#Plotting Graphs

What will you learn?

1. Scatter Graph
2. Line Graph
3. **Adding Styles** : Colors, LineWidth, Titles, Labels, Legends, Subplots
4. **Different Variations** : Pie Graphs, Bubble Charts, Histograms, Bar Graphs

To learn about and plot high quality, detailed graphs, we will be using Matplotlib, a Python library for data visualization. There is usually no need to install it separately because Anaconda has it installed by default. We will be drawing different types of graphs like line graph, bar graph, pie charts, etc. and adding styles, colours, legends, title, etc in these graphs.

##Pyplot

To start plotting graphs, we import the pyplot submodule from 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, plots some lines in a plotting area, decorates the plot with labels, etc.

In order to plot a 2-D graph (say), we need to have a pair of coordinates: those of the x and y axes. If we declare our coordinates as arrays, the size of the x-coordinate array and y-coordinate array must be same. Matplotlib is convenient, in that in order to plot the coordinates with, say a scatter plot, we simply have to write pyplot.scatter() and we can have the points plotted on a scatter plot. Note that this will plot the graph but won't display it; we need to write pyplot.show() in order to display the graph.

In [ ]:

```
#Importing pyplot library
import matplotlib.pyplot as plt
#Importing Numpy library
import numpy as np
```

Matplotlib works with all sorts of data. If it were limited to working with lists, it would be fairly useless for numeric processing. Generally, you will use numpy arrays. In fact, all sequences are converted to numpy arrays internally.

If we wish to have the points linked and not scattered, we can use pyplot.plot() for the same. The advantage of these functions is that there are several properties and parameters we can tinker with in order to enhance our graph further. We can give it colours and even the pattern of the lines in between the points, for instance dashed or bold.

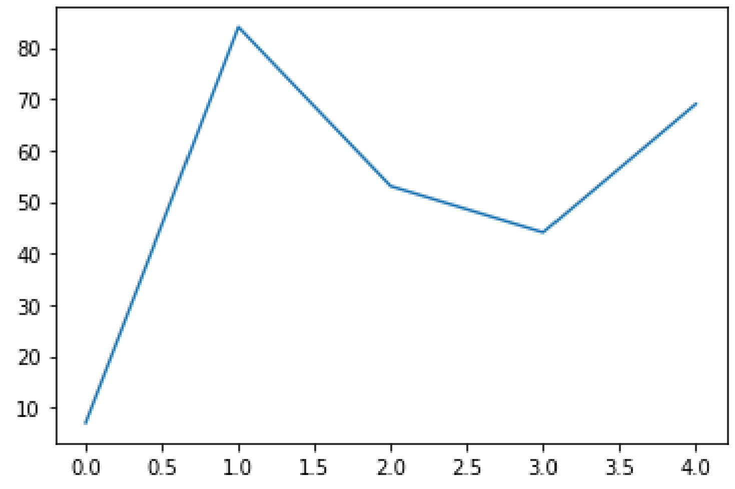
You can make several plots like bar graphs, histograms etc. and also adjust scales – like setting a logarithmic scale etc.

##Plot

The plot() function accepts an arbitrary number of arguments.

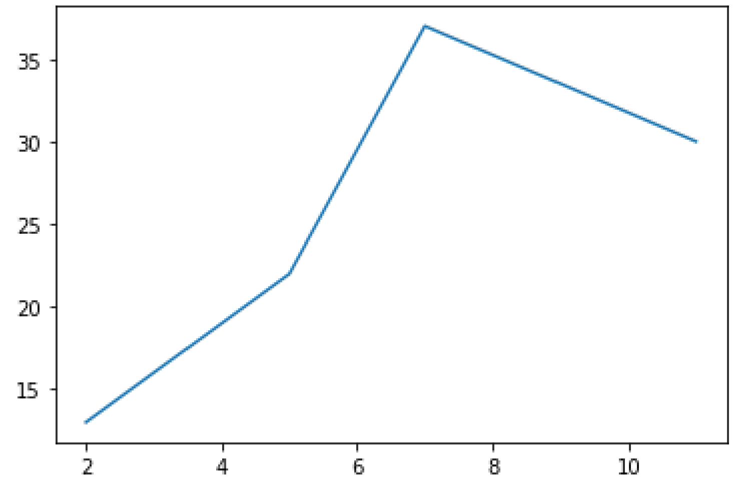
In [ ]:

```
# Plotting values on the graph
A = [7, 84, 53, 44, 69]
plt.plot(A)
plt.show()
```



In [ ]:

```
# Plotting graph with 2 inter-related variables (through some function)
X = [2, 5, 7, 11]
Y = [13, 22, 37, 30]
plt.plot(X, Y)
plt.show()
```

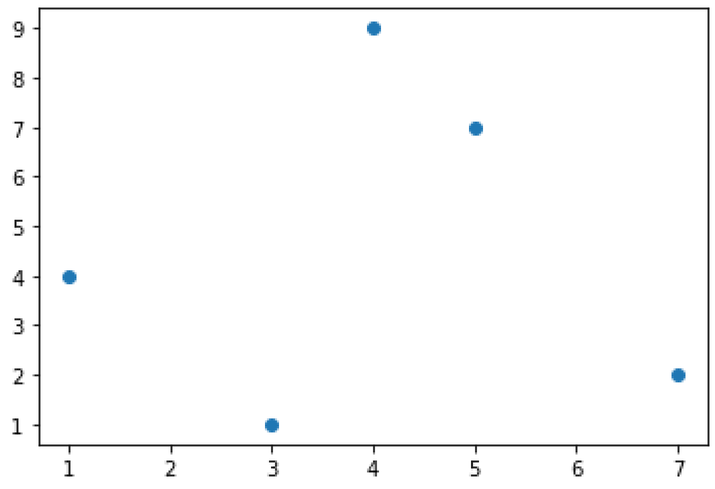


##Scatter

If we want to plot random data points without connecting them with lines, we can use scatter.

