COMPUTER SCIENCE With python

Textbook for Class XII

- Programming & Computational Thinking
- Computer Networks
- Data Management (SQL, Django)
- Society, Law and Ethics

SUMITA ARORA

DHANPAT RAI & Co.

8 Data Visualization using Pyplot

251 – 295

8 Data Visualization using Pyplot

In This Chapter

8.1 WHAT IS DATA VISUALIZATION ?

Modern age has become very fast and competitive with the pace of the development in all fields of life. The reach of all businesses and facilities have gone global because of the modern technologies. This has made things easier and competitive but at the same time data have increased multifold. In fact, data have grown so big that a specific term has been coined, *'big data'*.

As you all are aware that the role of data is to empower decision makers to make decisions based on *facts, trends* and *statistical numbers* available in the form of data. But since data is so huge these days that the decision makers must be able to sift through the unnecessary, unwanted data and get the right information presented in compact and apt way, so that they can make the best decisions. For this purpose, *data visualization techniques* have gained popularity. Confused? What is this *data visualization*? Well, read on.

Data visualization basically refers to the graphical or visual representation of information and data using visual elements like *charts, graphs,* and *maps* etc. Data visualization is immensely useful in decision making. Data visualization unveils *patterns, trends, outliers, correlations* etc. in the data, and thereby helps decision makers understand the meaning of data to drive business decisions.

For instance, a company has to decide about, *'which advertising solution should it invest in to promote its new product?' Data visualization* is here to help – just pull out data of company's previous campaigns, and plot a bar chart comparing the performance of different platforms and bingo! the right decision is right in front.

This chapter is dedicated to get you started with data visualization using Python's useful tool **PyPlot** where you shall learn to represent data visually in various forms such as *line, bar* and *pie charts* etc. So, let's get started.

DATA VISUALIZATION

Data visualization basically refers to the graphical or visual representation of information and data using visual elements like *charts, graphs,* and *maps* etc.

8.2 USING PYPLOT OF MATPLOTLIB LIBRARY

For data visualization in Python, the Matplotlib library's Pyplot interface is used.

The **matplotlib** is a Python library that provides many interfaces and functionality for 2D-graphics similar to MATLAB[1]'s in various forms. In short, you can call ***matplotlib*** as a high quality plotting library of Python. It provides both a very quick way to visualize data from Python and publication-quality figures in many formats. The **matplotlib** library offers many different named collections of methods; **PyPlot** is one such interface.

NOTE

PyPlot is a collection of methods within **matplotlib** library (of Python) which allow user to construct 2D plots easily and interactively.

PyPlot is a collection of methods within **matplotlib** which allow user to construct 2D plots easily and interactively. **PyPlot** essentially reproduces plotting functions and behavior of MATLAB.

8.2.1 Installing and Importing *matplotlib*

You shall learn to plot data using **PyPlot** but before that make sure that **matplotlib** library is installed on your computer.

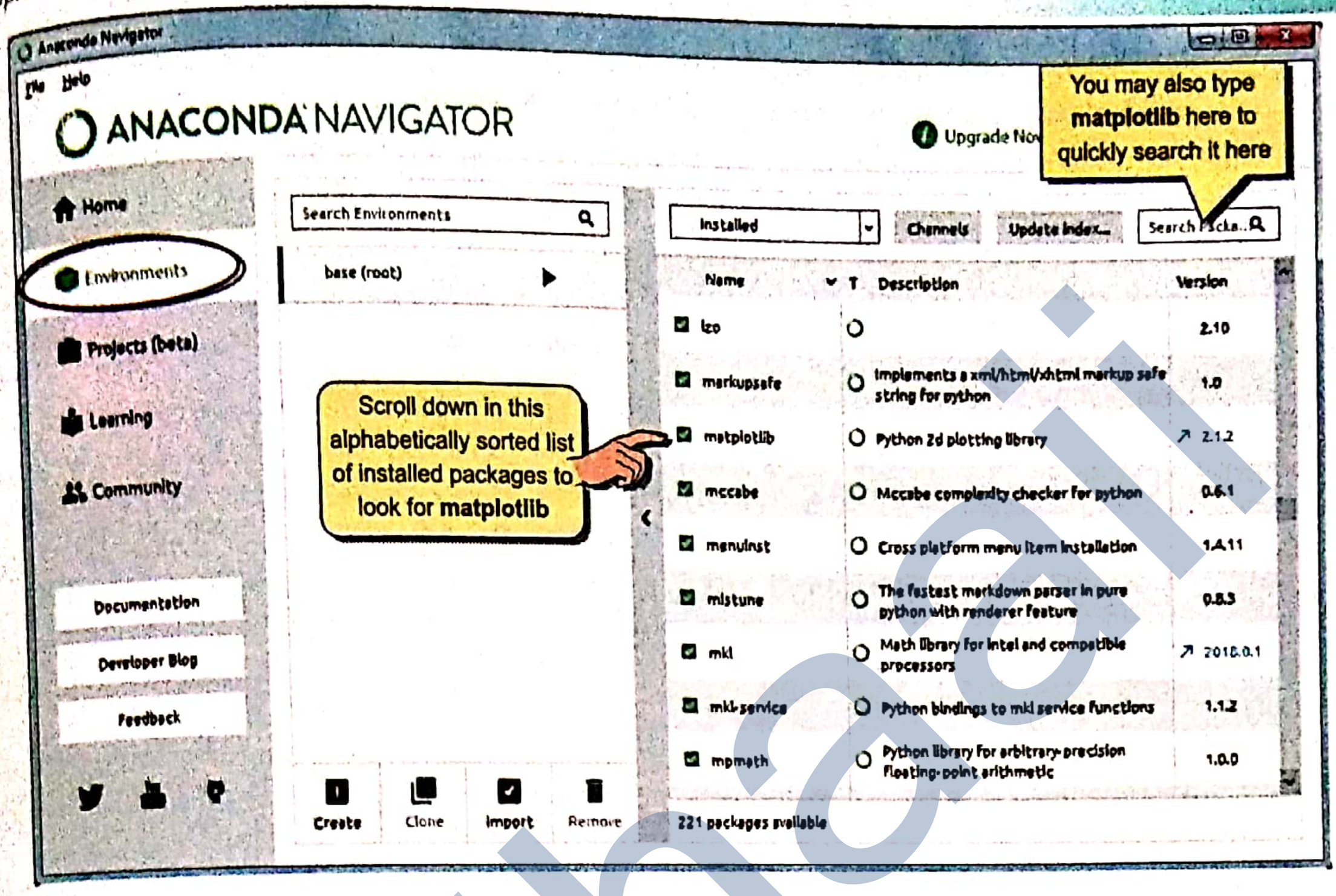
NOTE

The **matplotlib** library is preinstalled with **Anaconda distribution**; you need not install it separately if you have installed Python using Anaconda distribution.

