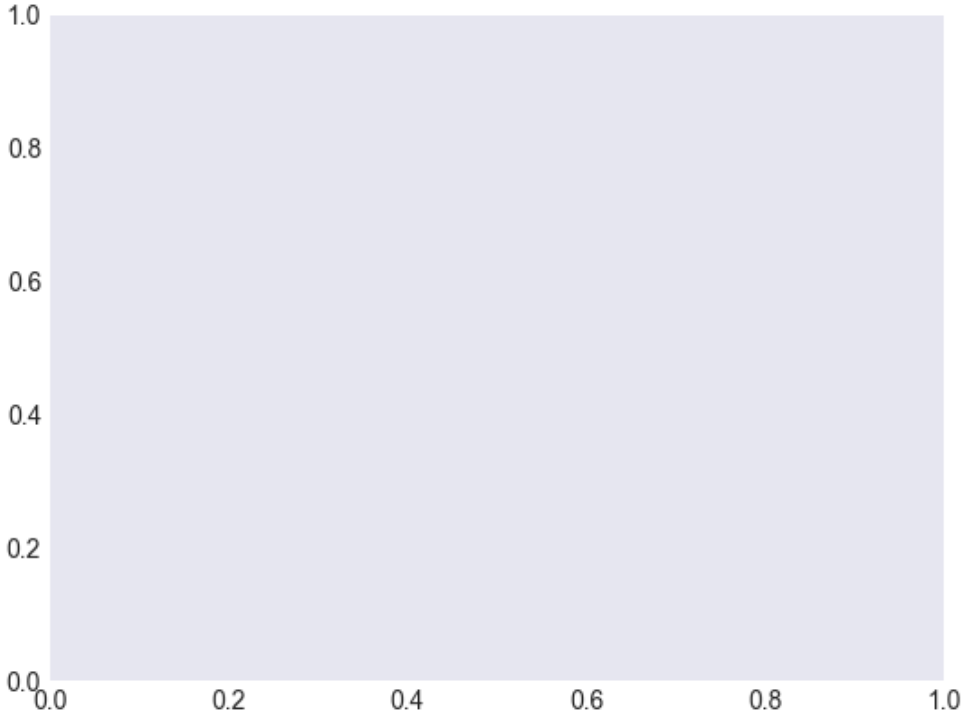


Saving the plot

In [219…
```
fig.savefig('plot1.png')
```

In [220…
```
from IPython.display import Image
Image('plot1.png')
```

Out[220…



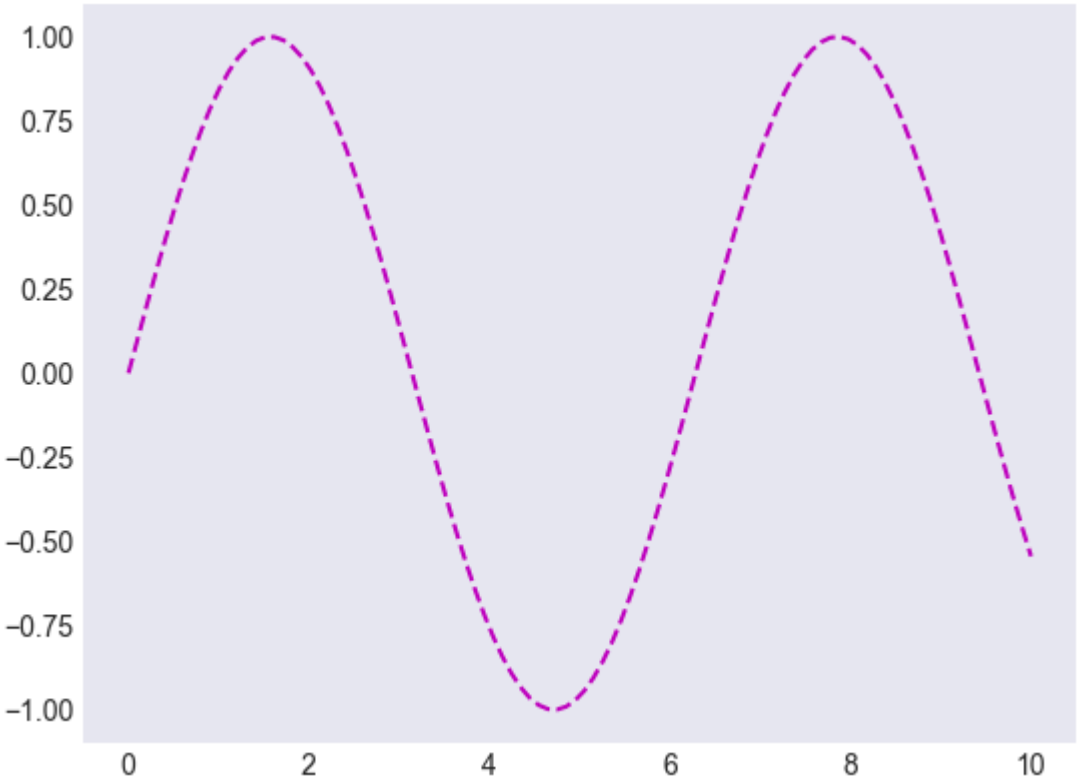
In [221…
```
fig.canvas.get_supported_filetypes()
```

Out[221…
```
{'eps': 'Encapsulated Postscript',
 'jpg': 'Joint Photographic Experts Group',
 'jpeg': 'Joint Photographic Experts Group',
 'pdf': 'Portable Document Format',
 'pgf': 'PGF code for LaTeX',
 'png': 'Portable Network Graphics',
 'ps': 'Postscript',
 'raw': 'Raw RGBA bitmap',
 'rgba': 'Raw RGBA bitmap',
 'svg': 'Scalable Vector Graphics',
 'svgz': 'Scalable Vector Graphics',
 'tif': 'Tagged Image File Format',
 'tiff': 'Tagged Image File Format',
 'webp': 'WebP Image Format'}
```

Line Plot

In [222…
```
fig=plt.figure()
ax = plt.axes()
x5=np.linspace(0,10,100)
ax.plot(x5,np.sin(x5),'m--')
```

Out[222…
```
[<matplotlib.lines.Line2D at 0x183c7d4f150>]
```



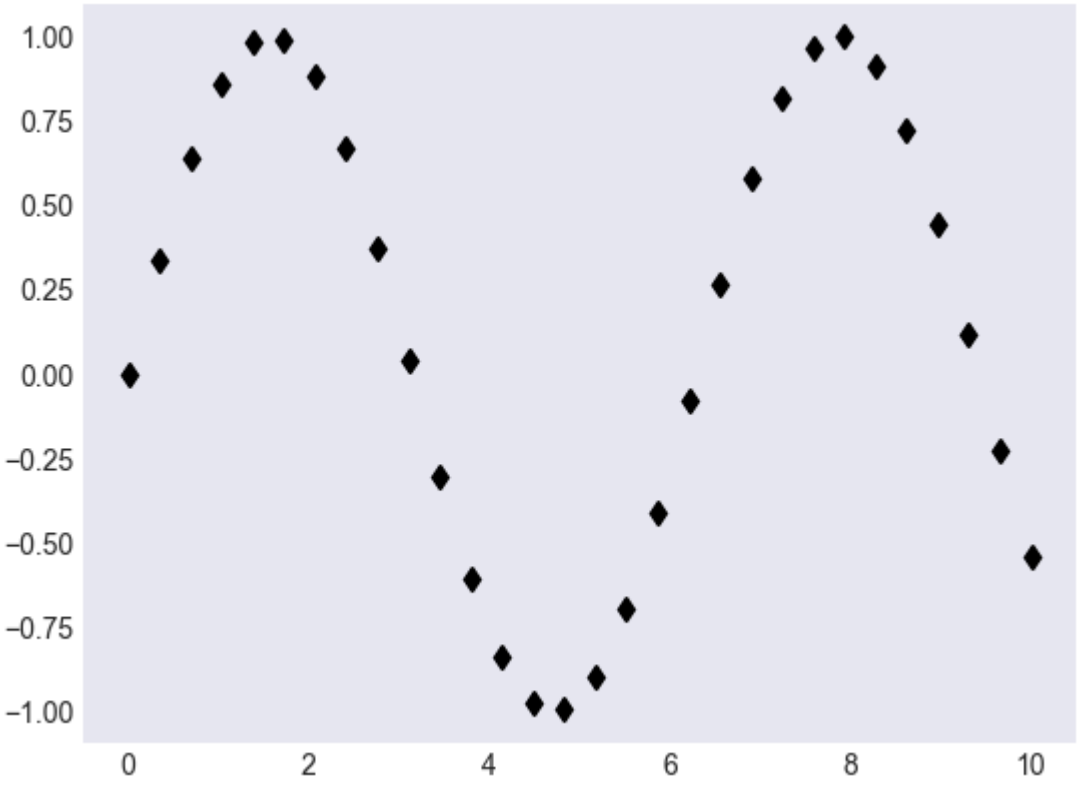
Scatter plot

In [223…
```
x7 = np.linspace(0,10,30)
y7=np.sin(x7)
```

```
plt.plot(x7,y7,'d', color='black')
```