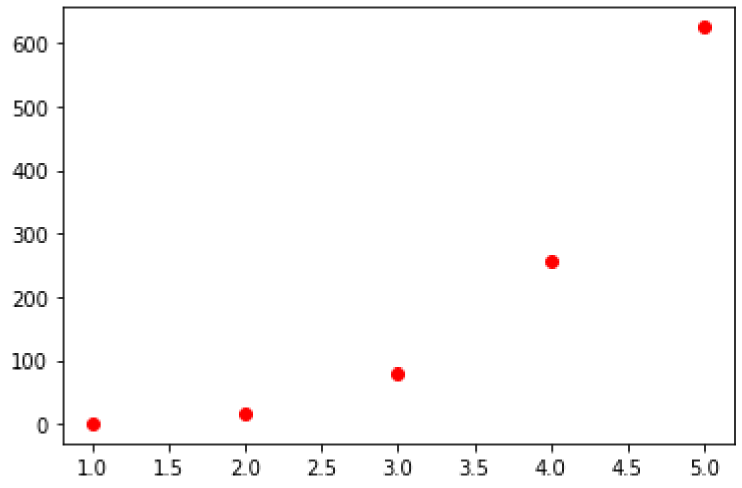
In [ ]:

```
X = np.array([1, 3, 4, 5, 7])
Y = np.array([4, 1, 9, 7, 2])
plt.scatter(X, Y)
plt.show()
```



In [ ]:

```
# Plotting a function with respect to given points. The 'ro'
# parameter specifies two things: the 'r' specifies the color of
# the points which is red, and the 'o' specifies the shape.
x = np.array([1, 2, 3, 4, 5])
y = x**4
plt.plot(x, y, 'ro')
plt.show()
```

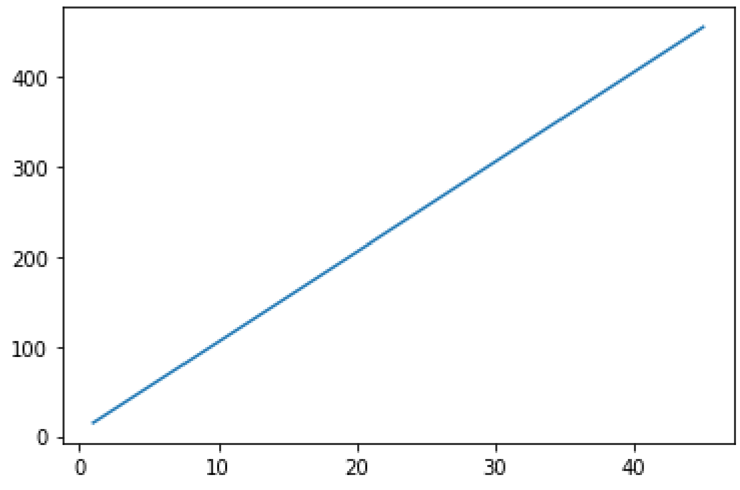


##Line Graph

We can plot linearly related mathematical functions (y=mx+b) using pyplot.

In [ ]:

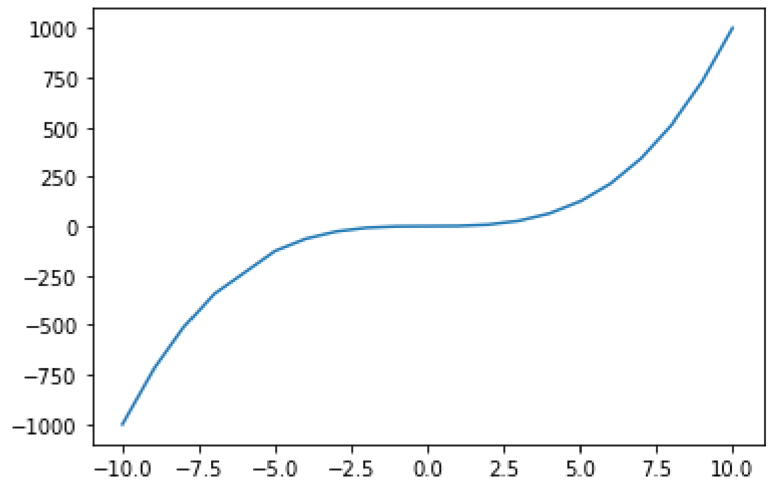
```
m = 10
b = 5
X = np.array([1,8,20,45])
Y = m*X + b
plt.plot(X,Y)
plt.show()
```



In [ ]:

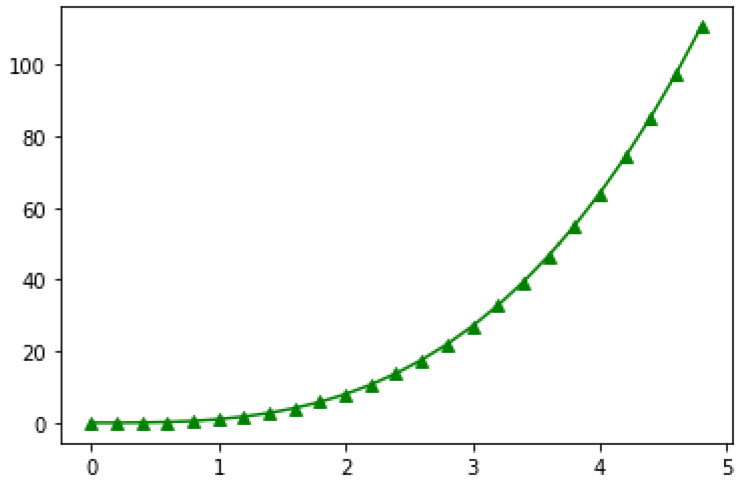
```
# Plotting function y = x^3

# X = np.array([-10, 0, 10])
X = np.array([-10,-9,-8,-7,-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7,8,9,10])
print('X --> ', X)
# X = np.linspace(-15,15,100)
# This is another way of creating X array with numbers 100 linearly spaced numbers from -15
# You can uncomment it to see the plot.
Y = X**3
plt.plot(X, Y)
plt.show()
```

```
X -->  [-10  -9  -8  -7  -5  -4  -3  -2  -1   0   1   2   3   4   5   6   7
8
   9  10]
```



In [ ]:

```
# Evenly sampled time at 200ms intervals
t = np.arange(0., 5., 0.2)

# Red dashes, blue squares and green triangles
# We can plot multiple functions within one plot as well.
plt.plot(t, t**3, color = 'green', marker='^')
plt.show()
```



##Adding Styles in Graphs

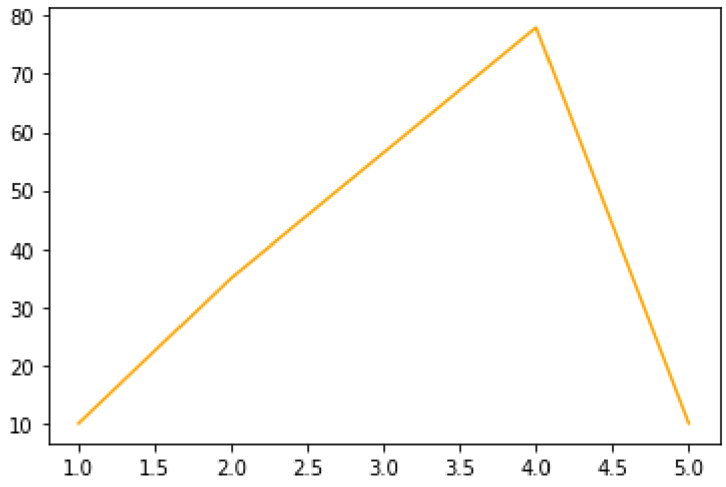
Now that we know how to create basic plots, how do we customize them?