- If you have installed Python using Anaconda, then **matplotlib** library is already installed on your computers. You can check it yourself by starting **Anaconda Navigator**. From Navigator window, click **Environments** and then scroll down in the alphabetically sorted list of installed packages on your computer. You will find **matplotlib** there.
- If you have installed Python using standard official distribution, you may need to install **matplotlib** separately as explained here.
  - First you will need to download wheel package of **matplotlib** as per Python's version installed and the platform (*MacOs or Windows or Linux*) on which it is installed. For this go to the link **https://pypi.org/project/matplotlib/#files** and download required wheel package as per your platform.
  - Next you need to install it by giving following commands :

```
python -m pip install -U pip
python -m pip install -U matplotlib
```

1. **MATLAB is a high-performance language for technical computing. It integrates computation, visualization, and programming in an easy-to-use environment where problems and solutions are expressed in familiar mathematical notation.**



Importing PyPlot

In order to use **pyplot** on your computers for data visualization, you need to first import it in your Python environment by issuing one of the following commands :

```
import matplotlib.pyplot
```

This would require you to refer to every command of ***pyplot*** *as* ***matplotlib.pyplot.<command>***

```
import matplotlib.pyplot as pl
```

With this, you can refer to every command of ***pyplot*** *as* ***pl.<command>*** *as you have given an alias name to* ***matplotlib.pyplot*** *as* ***pl***

With the *first command* above, you will need to issue every **pyplot** command as per following syntax :

```
matplotlib.pyplot.<command>
```

But with the *second command* above, you have provided **pl** as the shorthand for **matplotlib.pyplot** and thus now you can invoke PyPlot's methods as this :

```
pl.plot(X,Y)
```

You can choose any legal identifier in place of **pl** above.

8.2.2 Working with PyPlot Methods

The PyPlot interface provides many methods for 2D plotting of data. The *matplotlib's* Pyplot interface lets one plot data in multiple ways such as *line chart, bar chart, pie chart, scatter chart* etc. Let us talk about how you can plot sequence or array of data using PyPlot methods.

But before we proceed with it, I shall recommend you to have an idea of using NumPy library and some of its functions, for these two reasons, specifically :

(*i*) NumPy offers some other useful functions to create arrays of data, which prove useful while plotting data.

(*ii*) NumPy offers many useful functions and routines.

(*iii*) NumPy arrays which are like lists, support ***vectorized operations***, *i.e.*, if you apply a function, it is performed on every item (element by element) in the array unlike lists *e.g.*,

Let's suppose you want to add the number 2 to every item in the list or array and if you give an expression such as **List +2** *(where List is a Python list), Python will give error but not with NumPy array. For a NumPy array, it will simply add 2 to each element of the NumPy array.*

This feature may prove very useful in plotting, e.g., if you have a NumPy array namely **X** *(say., ([1,2,3, 4])), and you want to compute* **sin( )** *for each of the values of this array for plotting purpose, you just need to write is* ***numpy.sin(X)***

For these reasons, I recommend that you should know how to create NumPy arrays and use NumPy functions on them. Following section briefly introduces how to create and use NumPy arrays.

8.2.2A Using NumPy Arrays

NumPy (*'Numerical Python'* or *'Numeric Python'*, pronounced as *Num Pie*) is an open source module of Python that offers functions and routines for fast mathematical computation on arrays and matrices. In order to use NumPy, you must import in your module by using a statement like :

```
import numpy as np
```

← *You can use any identifier name in place of np but np has been a preferred choice, generally*

The above import statement has given **np** as an alias name for NumPy module. Once imported with as <name>, you use both names *i.e.*, **numpy** or **np** for functions *e.g.*, **numpy.array( )** is same as **np.array( )**.

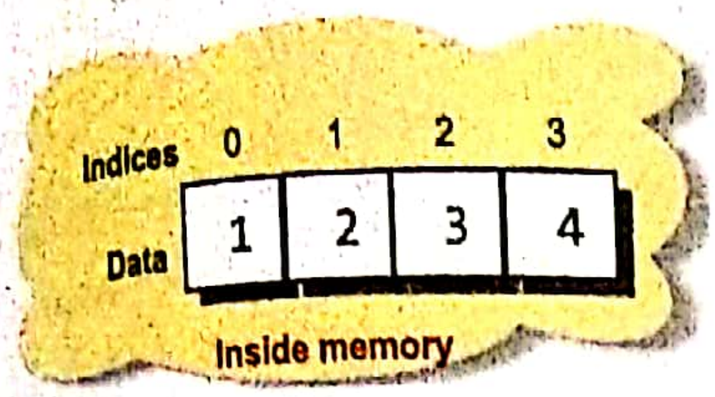
Array in general refers to a named group of homogeneous (of same type) elements. For example, if you store the similar details of all sections together such as if you store *number of students* in each section of class X in a school, *e.g.*, ***Students*** array containing 5 entries as [34, 37, 36, 41, 40] then ***Students*** is an array that represents number of students in each section of class X. Like lists, you can access individual elements by giving *index* with array name *e.g.*, **Students[1]** will give details about 2nd section, **Students[3]** will give you details about 4th section and so on.

NOTE

Please note that if you have installed Python through Anaconda installer then *NumPy, SciPy, Pandas, Matplotlib* etc. are by default installed with Python, otherwise you need to install these separately through **pip** installer of Python.

A NumPy array is simply a grid that contains values of the same/homogeneous type. You can think of a NumPy array like a *Python list* having all elements of similar types (but NumPy arrays are different from Python lists in functionality, which will be clear to you in coming lines).

Consider following code :