Out[223…

```
[<matplotlib.lines.Line2D at 0x183c977b710>]
```



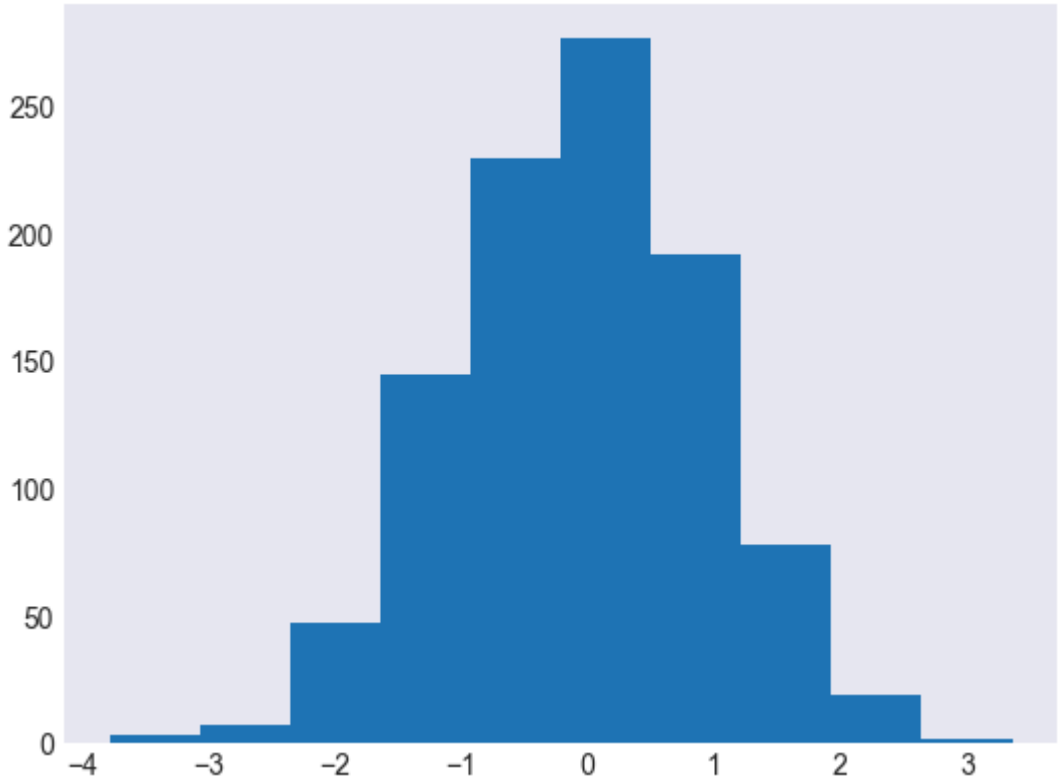
Histogram

In [224…

```
d1=np.random.randn(1000)
plt.hist(d1)
```

Out[224…

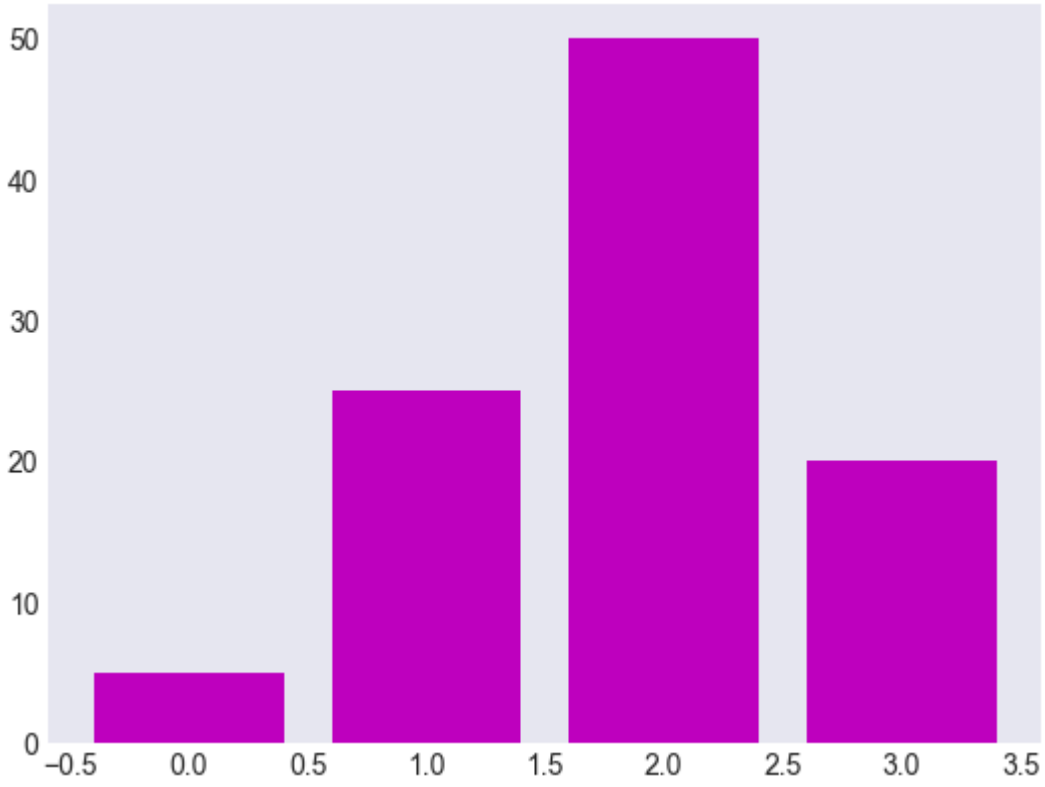
```
(array([  3.,   7.,  47., 145., 230., 277., 192.,  78.,  19.,   2.]),
 array([-3.77997749, -3.06631598, -2.35265448, -1.63899297, -0.92533146,
        -0.21166995,  0.50199156,  1.21565307,  1.92931458,  2.64297608,
         3.35663759]),
 <BarContainer object of 10 artists>)
```



Bar Chart

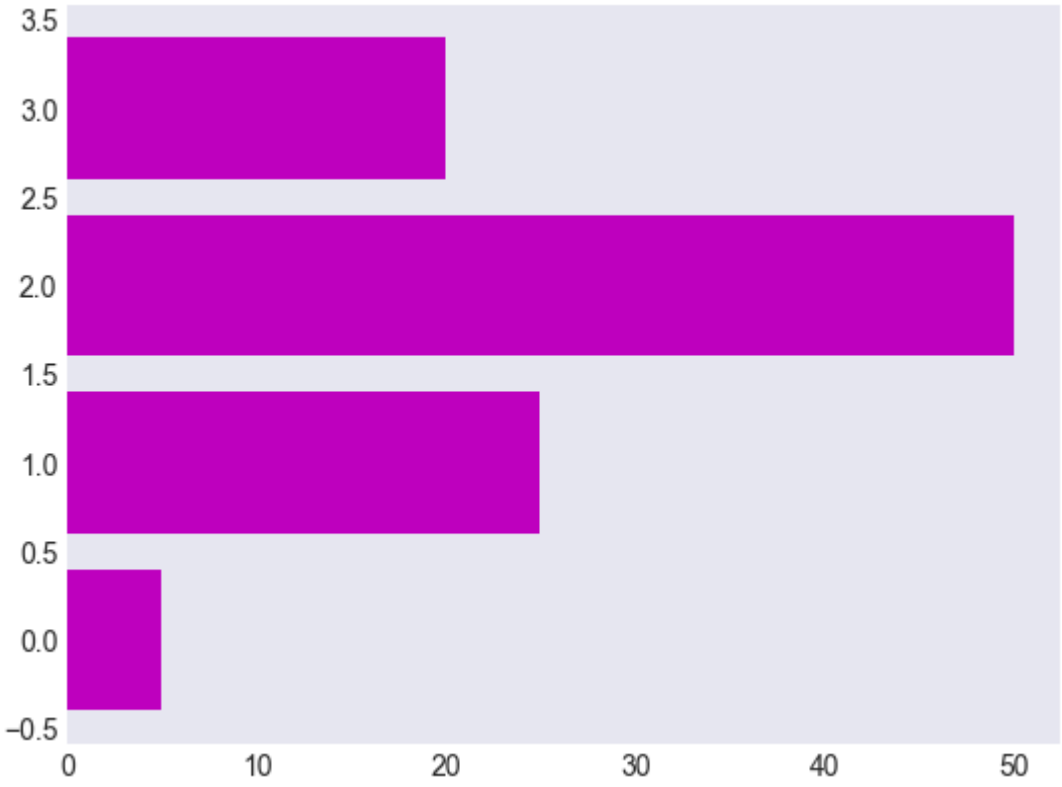
In [225…

```
d2=[5,25,50,20]
plt.bar(range(len(d2)),d2,color='m')
plt.show()
```