###Colors and Markers

You can change the colour of your plots by adding a ‘color’ parameter in the plot function definition (which can accept an arbitrary number of arguments, by the way). Adding a ‘marker’ parameter will shape your plot points – say circular, triangular etc. You need to assign the character ‘o’ for a circular shape and ‘^’ for triangular.

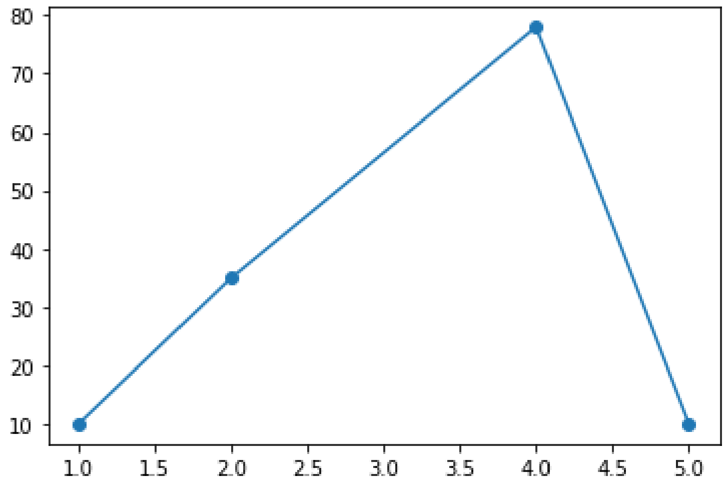
In [ ]:

```
N = 50
X = np.array([1,2,4,5])
Y = np.array([10,35,78,10])

# Color property can be used to change colours

plt.plot(X,Y,color = 'orange')
# We can use any color here in values

plt.show()
```



In [ ]:

```
N = 50
X = np.array([1,2,4,5])
Y = np.array([10,35,78,10])

# Marker property can be used to change markers to any valid marker style like circle,diamo

plt.plot(X,Y,marker = 'o')
# We can use any color here in values

plt.show()
```



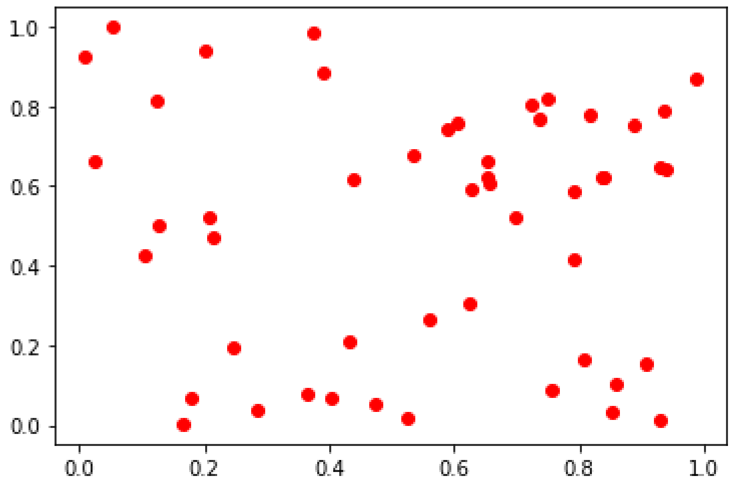
We can use format strings to change colour, markers and line styles.

fmt = '[color][marker][line]'

All of these are optional. By default, we have blue solid line.

In [ ]:

```
N = 50
X = np.random.rand(N)
Y = np.random.rand(N)

plt.plot(X,Y,'ro')
# Here, r is for red colour and o is for circle

# ro - is used for plotting red circles
# plt.plot(X,Y,'bo') - for blue circle
# plt.plot(X,Y,'r+') - red colour and + marker
# plt.plot(X,Y,'b--') - for blue dotted lines
# plt.plot(X,Y,'o-')
# plt.plot(X,Y,'g--d') - green dotted line with diamond shape marker
# plt.plot(X,Y,'k^:') - black triangle_up markers connected by a dotted line

plt.show()
```

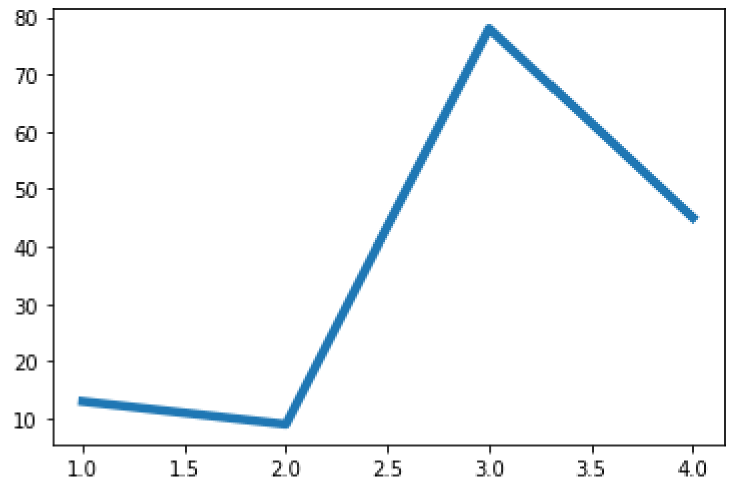


###Line Width

There are many, many other arguments that can be passed as well. For instance, the linewidth parameter adjusts the width of our plot line – how thick or thin you want it to be.