Code

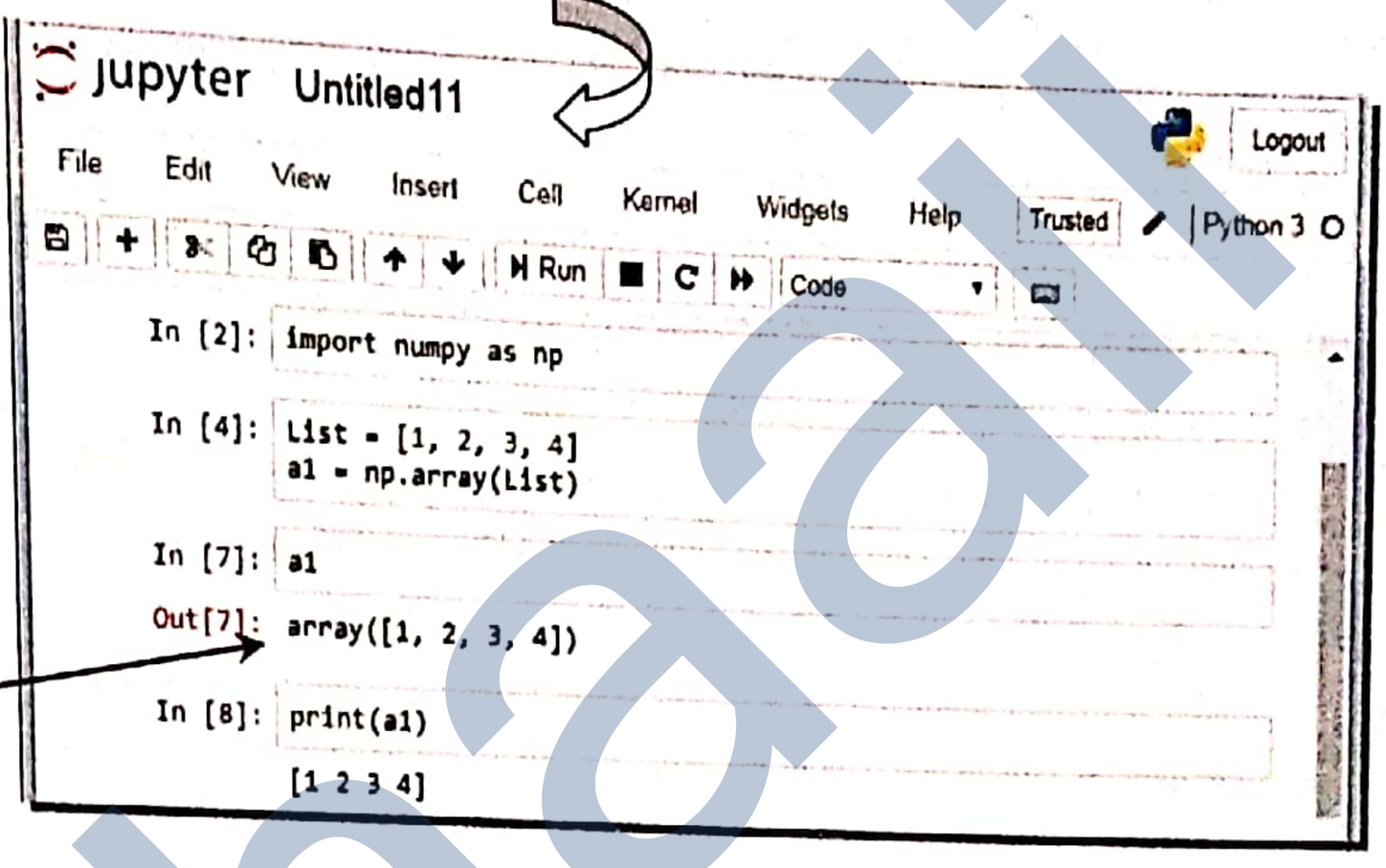
```
import numpy as np
List = [1, 2, 3, 4]
a1 = np.array(List)
print(a1)
```

Now, you can use numpy functions either as ***numPy****.functionName or* ***np****.functionName , e.g., numpy.array( ) or np.array( )*

It will create a NumPy array from the given list

```
In [13]: import numpy as np

In [14]: List = [1, 2, 3, 4]

In [15]: a1 = np.array(List)

In [16]: a1
Out[16]: array([1, 2, 3, 4])

In [17]: print(a1)
[1 2 3 4] .
```

Notice how it displays array and how it prints it with print( )



Individual elements of above array can be accessed just like you access a list's, *i.e.*,

```
<array-name>[<index>]
```

That is, ***a1[0]*** will give you ***1***, ***a1[1]*** will give you 2, ...***a1[3]*** will give you ***4***.

Following Fig. 8.1 illustrates the basic terms associated with a NumPy array :

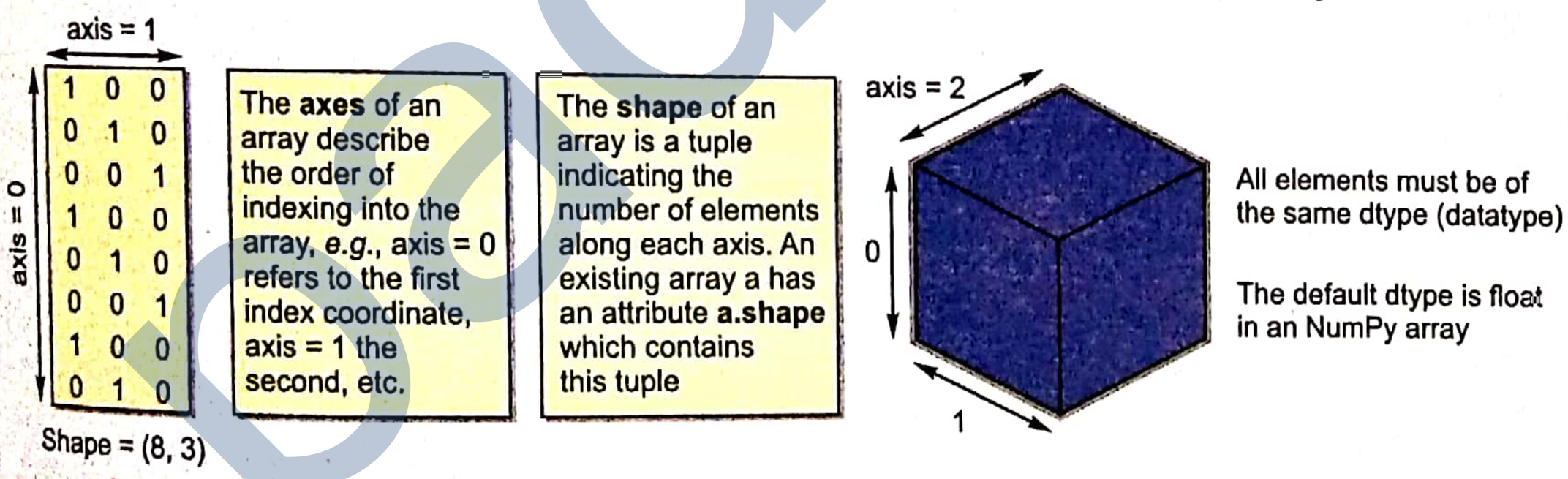


Figure 8.1 Anatomy of a NumPy Array

You can check *type* of a NumPy array in the same way you check the type of objects in Python, *i.e.*, using type( ). Also, there are associated attributes : (*i*) shape that returns dimensions of a NumPy array and (*ii*) itemsize that returns the length of each element of array in bytes. (see below)

```
In [25]: type (a1), type(a2)
Out[25]: (numpy.ndarray, numpy.ndarray)

In [26]: a1.shape
Out[26]: (4,)

In [27]: a2.shape
Out[27]: (2, 3)

In [28]: a2.itemsize
Out[28]: 4
```

See the type it returned for NumPy arrays you created

The **shape** attribute gives the dimensions of a NumPy array. For 1D array, see it returned as (4,) I.e., 4 elements , single index only

For 2D arrays, it returned both dimensions.

The **itemsize** attribute returns the length of each element of array in bytes

You can check data type of a NumPy array's elements **using <arrayname>.dtype** *e.g.*, (see on the right)

```
In [11]: a1.dtype
Out[11]: dtype('int32')
```

Some common NumPy data types (used as numpy.<datatype>) are being given on Table 8.1.