Horizontal Bar Chart

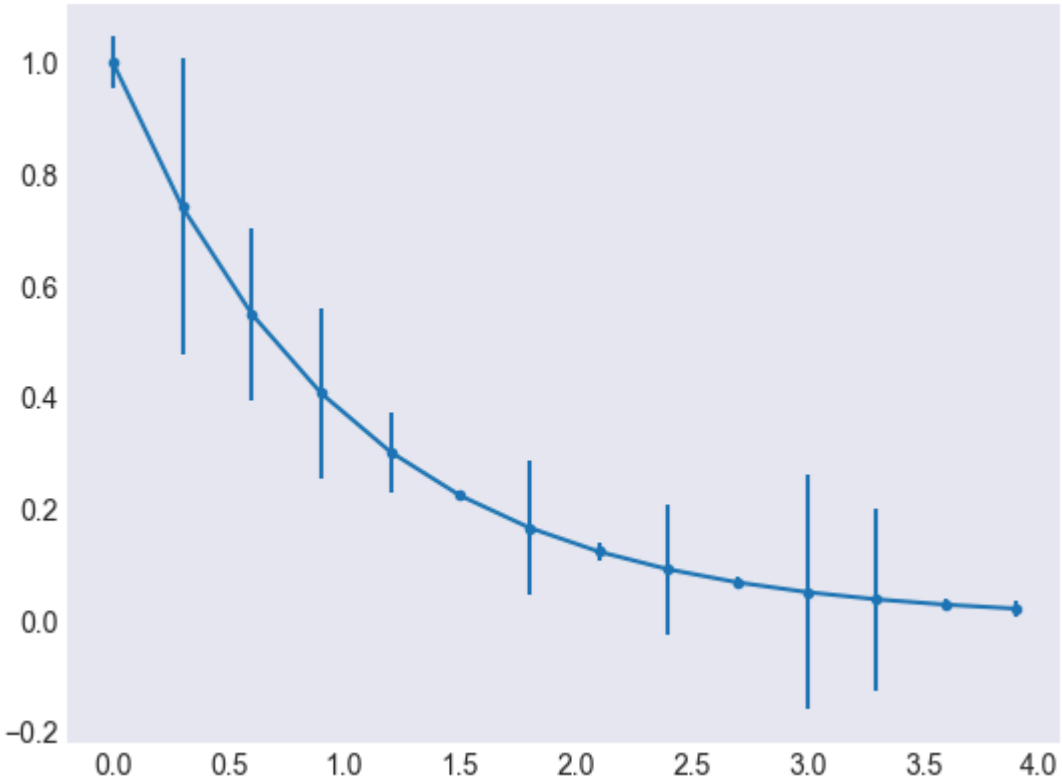
In [226…
```
d2=[5,25,50,20]
plt.barh(range(len(d2)),d2,color='m')
plt.show()
```



Error Bar Chart

In experimental design, the measurements lack perfect precision. So, we have to repeat the measurements. It results in obtaining a set of values. The representation of the distribution of data values is done by plotting a single data point (known as mean value of dataset) and an error bar to represent the overall distribution of data.
We can use Matplotlib's **errorbar()** function to represent the distribution of data values. It can be done as follows:-

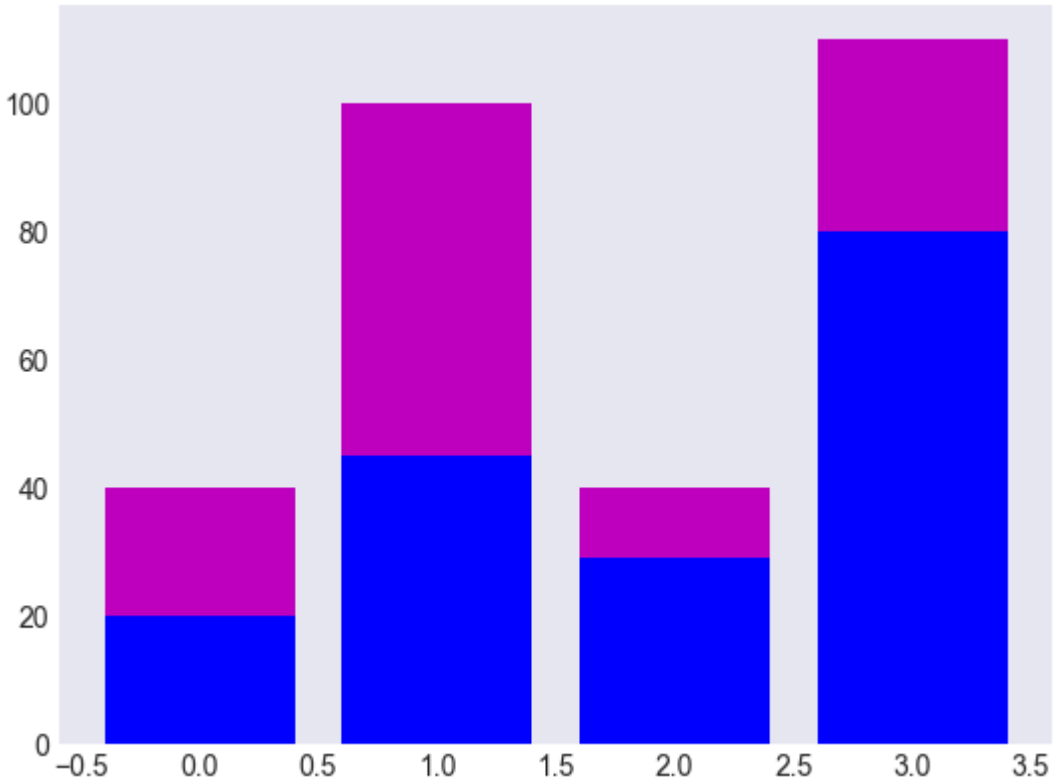
In [227…
```
x = np.arange(0,4,0.3)
y=np.exp(-x)
eb=0.1*np.abs(np.random.randn(len(x)))
plt.errorbar(x,y,yerr=eb, fmt='.-')
plt.show()
```



Stacked Bar Chart

In [228…

```
a=[20,45,29,80]
b=[20,55,11,30]
z=range(4)
plt.bar(z,a,color='b')
plt.bar(z,b,color='m', bottom=a)
plt.show()
```



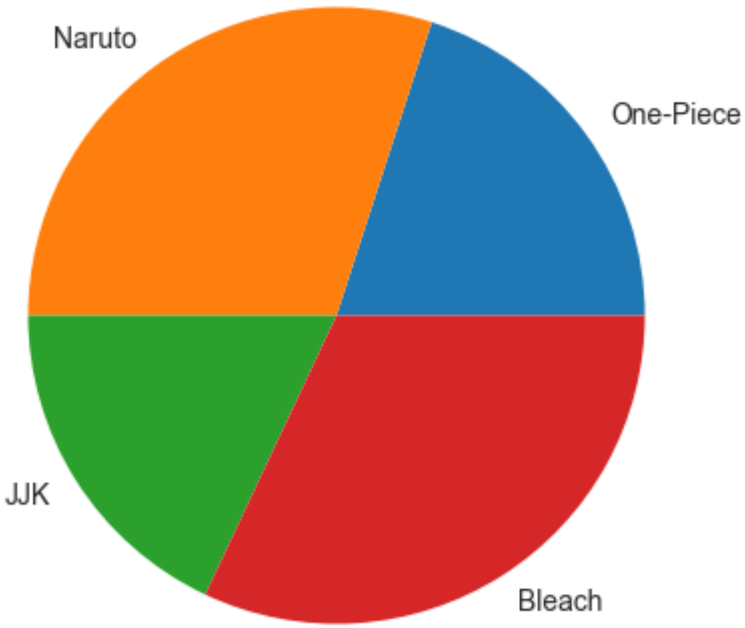
The optional **bottom** parameter of the plt.bar() function allows us to specify a starting position for a bar. Instead of running from zero to a value, it will go from the bottom to value. The first call to plt.bar() plots the blue bars. The second call to plt.bar() plots the red bars, with the bottom of the purple bars being at the top of the blue bars.