In [ ]:

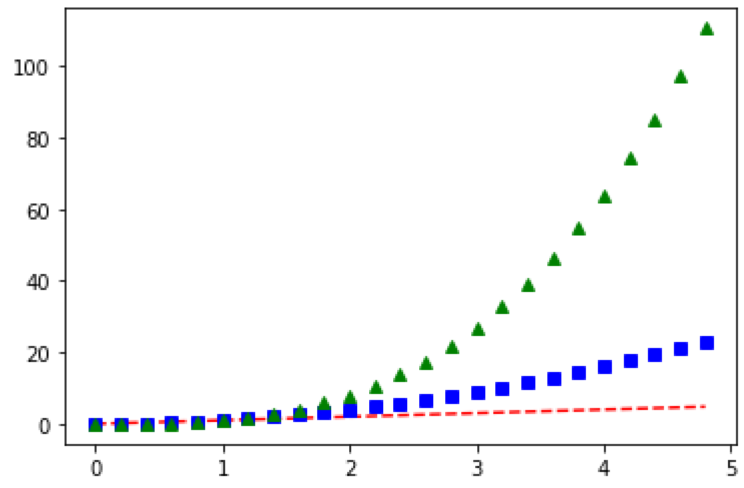
```
X = np.array([1,2,3,4])
Y = np.array([13,9,78,45])
plt.plot(X,Y,linewidth = 4.5)
# Linewidth can take any float values
plt.show()
```



###Multiple Equations

In [ ]:

```
t = np.arange(0, 5, 0.2)
# t has evenly sampled values from 0 to 5 with 0.2 step value
plt.plot(t, t, 'r--', t, t**2, 'bs', t, t**3, 'g^')
# red dashes, blue squares and green triangles
# we can plot multiple equations on the same figure
plt.show()
```



###Titles, Labels and Legends

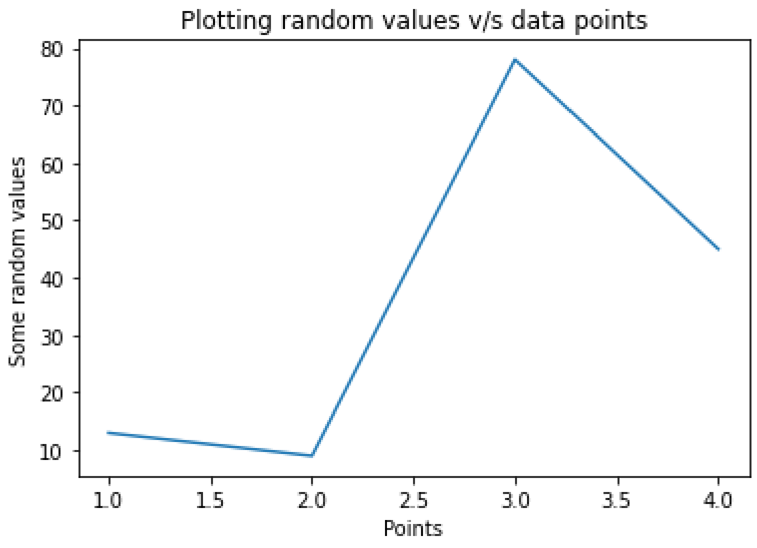
Now how do we make our graph more descriptive? That is usually done by adding labels to the axes, a legend (scale) to the graph and so on.

To give a label to the y-axis, use the ylabel function and give it the label name which you prefer, and similarly for the x-axis use the xlabel function. Note that these have to be defined before the show() function, or they won't run.

The title() function will be used to provide the title for the graph, and the legend() function will set the legend for the plot – but only after you specify the ‘label’ attribute in the plot() function definition.

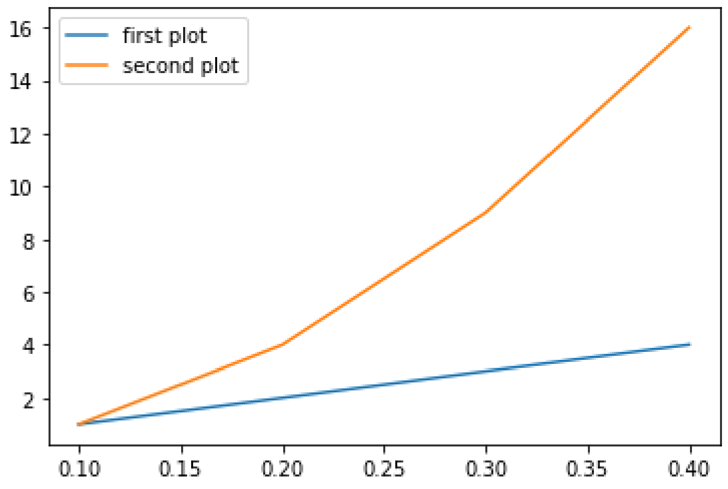
In [ ]:

```
X = np.array([1,2,3,4])
Y = np.array([13,9,78,45])
plt.plot(X,Y)
plt.ylabel("Some random values")
plt.xlabel("Points")
# Adding x and y axis labels
plt.title("Plotting random values v/s data points")
# Adding title on the graphs
plt.show()
```



In [ ]:

```
plt.plot([0.1, 0.2, 0.3, 0.4], [1, 2, 3, 4], label='first plot')
plt.plot([0.1, 0.2, 0.3, 0.4], [1, 4, 9, 16], label='second plot')
plt.legend()
plt.show()
```

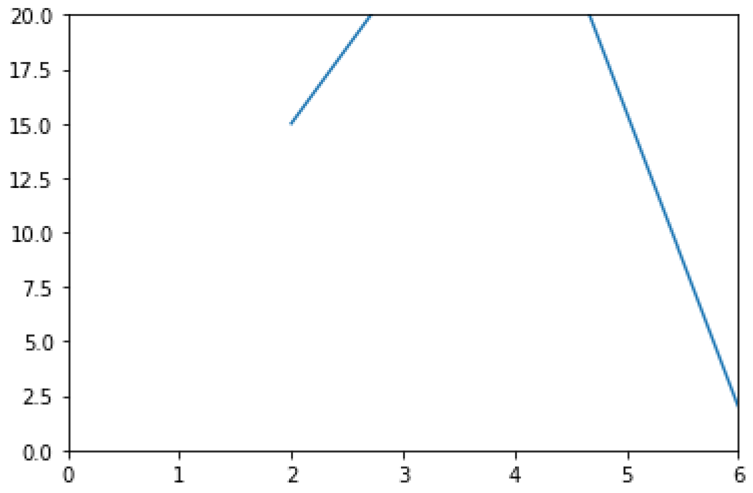


###**Axis Lengths**

You can also set the axis length for both your axes by specifying the axis() function and giving it the start and end points for both axes.