NOTE

Numpy arrays are also called ndarrays

Table 8.1 *NumPy Data Types*

S.No.	Data Type	Description	Size
1.	`numpy.int8`	Stores signed integers in range –128 to 127	1 byte
2.	`numpy.int16`	Stores signed integers in range –32768 to 32767	2 bytes
3.	`numpy.int32`	Stores signed integers in range -2^{16} to $2^{16}-1$	4 bytes
4.	`numpy.int64`	Stores signed integers in range -2^{32} to $2^{32}-1$	8 bytes
5.	`numpy.float_`	Default type to store floating point (*float64*)	8 bytes
6.	`numpy.float16`	Stores **half precision floating point values** (*5 bits* exponent, *10 bit* mantissa)	2 bytes
7.	`numpy.float32`	Stores **single precision floating point values** (*8 bits* exponent, *23 bit* mantissa)	4 bytes
8.	`numpy.float64`	Stores **double precision floating point values** (*11 bits* exponent, *52 bit* mantissa)	8 bytes

Some useful ways of creating NumPy arrays

(*i*) Creating arrays with a numerical range using arange( ). The arange( ) function is similar to Python's **range( )** function but it returns an *ndarray* in place of Python list returned by **range( )** of Python. In other words, the arange( ) creates a NumPy array with evenly spaced values within a specified numerical range. It is used as :

```
<arrayname> = numpy.arange([start,] stop [, step ] [, dtype ])
```

- The **start, stop** and **step** attribute provide the values for *starting value, stopping value* and *step value* for a numerical range. ***Start*** and ***step values*** are optional. When only *stop value* is given, the numerical range is generated from *zero* to *stop value* with *step* 1.
- The **dtype** specifies the datatype for the NumPy array.

Consider the following statements :

```
In [73]: arr5 = np.arange(7)

In [74]: arr5
Out[74]: array([0, 1, 2, 3, 4, 5, 6])

In [75]: arr5.dtype
Out[75]: dtype('int32')
```

Start value *Stop value* *Step value*

```
In [77]: arr6 = np.arange(1,7, 2, np.float32)

In [78]: arr6
Out[78]: array([1., 3., 5.], dtype=float32)
```

(*ii*) Creating arrays with a numerical range using linspace( ). Sometimes, you need evenly spaced elements between two given limits. For this purpose, NumPy provides linspace( ) function to which you provide, *the start value, the end value* and number of elements to be generated for the *ndarray*. The syntax for using linspace( ) function is :

```
<arrayname> = numpy.linspace(<start>, <stop>, <number of values to be generated>)
```

For example, following code will create an ndarray with 6 values falling in the range 2 to 3 :

```
Arr1 = np.linspace(2, 3, 6)
```

Consider some more examples :

Start value *Stop value* *No. of elements*

```
In [25]: a1 = np.linspace(2.5, 5, 6)

In [26]: a1
Out[26]: array([2.5, 3. , 3.5, 4. , 4.5, 5. ])

In [27]: a2 = np.linspace(2.5, 5, 8)

In [28]: a2
Out[28]:
array([2.5       , 2.85714286, 3.21428571, 3.57142857, 3.92857143,
       4.28571429, 4.64285714, 5.        ])
```

Ndarray with 6 values falling In range 2.5 to 5

Ndarray with 8 values falling In range 2.5 to 5

Some useful NumPy functions

We are giving some NumPy functions (without much details) that may prove useful while plotting data.

Trigonometric functions	
sin(x, /[, out])	Trigonometric sine, element-wise.
cos(x, /[, out])	Cosine element-wise.
tan(x, /[, out])	Compute tangent element-wise.
arcsin(x, /[, out])	Inverse sine, element-wise.
arccos(x, /[, out])	Trigonometric inverse cosine, element-wise.
arctan(x, /[, out])	Trigonometric inverse tangent, element-wise.
Hyperbolic functions	
sinh(x, /[, out])	Hyperbolic sine, element-wise.
cosh(x, /[, out])	Hyperbolic cosine, element-wise.
tanh(x, /[, out])	Compute hyperbolic tangent element-wise.
arcsinh(x, /[, out])	Inverse hyperbolic sine element-wise.
arccosh(x, /[, out])	Inverse hyperbolic cosine, element-wise.
arctanh(x, /[, out])	Inverse hyperbolic tangent element-wise.
Rounding Functions	
round_(a[, decimals, out])	Round an array to the given number of *decimals.*
floor(x, /[, out])	Return the floor of the input, element-wise.
ceil(x, /[, out])	Return the ceiling of the input, element-wise.
trunc(x, /[, out])	Return the truncated value of the input, element-wise.
Exponents and logarithms	
exp(x, /[, out])	Calculate the exponential of all elements in the input array.
exp2(x, /[, out])	Calculate 2**p for all p in the input array.
log(x, /[, out])	Natural logarithm, element-wise.
log10(x, /[, out])	Return the base 10 logarithm of the input array, element-wise.
log2(x, /[, out])	Base-2 logarithm of x.
log1p(x, /[, out ...])	Return the natural logarithm of one plus the input array, element-wise.

Parameters used in the above functions :

x : *array_like sequence*

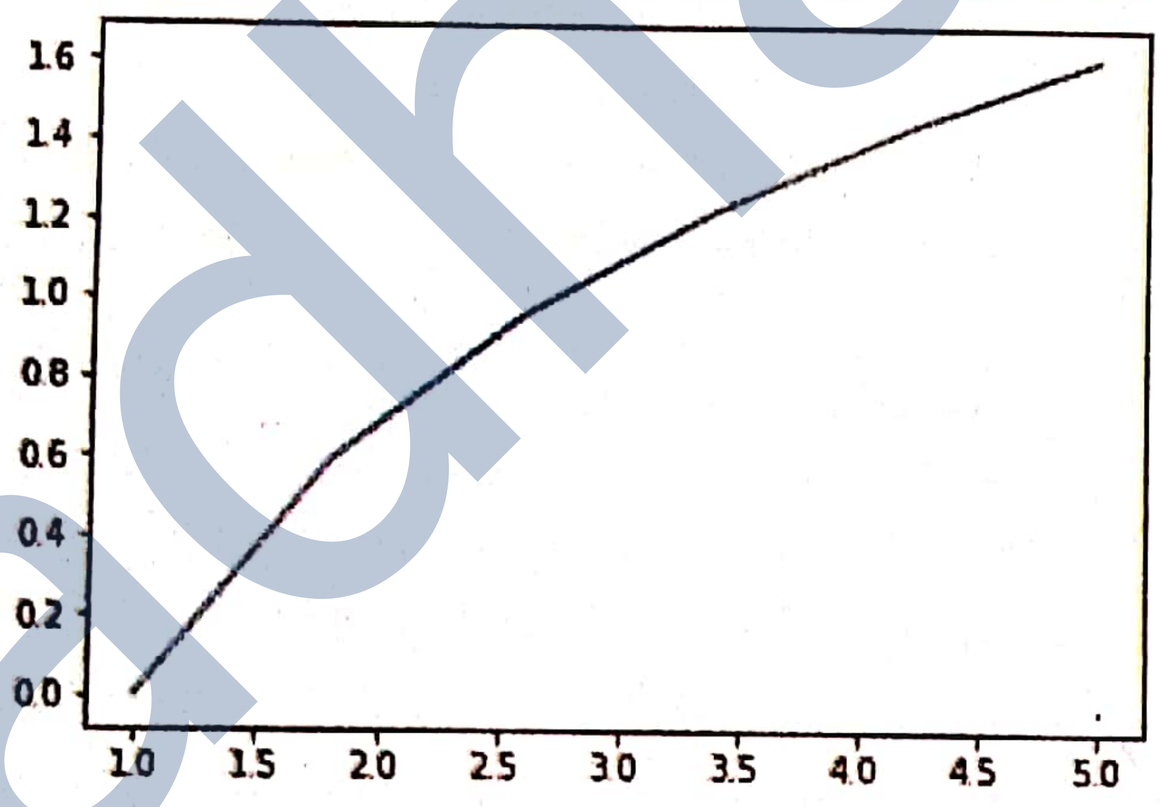
out : *ndarray, None, or tuple of ndarray and None, optional argument*