Pie Chart

Pie charts are circular representations, divided into sectors. The sectors are also called **wedges**. The arc length of each sector is proportional to the quantity we are describing. It is an effective way to represent information when we are interested mainly in comparing the wedge against the whole pie, instead of wedges against each other. Matplotlib provides the **pie()** function to plot pie charts from an array X. Wedges are created proportionally, so that each value x of array X generates a wedge proportional to x/sum(X).

In [229…

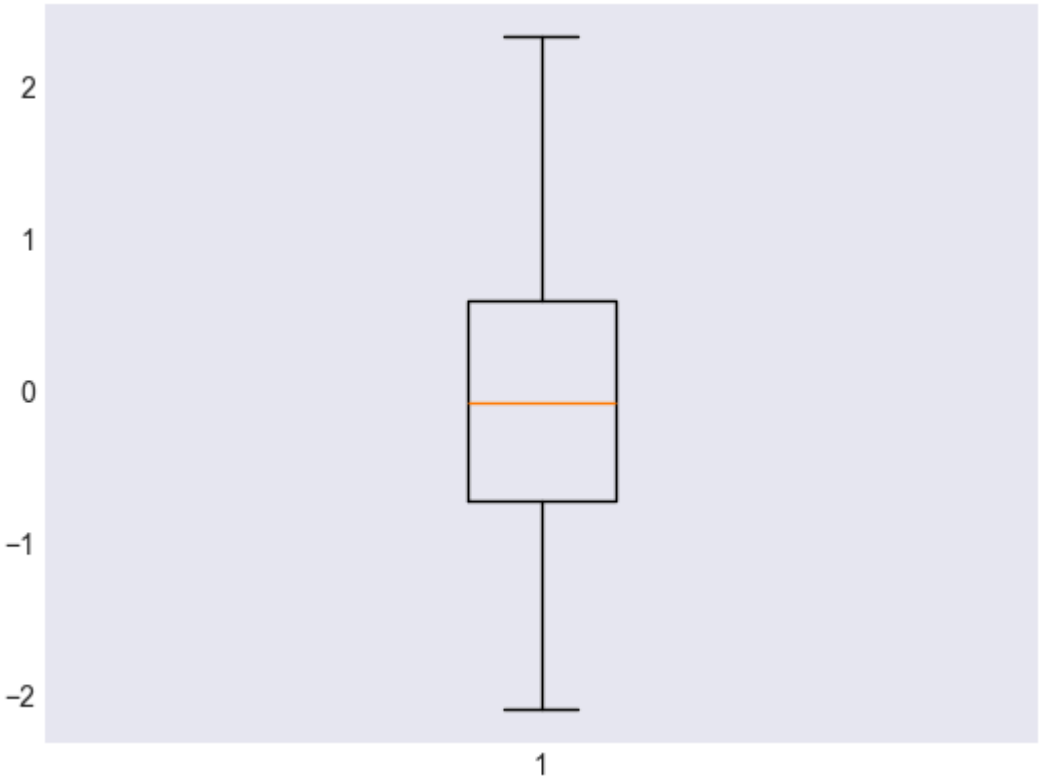
```
plt.figure(figsize=(5,5))
x=[20,30,18,32]
labels=['One-Piece', 'Naruto', 'JJK','Bleach']
plt.pie(x, labels=labels)
plt.show()
```



Boxplot

Boxplot allows us to compare distributions of values by showing the median, quartiles, maximum and minimum of a set of values. We can plot a boxplot with the **boxplot()** function as follows:-

In [230…
```
d=np.random.randn(30)
plt.boxplot(d)
plt.show()
```



The **boxplot()** function takes a set of values and computes the mean, median and other statistical quantities. The following points describe the preceeding boxplot:

- The red bar is the median of the distribution.
- The blue box includes 50 percent of the data from the lower quartile to the upper

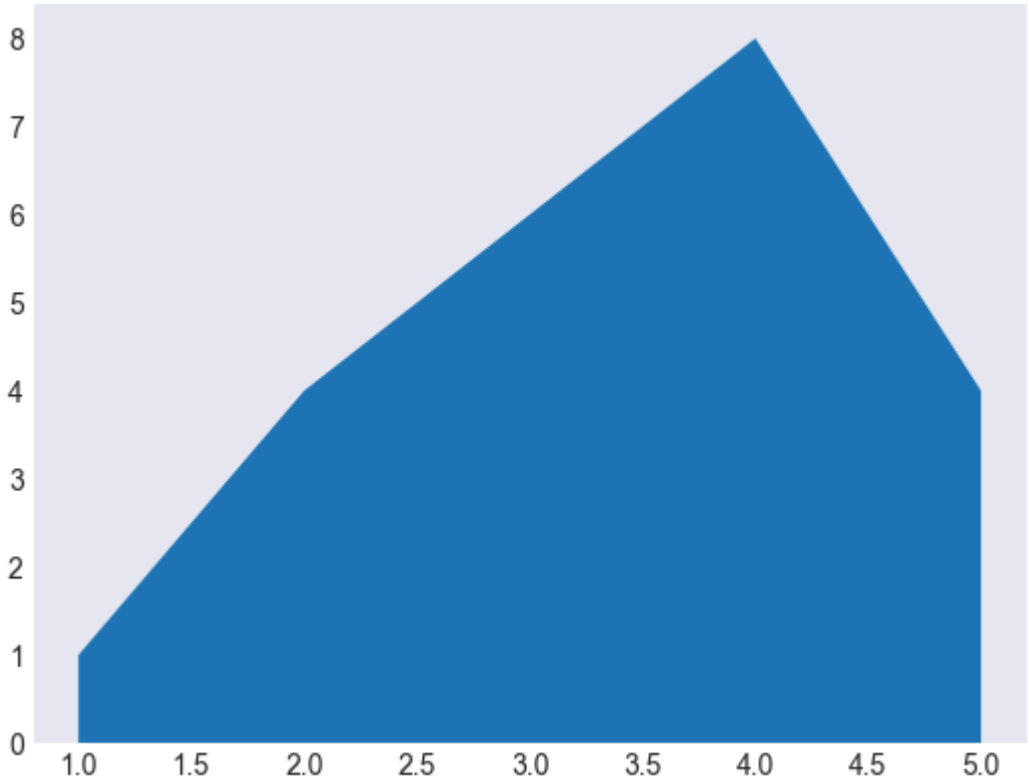
quartile. Thus, the box is centered on the median of the data.

- The lower whisker extends to the lowest value within 1.5 IQR from the lower quartile.
- The upper whisker extends to the highest value within 1.5 IQR from the upper quartile.
- Values further from the whiskers are shown with a cross marker.

Area Chart

An **Area Chart** is very similar to a **Line Chart**. The area between the x-axis and the line is filled in with color or shading. It represents the evolution of a numerical variable following another numerical variable. We can create an Area Chart as follows:-

In [231…
```
x=range(1,6)
y=[1,4,6,8,4]
#plt.fill_between(x,y)
plt.stackplot(x,y)
plt.show()
```



Contour Plot

Contour plots are useful to display three-dimensional data in two dimensions using contours or color-coded regions. **Contour** lines are also known as **level lines** or **isolines**. **Contour lines** for a function of two variables are curves where the function has constant values. They have specific names beginning with iso- according to the nature of the variables being mapped.