In [ ]:

```
X = np.array([2,4,6,8,10])
Y = np.array([15,29,2,44,9])
plt.plot(X,Y)
#We can set xlimits and ylimits for the plots in the following format [xmin,xmax,ymin,ymax]
plt.axis([0,6,0,20])
plt.show()
```

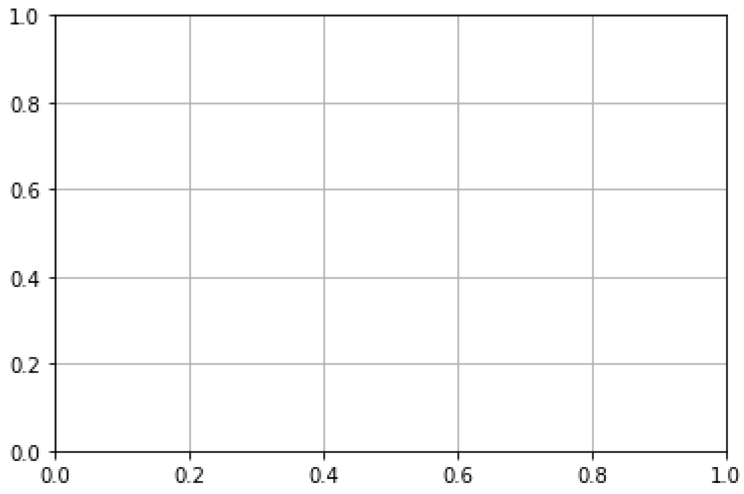


###Grid Style

We can also visualize your graph through a grid by using the grid() function. The advantage of using a grid is that the plotted points are easier to track and visualize.

In [ ]:

```
plt.grid(True)
plt.show()
```

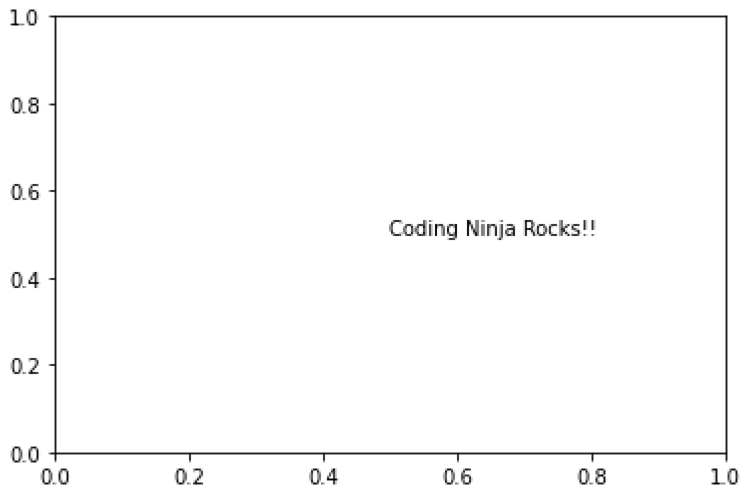


###Adding Texts

What if you want to add some text to your graph at some random place in it? The text() function comes in handy. You need to specify the coordinates where you want to add your text and the content you want to be displayed. You can also add the fontsize parameter to it to adjust the size of the text.

In [ ]:

```
plt.text(0.5, 0.5, "Coding Ninja Rocks!!")
plt.show()
```

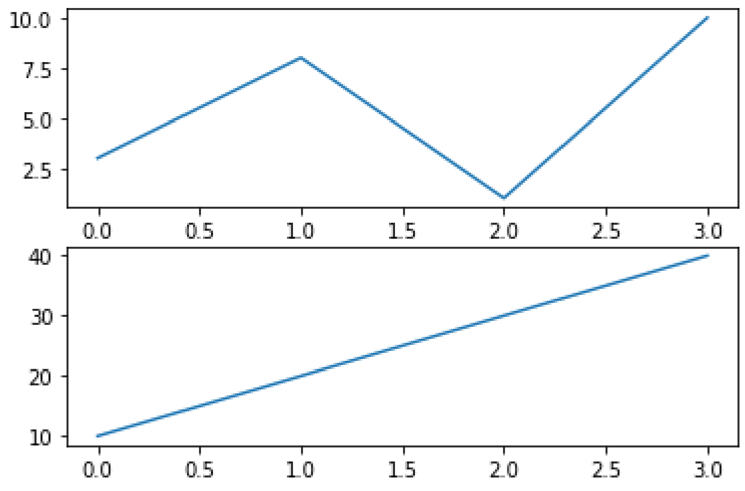


###Subplots

The subplot function allows one to plot multiple subplots within one plot. You need to specify the sizes which will be occupied by it in the bigger plot. For example, subplot(2, 3, 3) and subplot(233) both create an Axes at the top right corner of the current figure, occupying half of the figure height and a third of the figure width.

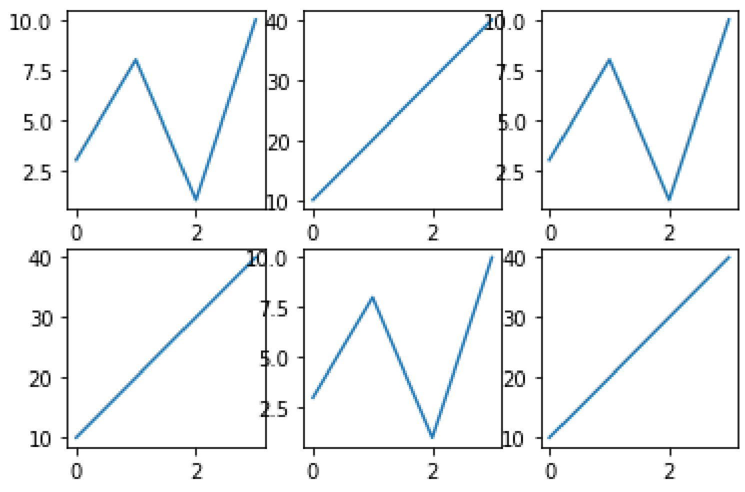
In [ ]:

```
# Plot 1:
x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 10])

plt.subplot(2, 1, 1)
plt.plot(x,y)

# Plot 2:
x = np.array([0, 1, 2, 3])
y = np.array([10, 20, 30, 40])

plt.subplot(2, 1, 2)
plt.plot(x,y)

plt.show()
```



In [ ]:

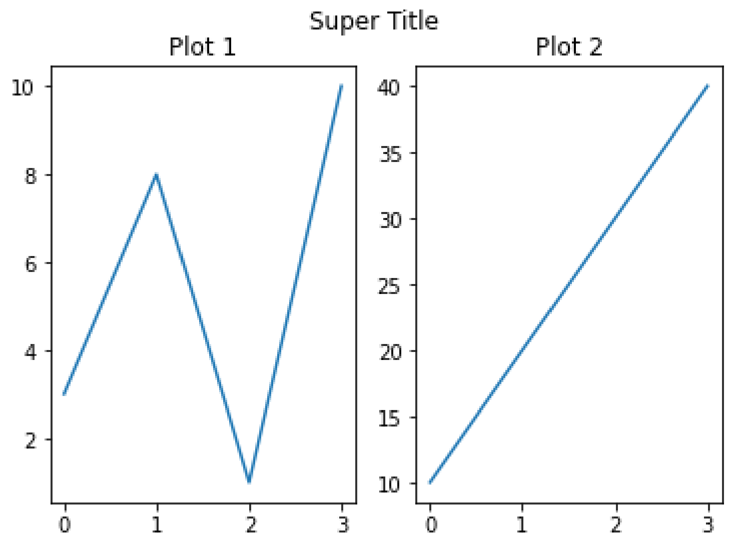
```
x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 10])

plt.subplot(2, 3, 1)
plt.plot(x,y)

x = np.array([0, 1, 2, 3])
y = np.array([10, 20, 30, 40])

plt.subplot(2, 3, 2)
plt.plot(x,y)

x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 10])

plt.subplot(2, 3, 3)
plt.plot(x,y)

x = np.array([0, 1, 2, 3])
y = np.array([10, 20, 30, 40])

plt.subplot(2, 3, 4)
plt.plot(x,y)

x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 10])

plt.subplot(2, 3, 5)
plt.plot(x,y)

x = np.array([0, 1, 2, 3])
y = np.array([10, 20, 30, 40])

plt.subplot(2, 3, 6)
plt.plot(x,y)

plt.show()
```



You can add a title to the entire figure with the suptitle() function:

In [ ]:

```
# Plot 1:
x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 10])

plt.subplot(1, 2, 1)
plt.plot(x,y)
plt.title("Plot 1")

# Plot 2:
x = np.array([0, 1, 2, 3])
y = np.array([10, 20, 30, 40])

plt.subplot(1, 2, 2)
plt.plot(x,y)
plt.title("Plot 2")

plt.suptitle("Super Title")
plt.show()
```



##Different Variations for Graph Plotting

###Pie Charts

We next come to a fancier, more interesting way of representing data – pie graphs. Pie charts can be plotted very easily using the pie() function. We need to specify the sizes of each portion in the chart in a list.

Once you plot the pie chart, you might notice that the chart isn't round – which can be remedied using the axis() function and passing it the parameter 'equal'.

We're done with our basic pie chart – how else can we customise it? We can pass it the 'colors' parameter and specify the colours of each section, and also give each section labels using the 'labels' parameter.

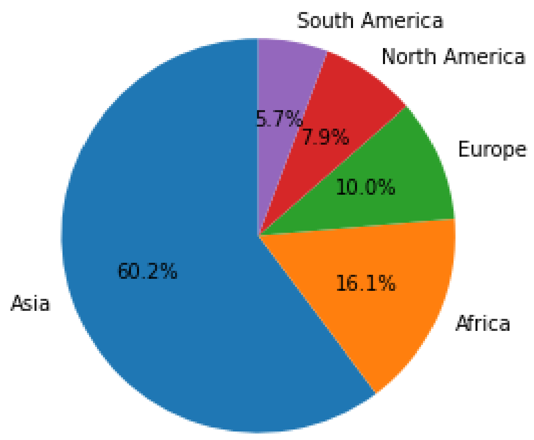
How do we display information on the chart itself? We can display the percentage of each section of the pie by giving the parameter 'autopct'. The autopct parameter needs a function (in the form of a string) that demonstrates how to display the percentage on the chart – for instance, '%.2f%%' will display the percentages with two digits to the right of decimal. And it is not necessary that just the percentages can be displayed on the pie chart; we can pass any function we wish into the autopct parameter and it'll decide the values on its own.

You must have seen in several places that sometimes, for more interesting insight, a certain part of the pie chart is usually shown to be slightly out of the pie – a way of highlighting it. We can do that by passing the 'explode' parameter and giving it the values by which each section should be pushed out of the pie chart. A value of 1 indicated that the section is to be pushed entirely out of the pie.

There are many, many other parameters we can pass to the pie() function to make our graph even better and informative. For instance, the counterclock parameter can be used to specify fractions direction, clockwise or counter-clockwise. The wedgeprops parameter takes in a dictionary of arguments passed to the wedge objects making the pie. For example, you can pass in wedgeprops = {'linewidth': 3} to set the width of the wedge border lines equal to 3.

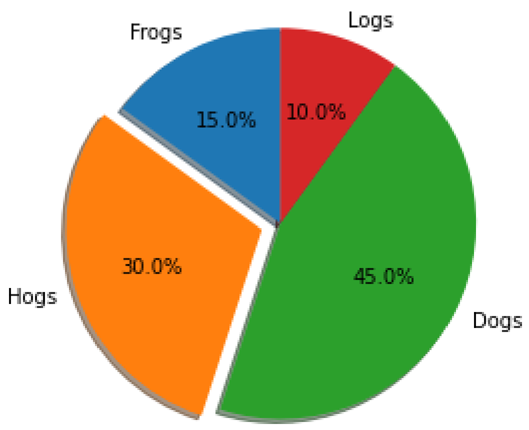
In [ ]:

```
# Plot a pie chart

pieLabels = ['Asia', 'Africa', 'Europe', 'North America', 'South America']
populationShare = [59.69, 16, 9.94, 7.79, 5.68]

figureObject, axesObject = plt.subplots()

axesObject.pie(populationShare,labels=pieLabels,autopct='%.1f%%',startangle=90, wedgeprops
axesObject.axis('equal')
plt.show()
```



In [ ]:

```
# Pie chart, where the slices will be ordered and plotted counter-clockwise:
labels = 'Frogs', 'Hogs', 'Dogs', 'Logs'
sizes = [15, 30, 45, 10]
explode = (0, 0.1, 0, 0)
# only "explode" the 2nd slice (i.e. 'Hogs')

plt.pie(sizes, explode=explode, labels=labels, autopct='%1.1f%%', shadow=True, startangle=9
plt.axis('equal')  # Equal aspect ratio ensures that pie is drawn as a circle.
plt.show()
```