it is a location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned.

See below how useful the numpy functions are while plotting. Although you shall learn about all components of plotting in the coming section, yet following code lines make you appreciate the use of numpy arrays and functions. (Don't worry if things are not clear right now, they will be, soon. I promise ☺)

```
In [39]: import numpy as np

In [40]: import matplotlib.pyplot as pl

In [41]: x = np.linspace(1,5, 6)

In [42]: y = np.log(x)

In [43]: pl.plot(x,y)
Out[43]: [<matplotlib.lines.Line2D at 0x8ef79d0>]
```

Generate an array in range 1..5, having 6 elements



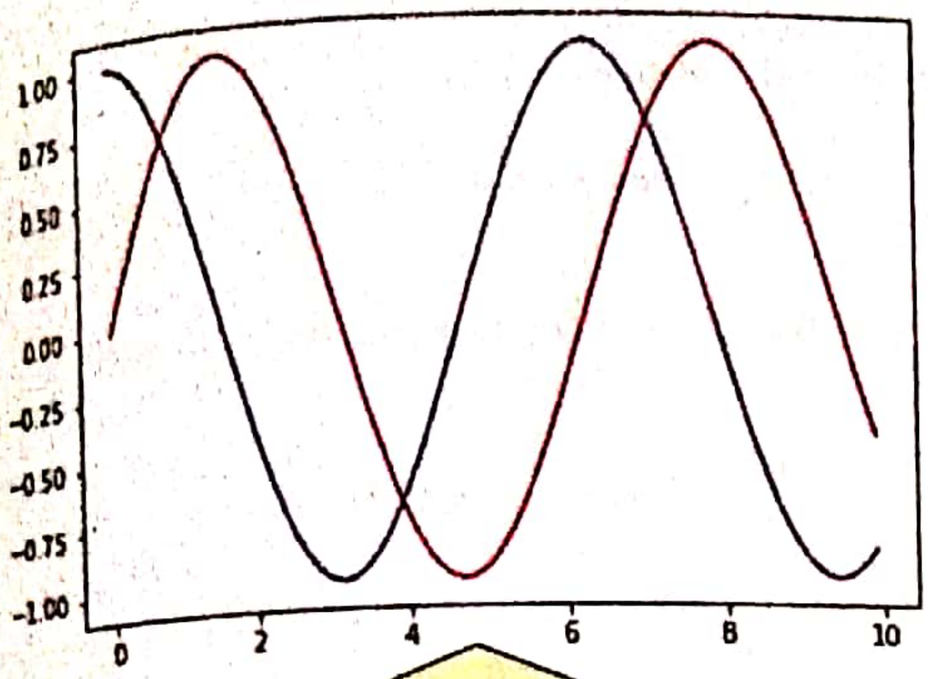
8.2.2B Basics of Simple Plotting

Data visualization essentially means graphical representation of compiled data. Thus, graphs and charts are very effective tools for data visualization. You can create many different types of graphs and charts using PyPlot but we shall stick to just a few of them as per your syllabus.

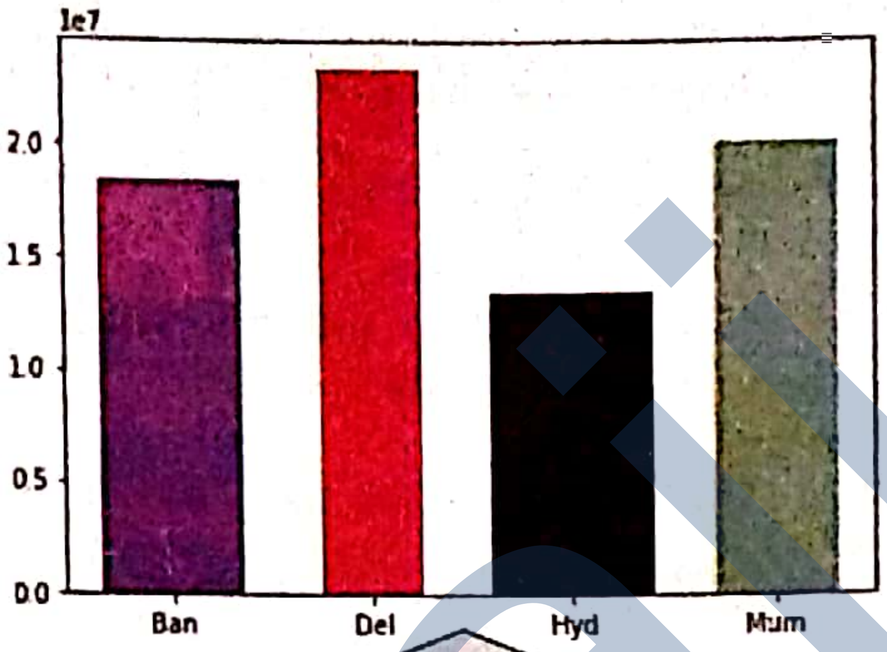
Some commonly used chart types are :

- **Line Chart.** A line chart or line graph is a type of chart which displays information as a series of data points called 'markers' connected by straight line segments.
- **Bar Chart.** 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.
- **Pie Chart.** A pie chart (or a circle chart) is a circular statistical graphic, which is divided into slices to illustrate numerical proportion.

Some commonly used chart types are shown in figure below.



A line chart or line graph is a type of chart which displays information as a series of data points called **'markers'** connected by straight line segments.



A bar chart is a chart that presents categorical data with rectangular bars with heights proportional to the values that they represent.