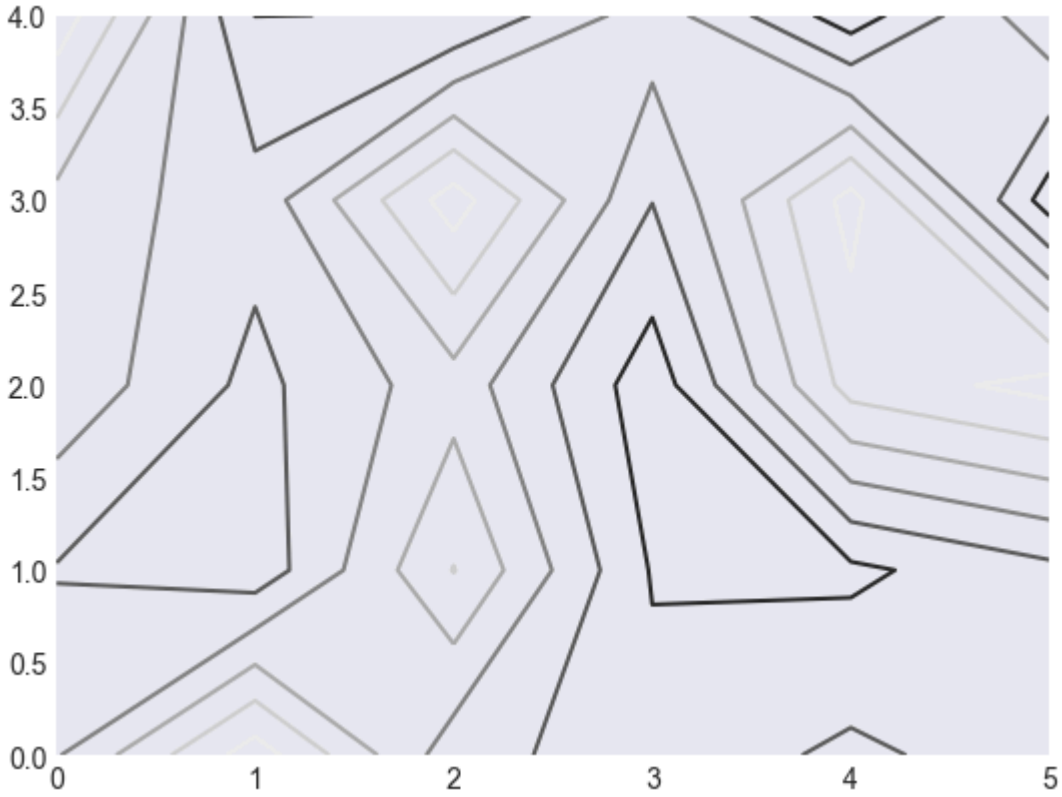
There are lot of applications of **Contour lines** in several fields such as meteorology(for temperature, pressure, rain, wind speed), geography, magnetism, engineering, social sciences and so on.

The density of the lines indicates the **slope** of the function. The **gradient** of the function is always perpendicular to the contour lines. When the lines are close together, the length of the gradient is large and the variation is steep.

A **Contour plot** can be created with the **plt.contour()** function as follows:-

In [232…

```
m2=np.random.rand(5,6)
ax=plt.contour(m2)
plt.show()
```



The **contour()** function draws contour lines. It takes a 2D array as input.Here, it is a matrix of 5 x 6 random elements.
The number of level lines to draw is chosen automatically, but we can also specify it as an additional parameter, N.

```
plt.contour(matrix, N)
```

Styles with Matplotlib Plots

The Matplotlib version 1.4 which was released in August 2014 added a very convenient style module. It includes a number of new default stylesheets, as well as the ability to create and package own styles.
We can view the list of all available styles by the following command.

```
print(plt.style.availabe)
```

In [233…
```
print(plt.style.available)
```

```
['Solarize_Light2', '_classic_test_patch', '_mpl-gallery', '_mpl-gallery-nogrid',
'bmh', 'classic', 'dark_background', 'fast', 'fivethirtyeight', 'ggplot', 'graysc
ale', 'petroff10', 'seaborn-v0_8', 'seaborn-v0_8-bright', 'seaborn-v0_8-colorblin
d', 'seaborn-v0_8-dark', 'seaborn-v0_8-dark-palette', 'seaborn-v0_8-darkgrid', 's
eaborn-v0_8-deep', 'seaborn-v0_8-muted', 'seaborn-v0_8-notebook', 'seaborn-v0_8-p
aper', 'seaborn-v0_8-pastel', 'seaborn-v0_8-poster', 'seaborn-v0_8-talk', 'seabor
n-v0_8-ticks', 'seaborn-v0_8-white', 'seaborn-v0_8-whitegrid', 'tableau-colorblin
d10']
```

We can set the **Styles** for Matplotlib plots as follows:-

```
plt.style.use('seaborn-bright')
```

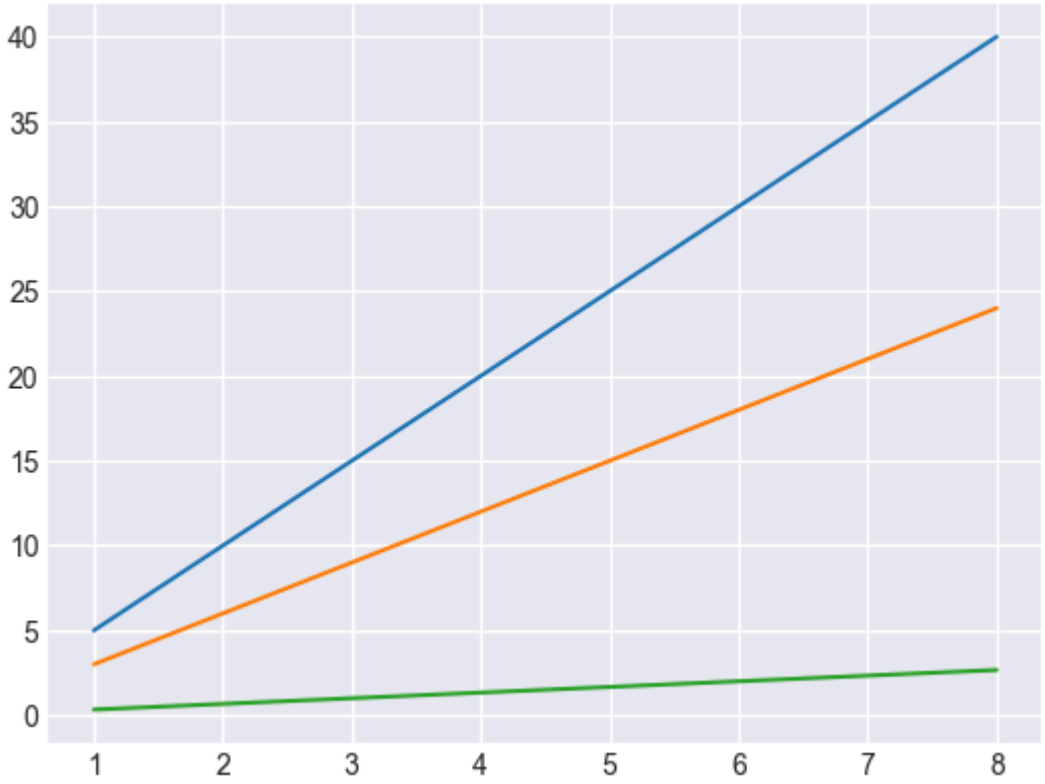
In [234…
```
plt.style.use('seaborn-v0_8-dark')
```

I have set the seaborn-bright style for plots. So, the plot uses the seaborn-bright Matplotlib style for plots.

Adding a grid

In some cases, the background of a plot was completely blank. We can get more information, if there is a reference system in the plot. The reference system would improve the comprehension of the plot. An example of the reference system is adding a **grid**. We can add a grid to the plot by calling the **grid()** function. It takes one parameter, a Boolean value, to enable(if True) or disable(if False) the grid.

In [235…
```
x=np.arange(1,9)
plt.plot(x, x*5,x,x*3,x,x/3)
plt.grid(True)
plt.show()
```