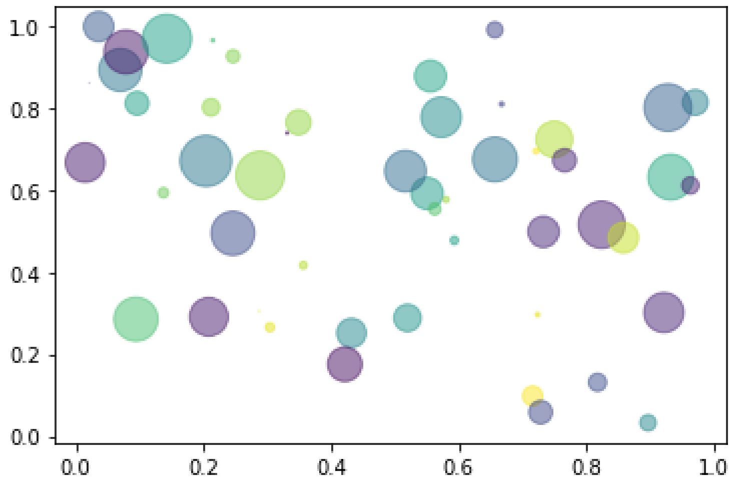


###Bubble Plots

Bubble plots are an improved version of the scatter plot. In a scatter plot, there are two dimensions x, and y. In a bubble plot, there are three dimensions x, y, and z. Where the third dimension z denotes weight. That way, bubble plots give more information visually than a two dimensional scatter plot.

In [ ]:

```
N = 50
x = np.random.rand(N)
y = np.random.rand(N)
colors = np.random.rand(N)
area = np.pi * (15 * np.random.rand(N))**2

# s - size of points, can be a single number or an array
# c - colour of the points
# alpha - transparency value of colours
plt.scatter(x, y, s=area, c=colors, alpha=0.5)
plt.show()
```



###**Histograms**

A histogram is a graph showing frequency distributions.

It is a graph showing the number of observations within each given interval.

In [ ]:

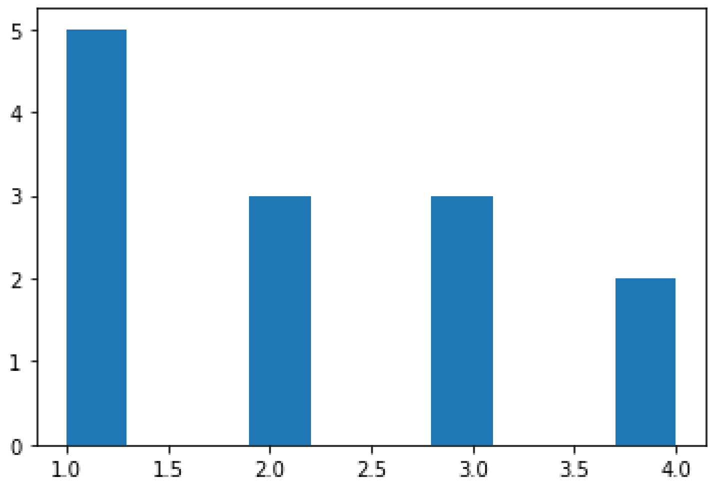
```
import numpy as np
```

In [ ]:

```
a = [1, 2, 3, 4, 1, 2, 3, 1, 2, 3, 1, 1, 4]
```

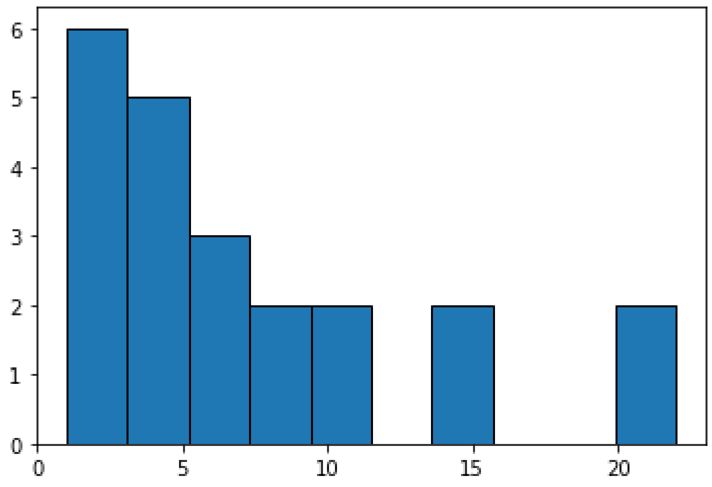
In [ ]:

```
plt.hist(a)
plt.show()
```



In [ ]:

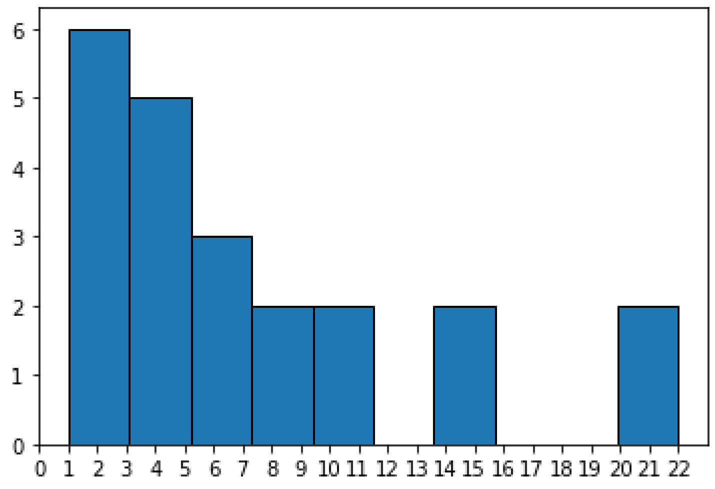
```
a = [1, 2, 1, 3, 5, 4, 5, 5, 4, 3, 3, 10, 7, 8, 8, 6, 7, 20, 22, 11, 14, 15]
plt.hist(a, edgecolor = 'black')
plt.show()
```



As you can observe, it has clubbed the frequency of few numbers inside a single box. These blue boxes are called bins. For a clearer look, lets see the following graph.

In [ ]:

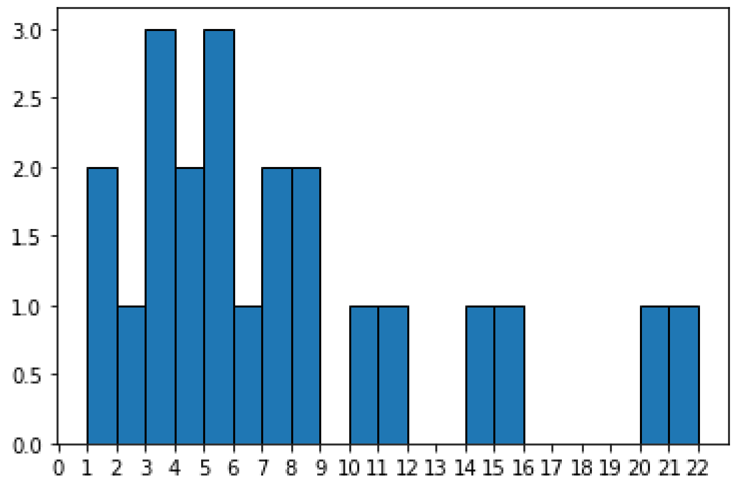
```
a = [1, 2, 1, 3, 5, 4, 5, 5, 4, 3, 3, 10, 7, 8, 8, 6, 7, 20, 22, 11, 14, 15]
plt.xticks(np.arange(23))
plt.hist(a, edgecolor = 'black')
plt.show()
```



This happens becasue the default number of bins in histogram is 10. Hence these numbers have been clubbed together. We can give our own bin number.