A pie chart (or a circle chart) is a circular statistical graphic, which is divided into slices to illustrate numerical proportion

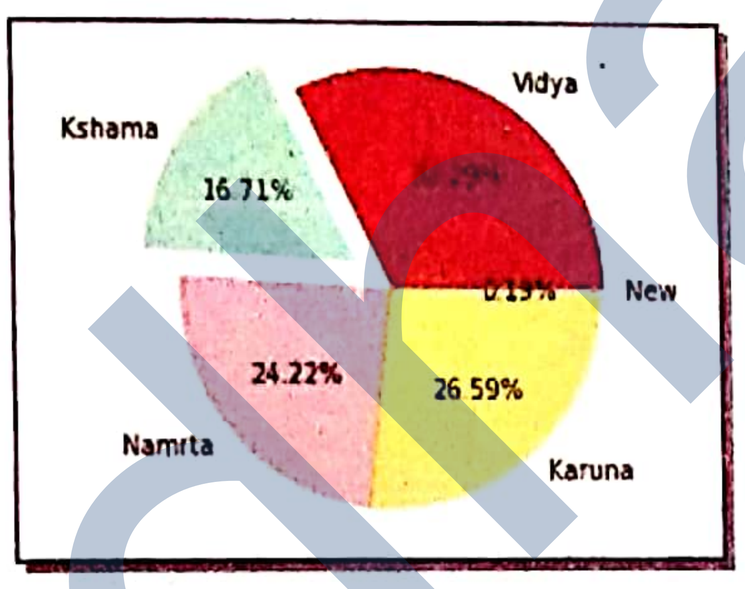


Figure 8.2 Some commonly used chart types.

Following section talks about how you can create various chart types using pyplot methods..

8.3 CREATING CHARTS WITH MATPLOTLIB LIBRARY'S PYPLOT INTERFACE

You know that graphs and charts play a big and an important role in data visualization and decision making. Every chart type has a different utility and serves a different purpose.

Let us talk about how you can create *line, bar* and *pie charts* using *matplotlib.pyplot* library of Python.

As stated earlier make sure to **import matplotlib.pyplot** library interface and other required libraries before you start using any of their functions.

8.3.1 Line Chart

A line chart or line graph is a type of chart which displays information as a series of data points called *'markers'* connected by straight line segments. The PyPlot interface offers **plot( )** function for creating a line graph. Carefully go through the example codes given below to understand the working of **plot( )**.

LINE CHART

A **line chart** or **line graph** is a type of chart which displays information as a series of data points called *'markers'* connected by straight line segments.

Say we have following three lists, namely *a*, *b* and *c* :

```
a = [1, 2, 3, 4]
b = [2, 4, 6, 8]
c = [1, 4, 9, 16]
```

List b containing values as double of values in list a

List c containing values squares of values in list a

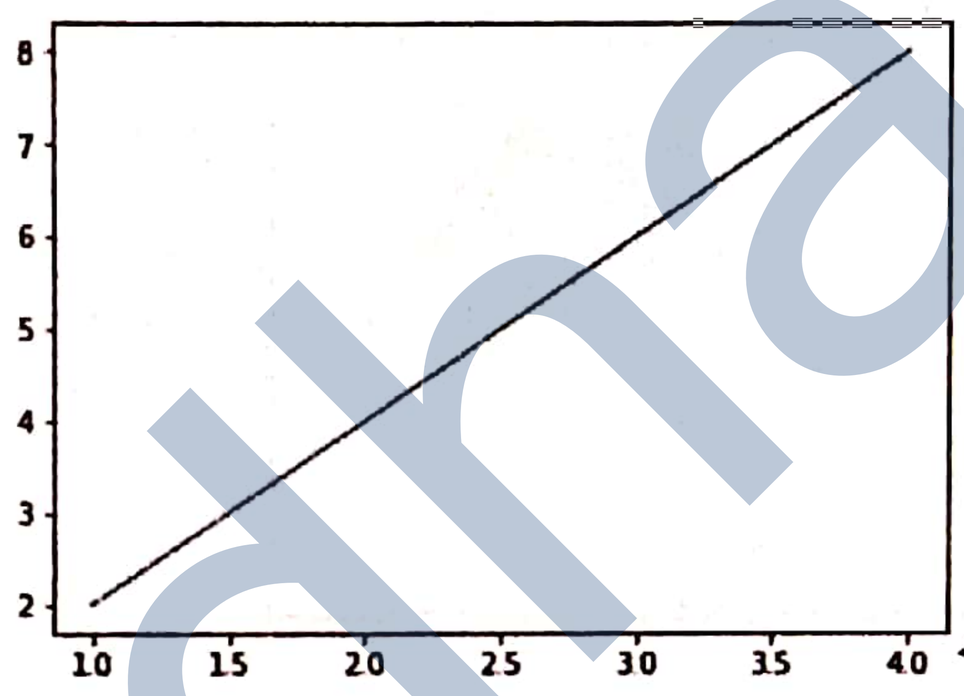
Now, if you want to plot a line chart for all values of ***list a*** vs values of ***list b***, then, in simplest form you may write :

```
import matplotlib.pyplot as pl
pl.plot(a, b)
```

The import statement is to be given just once

And python will show you result as :

```
In [28]: pl.plot(a,b)
Out[28]: [<matplotlib.lines.Line2D at 0xd277e30>]
```



The values of ***list b*** are plotted on vertical axis

The values of ***list a*** are plotted on horizontal axis

NOTE

Data points are called ***markers***

But are you satisfied with the result? Shouldn't the axes show the labels of the axes ? Let us say we want to label the **x-axis**, the **horizontal axis**, as '**Some values**' and the y-axis, the vertical axis as '**Doubled values**'. You can set *x*-axis' and *y*-axis' labels using functions **xlabel( )** and **ylabel( )** respectively, *i.e.*, :

NOTE

Before using any **PyPlot function** or **numpy function**, make sure that these libraries are imported beforehand.

```
<matplotlib.pyplot or its alias>.xlabel(<str>)
```