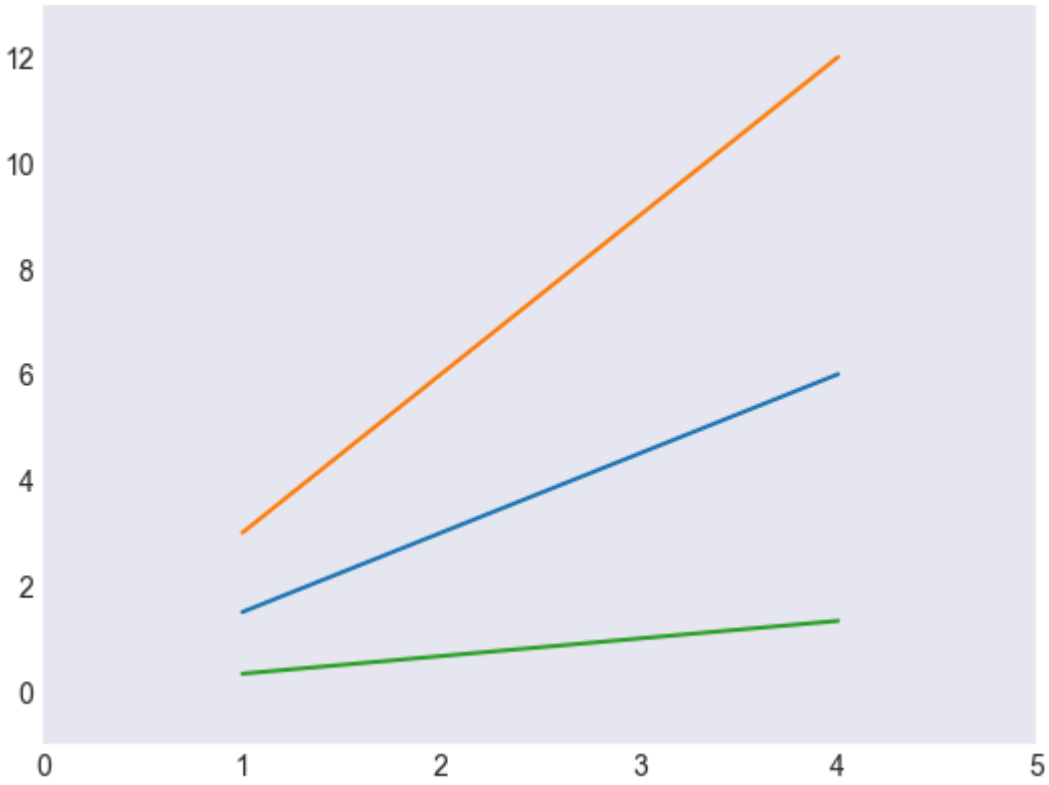


Handling axes

Matplotlib automatically sets the limits of the plot to precisely contain the plotted datasets. Sometimes, we want to set the axes limits ourself. We can set the axes limits with the **axis()** function as follows:-

In [236…
```
x=np.arange(1,5)
plt.plot(x, x*1.5,x,x*3,x,x/3)
```

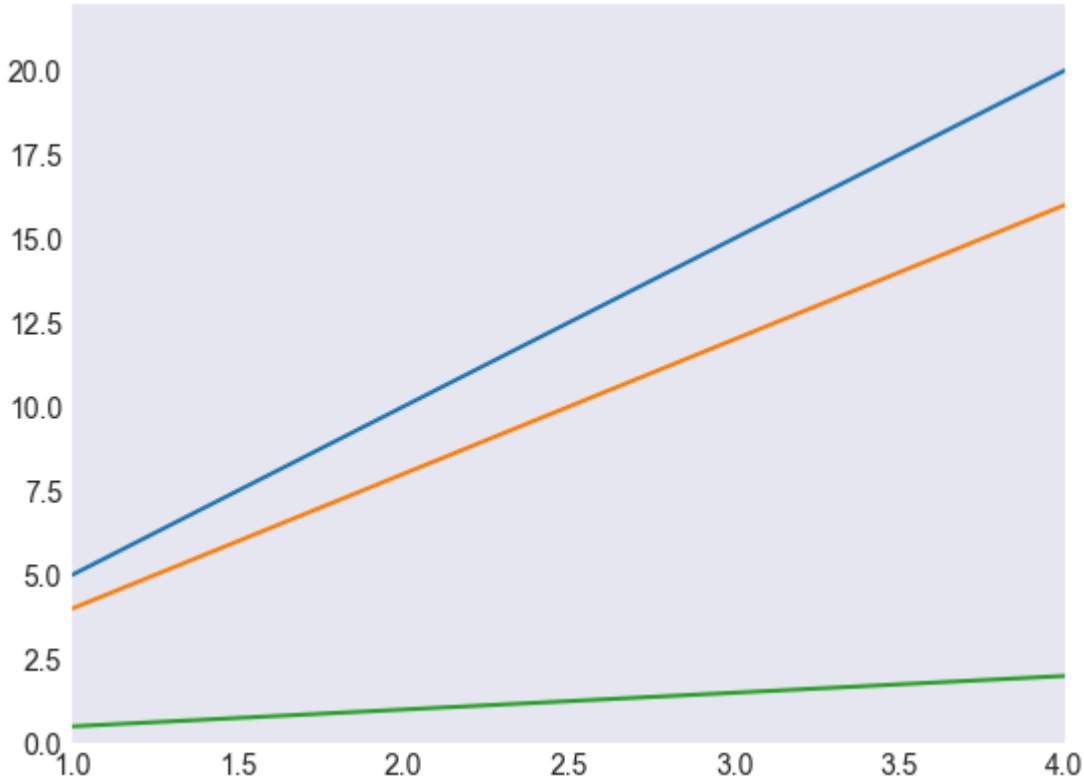
```
plt.axis()
plt.axis([0,5,-1,13])
plt.show()
```



We can see that we now have more space around the lines.
If we execute **axis()** without parameters, it returns the actual axis limits.
We can set parameters to **axis()** by a list of four values.
The list of four values are the keyword arguments [xmin, xmax, ymin, ymax] allows the minimum and maximum limits for X and Y axis respectively.
We can control the limits for each axis separately using the `xlim( )` and `ylim()` functions. This can be done as follows:

In [237…
```
x=np.arange(1,5)
plt.plot(x,x*5,x,x*4,x,x/2)
plt.xlim([1,4])
plt.ylim([0,22])
```

Out[237…
```
(0.0, 22.0)
```



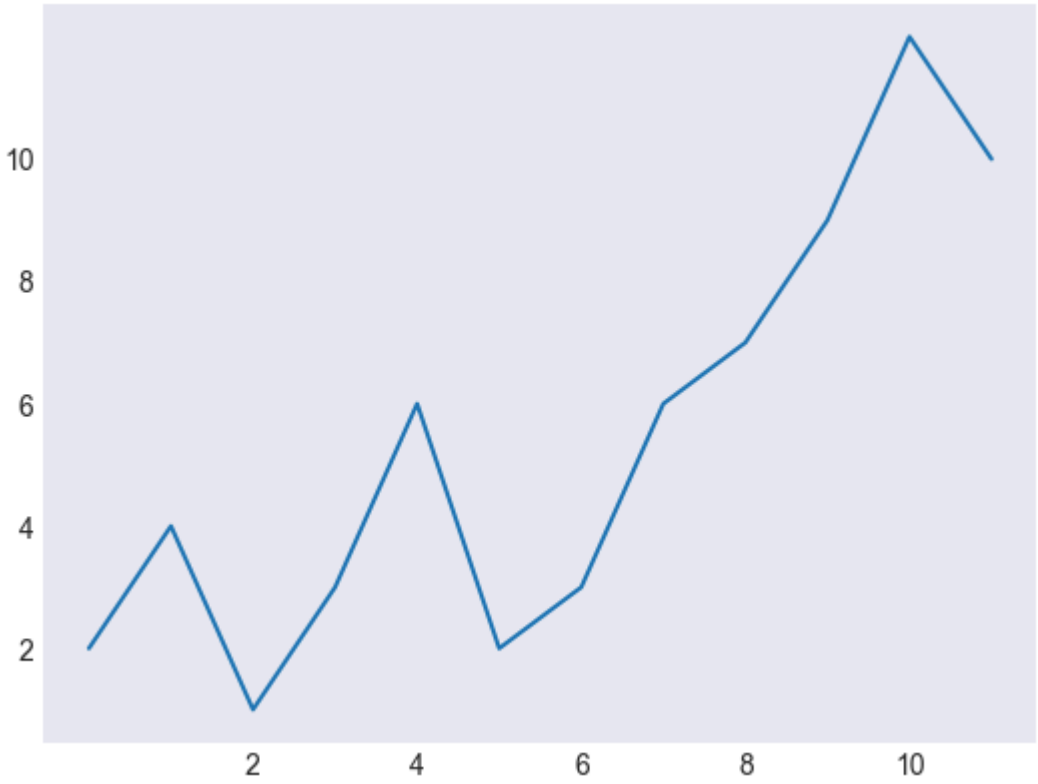
Handling X and Y ticks

Vertical and horizontal ticks are those little segments on the axes, coupled with axes labels, used to give a reference system on the graph.So, they form the origin and the grid lines.
Matplotlib provides two basic functions to manage them - **xticks()** and **yticks()**.
Executing with no arguments, the tick function returns the current ticks' locations and the labels corresponding to each of them.
We can pass arguments(in the form of lists) to the ticks functions. The arguments are:-
1. Locations of the ticks.
2. Labels to draw at these locations.