In [ ]:

```
a = [1, 2, 1, 3, 5, 4, 5, 5, 4, 3, 3, 10, 7, 8, 8, 6, 7, 20, 22, 11, 14, 15]
plt.xticks(np.arange(23))
plt.hist(a, bins = 21, edgecolor = 'black')
plt.show()
```



If bins is an integer, it defines the number of equal-width bins in the range.

If bins is a sequence, it defines the bin edges, including the left edge of the first bin and the right edge of the last bin; in this case, bins may be unequally spaced. All but the last (righthand-most) bin is half-open. In other words, if bins is:

[1, 2, 3, 4]

then the first bin is [1, 2) (including 1, but excluding 2) and the second [2, 3). The last bin, however, is [3, 4], which includes 4.

If bins is a string, it is one of the binning strategies supported by numpy.histogram_bin_edges: 'auto', 'fd', 'doane', 'scott', 'stone', 'rice', 'sturges', or 'sqrt'.

##**Bar Graphs**

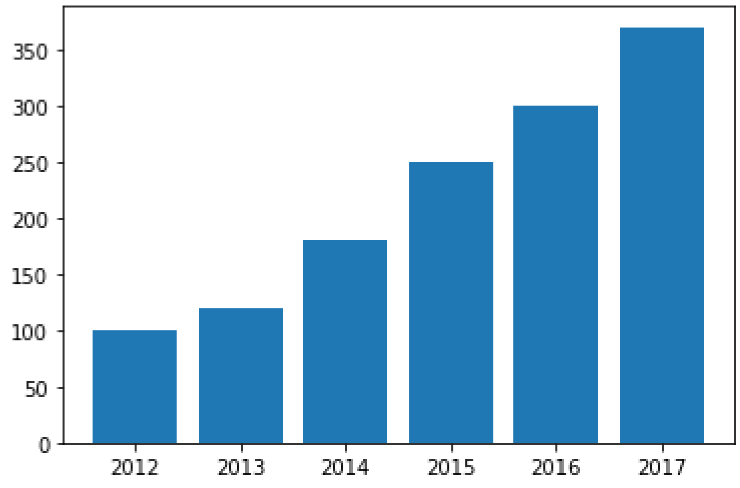
A bar chart or bar graph is a chart or graph that presents categorical data with rectangular bars with heights or lengths proportional to the values that they represent. The bars can be plotted vertically or horizontally. A vertical bar chart is sometimes called a column chart.

In [ ]:

```
year = [2012, 2013, 2014, 2015, 2016, 2017]
salary = [12, 13, 14, 17, 19, 20]
population = [100, 120, 180, 250, 300, 370]
```

In [ ]:

```
plt.bar(year, population)
plt.show()
```



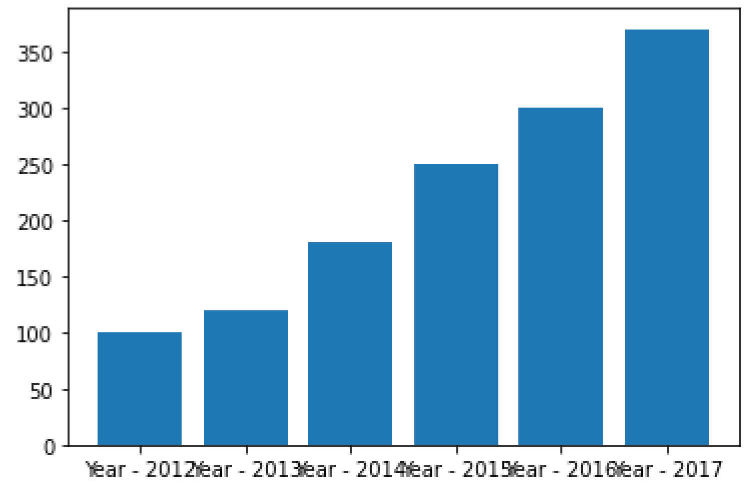
Suppose the years were like this:

In [ ]:

```
year = ['Year - 2012', 'Year - 2013', 'Year - 2014', 'Year - 2015', 'Year - 2016', 'Year -
```

In [ ]:

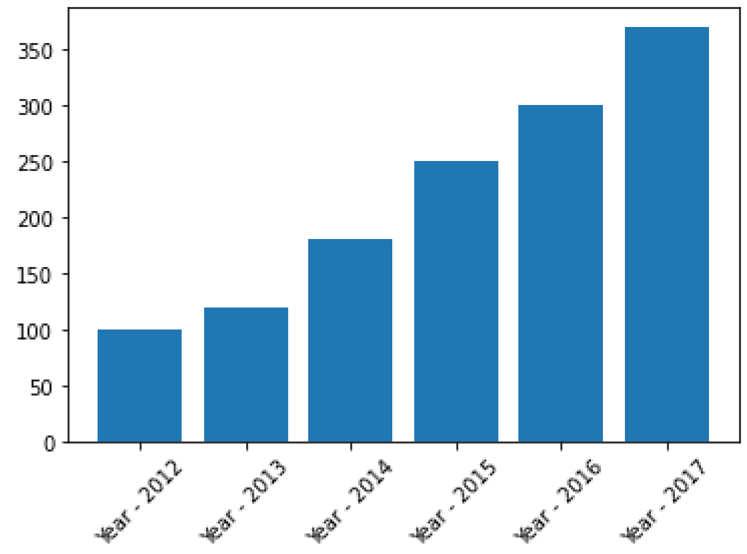
```
plt.bar(year, population)
plt.show()
```



The value in the axis have overlapped. Thus we need to rotate the xticks. This can be done using the following way:

In [ ]:

```
plt.bar(year, population)
plt.xticks(rotation = 45)
plt.show()
```



##Your Next Task

There are many more functionalities for altering the look and design of your graphs. Feel free to explore and try them out.