and

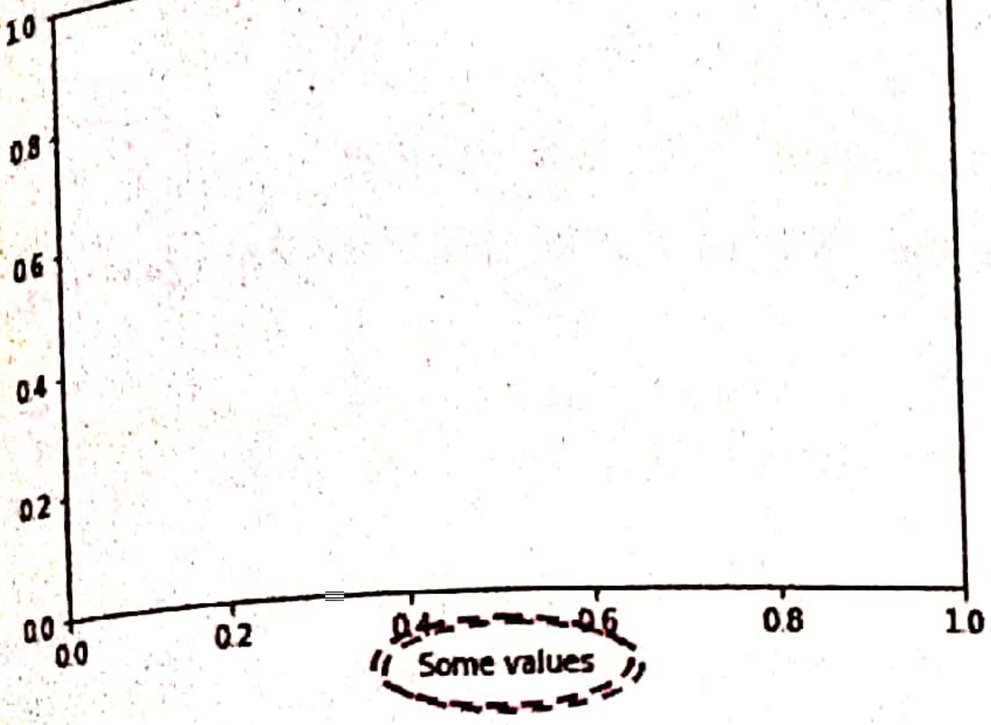
```
<matplotlib.pyplot or its alias>.ylabel(<str>)
```

So you may write (since we used identifier **pl** as alias for **matplotlib.pyplot** in **import** statement) :

```
pl.xlabel("Some Values")
pl.ylabel("Doubled Values")
pl.plot(a, b)
```

Let us see, what Python does.

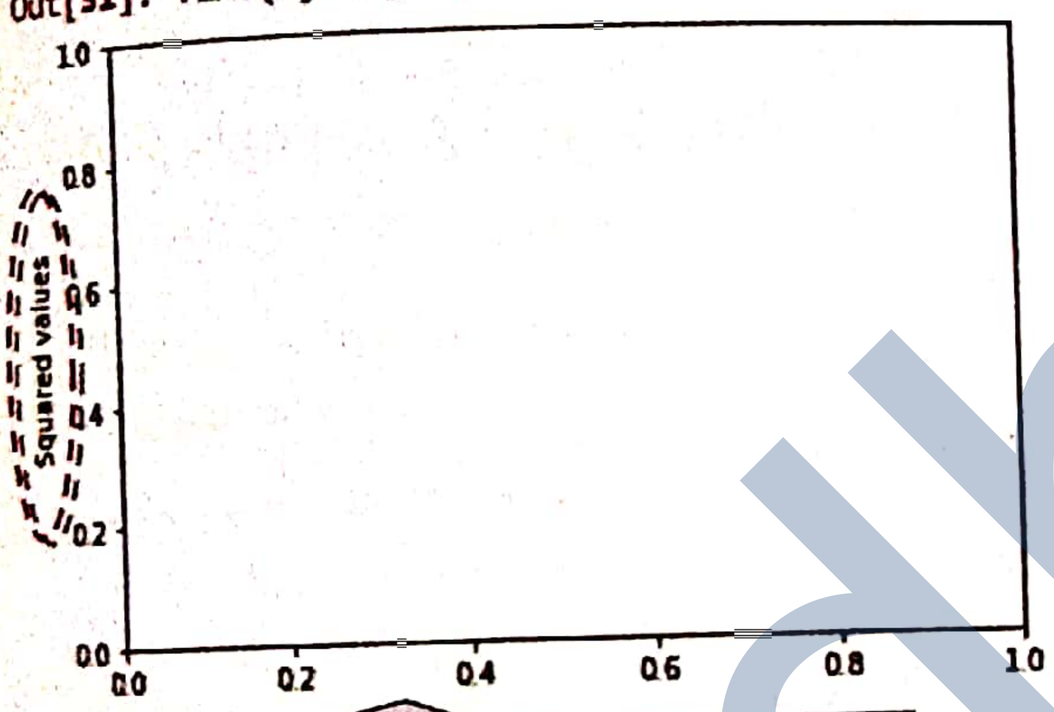
```
In [30]: pl.xlabel('Some values')
Out[30]: Text(0.5,0,'Some values')
```



See, on IPython console's prompt, the three statements have produced three charts, even if you wanted a single chart

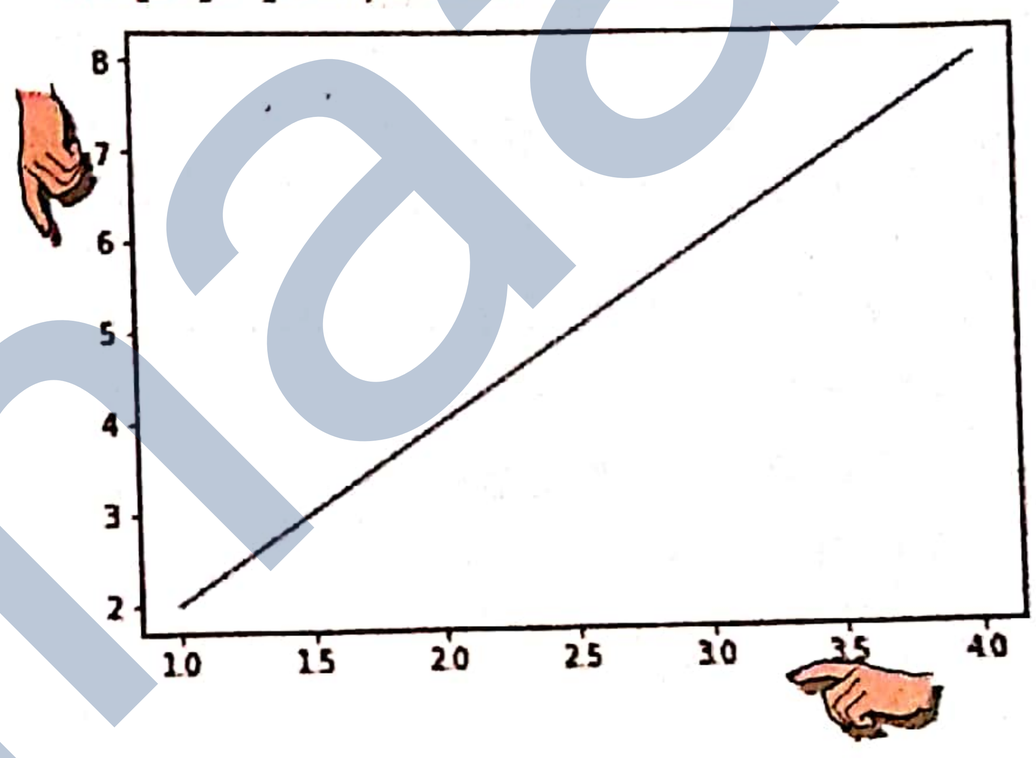
No axes titles because Python has not considered any of the previous statements

```
In [31]: pl.ylabel('Squared values')
Out[31]: Text(0,0.5,'Squared values')
```