We can demonstrate the usage of the ticks functions in the code snippet below:-

In [238…

```
c=[2,4,1,3,6,2,3,6,7,9,12,10]
plt.plot(c)
plt.xticks([2,4,6,8,10])
plt.yticks([2,4,6,8,10])
plt.show()
```

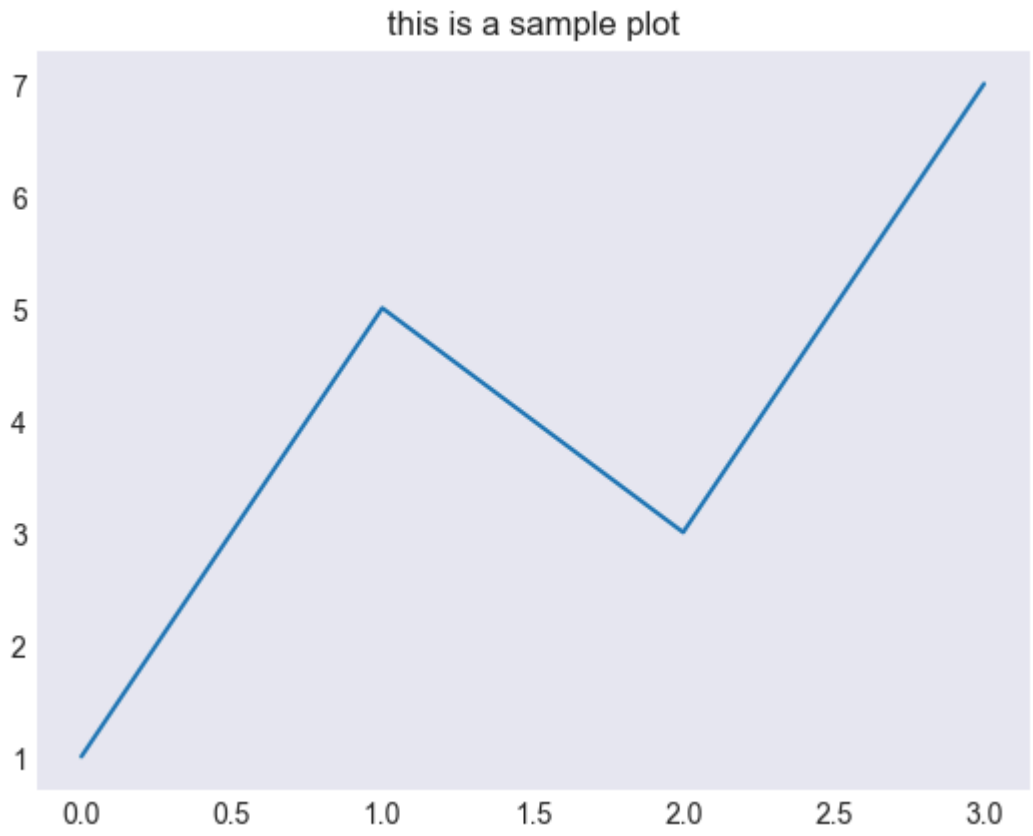


Adding a title

The title of a plot describes about the plot. Matplotlib provides a simple function title() to add a title to an image.

In [239…

```
plt.plot([1,5,3,7])
plt.title('this is a sample plot')
plt.show()
```

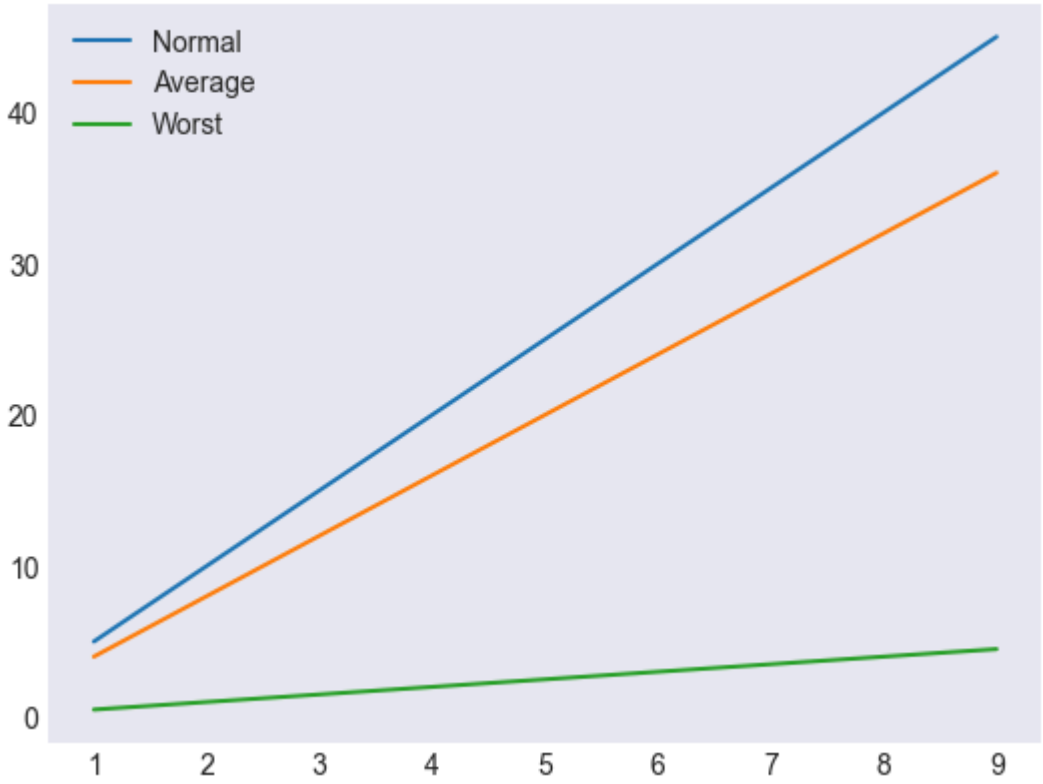


Adding a legend

Legends are used to describe what each line or curve means in the plot.
Legends for curves in a figure can be added in two ways. One method is to use the **legend** method of the axis object and pass a list/tuple of legend texts as follows:-

In [240…
```
x=np.arange(1,10)
f,ax=plt.subplots()
ax.plot(x,x*5)
ax.plot(x,x*4)
ax.plot(x,x/2)
ax.legend(['Normal', 'Average', 'Worst'])
```

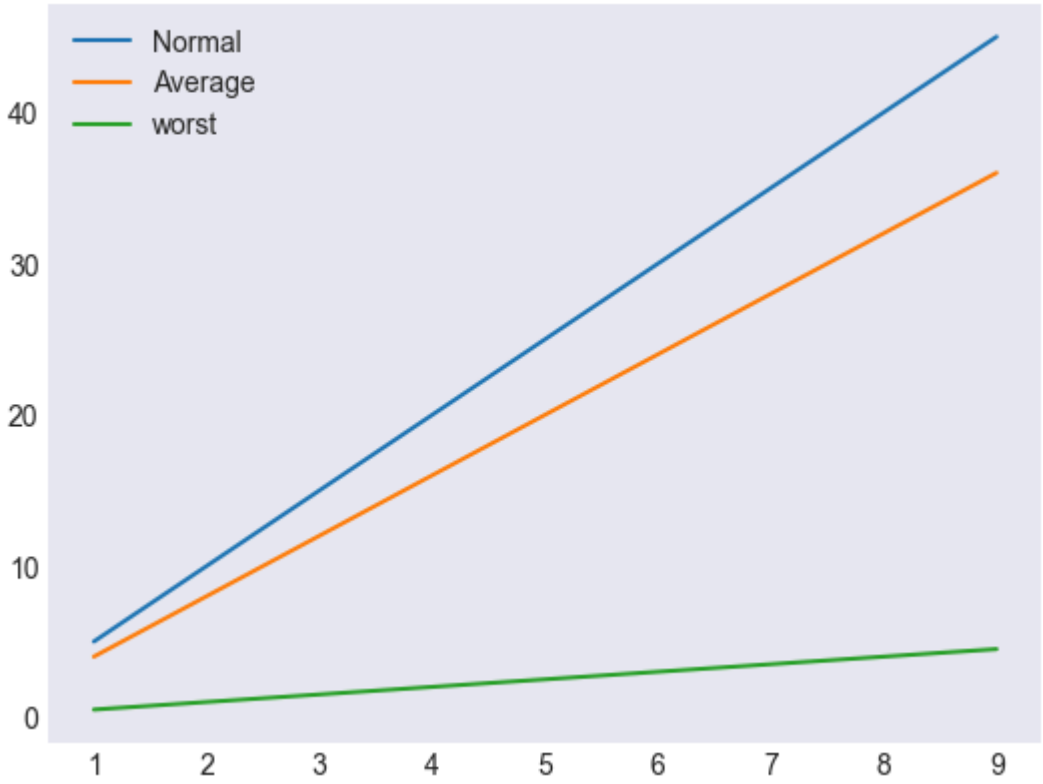
Out[240…
```
<matplotlib.legend.Legend at 0x183c99ddc10>
```



The above method follows the MATLAB API. It is prone to errors and unflexible if curves are added to or removed from the plot. It resulted in a wrongly labelled curve.
A better method is to use the **label** keyword argument when plots are added to the figure. Then we use the **legend** method without arguments to add the legend to the figure.
The advantage of this method is that if curves are added or removed from the figure, the legend is automatically updated accordingly. It can be achieved by executing the code below:-
The above method follows the MATLAB API. It is prone to errors and unflexible if curves are added to or removed from the plot. It resulted in a wrongly labelled curve.

In [241…
```
x=np.arange(1,10)
f,ax=plt.subplots()
ax.plot(x,x*5, label='Normal')
ax.plot(x,x*4, label='Average')
ax.plot(x,x/2, label='worst')
ax.legend()
```

Out[241…
```
<matplotlib.legend.Legend at 0x183c610b2d0>
```



The **legend** function takes an optional keyword argument **loc**. It specifies the location of the legend to be drawn. The **loc** takes numerical codes for the various places the legend can be drawn. The most common **loc** values are as follows:-

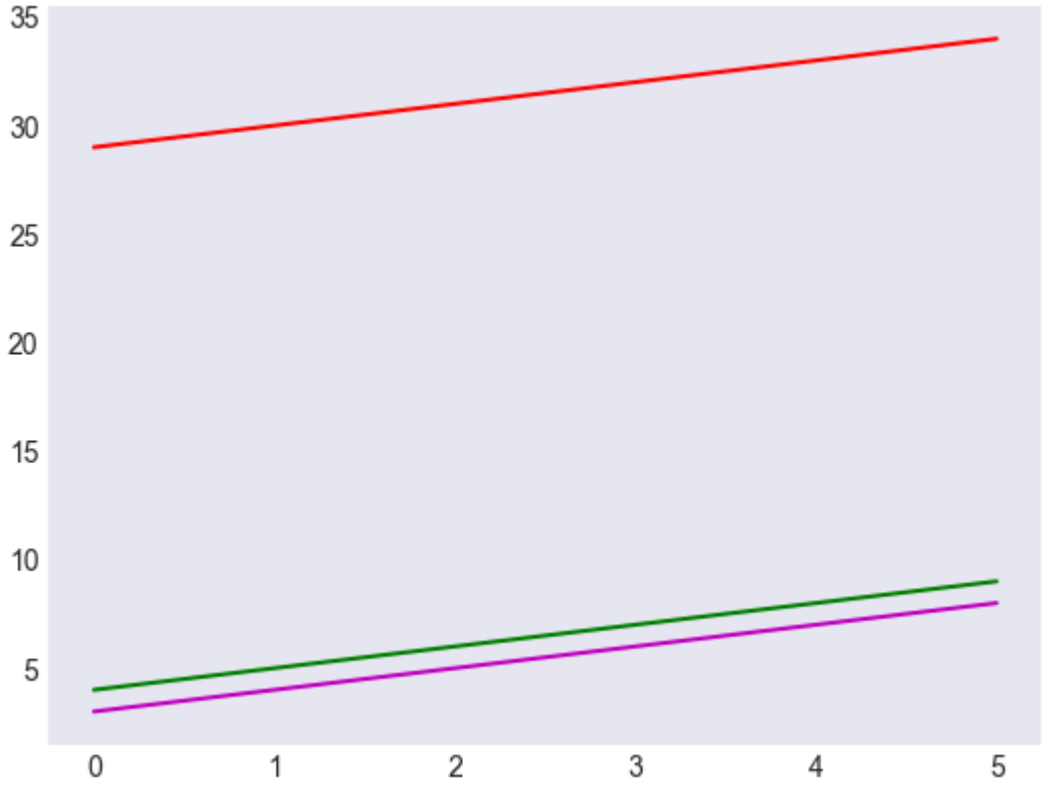
ax.legend(loc=0) # let Matplotlib decide the optimal location
ax.legend(loc=1) # upper right corner
ax.legend(loc=2) # upper left corner
ax.legend(loc=3) # lower left corner
ax.legend(loc=4) # lower right corner
ax.legend(loc=5) # right
ax.legend(loc=6) # center left
ax.legend(loc=7) # center right
ax.legend(loc=8) # lower center
ax.legend(loc=9) # upper center

Control colours

We can draw different lines or curves in a plot with different colours. In the code below, we specify colour as the last argument to draw red, blue and green lines.

In [242…

```
x=np.arange(3,9)
plt.plot(x, 'm')
plt.plot(x+1, 'g')
plt.plot(x+26, 'r')
plt.show()
```



The colour names and colour abbreviations are given in the following table:-

Colour abbreviation Colour name

b -> blue

c -> cyan

g -> a green

k -> black

m -> magenta

r -> a red

w -> white

y -> yellow

There are several ways to specify colours, other than by colour abbreviations:

- The full colour name, such as yellow
- Hexadecimal string such as ##FF00FF

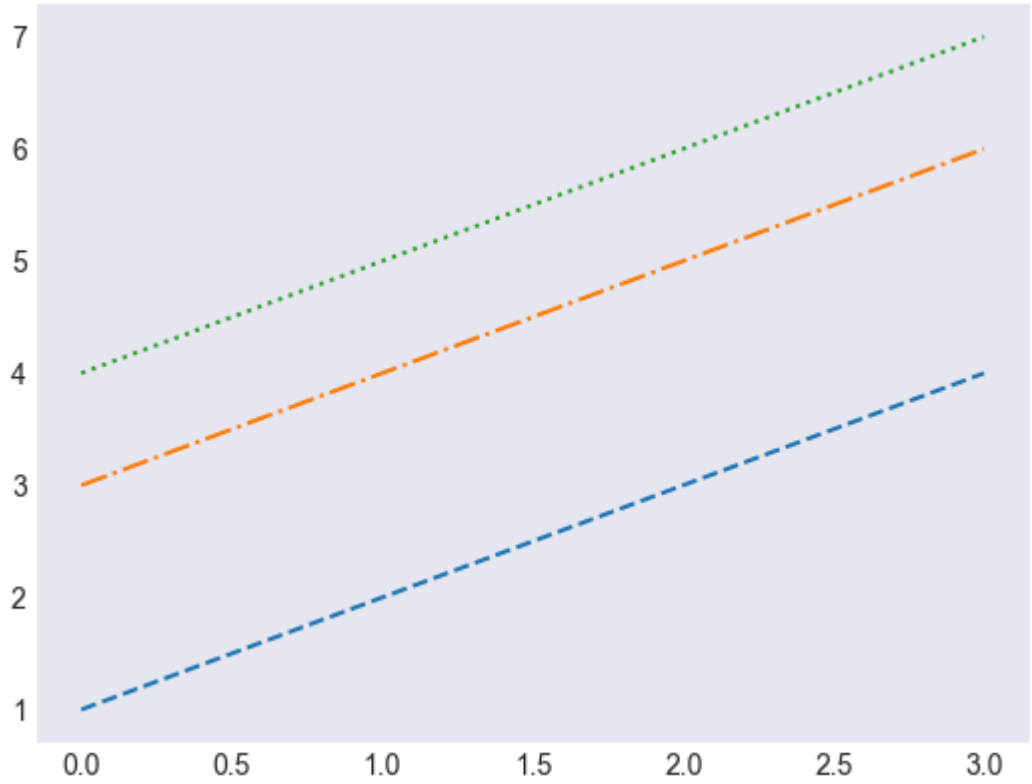
- RGB tuples, for example (1, 0, 1)
- Grayscale intensity, in string format such as '0.7'.

Control line styles

Matplotlib provides us different line style options to draw curves or plots. In the code below, I use different line styles to draw different plots.

In [243…

```
x=np.arange(1,5)
plt.plot(x,'--', x+2,'-.', x+3,':')
plt.show()
```



The above code snippet generates a blue dashed line, a green dash-dotted line and a red dotted line.
All the available line styles are available in the following table:

Style abbreviation Style

- solid line

-- dashed line

-. dash-dot line
: dotted line

Now, we can see the default format string for a single line plot is 'b-'.