No X-axis title only *Y-axis title* as only current statement is considered

```
In [32]: pl.plot(a,b)
Out[32]: [<matplotlib.lines.Line2D at 0xd42b390>]
```



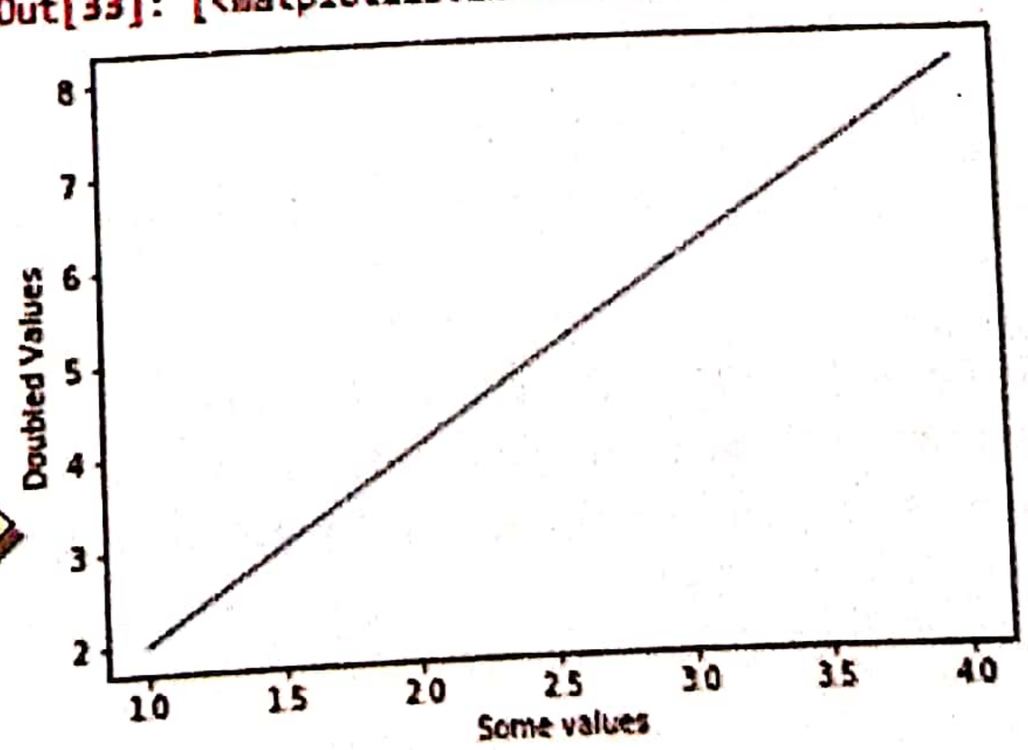
See, Python has shown us *three* different charts, one for each statement we issued. But this is not what we wanted. We wanted all these three statements to take effect in a single plot.

On the prompt, if you give these *three* statements, Python interactively draws plot for each statement separately. **To apply multiple statements on a single plot**, do one of the following :

(*i*) Either write a script, store it with **.py** extension and then run it

(*ii*) **Or**, on the *IPython console* prompt, press **CTRL + ENTER** to end each statement and **ENTER** in the end. This will group multiple statements and execute them as one unit, *i.e.*,

```
In [33]: pl.xlabel('Some values')      <- Press CTRL+ENTER here
   ...: pl.ylabel('Doubled Values')
   ...: pl.plot(a, b)                  <- Press ENTER in the end
   ...:
Out[33]: [<matplotlib.lines.Line2D at 0xd464b70>]
```



Now all these statements have impacted one plot as you can see the labels on both X and Y axes and graph plotted in the same plot

But the first method of writing plotting commands in a script(.py extension) is often a preferred choice for many. Also, in a script, use **<matplotlib.pyplot>.show( )** command in the end to show a plot as per given specifications.

8.3.1A Changing Line Style, Line Width and Line Color in a Line Chart

While plotting in a line chart, you can control the look of the plotted line by specifying its features like :

- Line width, Line color, line style
- Line marker type, size
- etc.

To change line color, you can specify the color code next to the data being plotting in plot() function as shown below :

```
<matplotlib>.plot(<data1>, [,data2], <color code> )
```

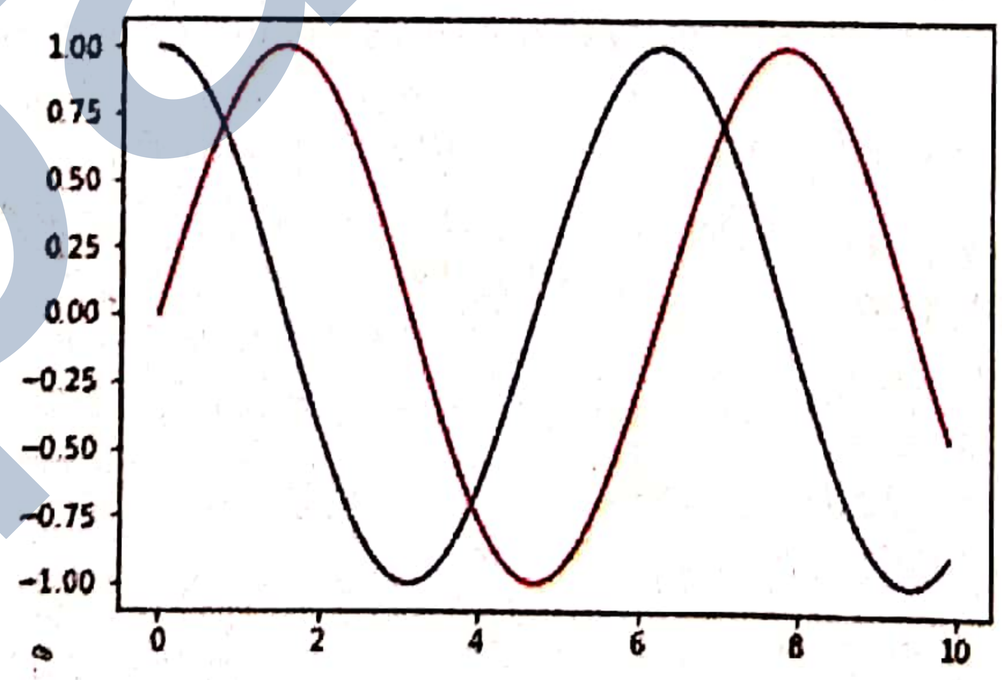
You can use color codes as: **'r'** for red, **'y'** for yellow, 'g' for green, 'b' for blue etc. (Complete color codes list has been given in Table 8.2), *e.g.*,

```
import matplotlib.pyplot as plt
import numpy as np
x = np.arange(0., 10, 0.1)
a = np.cos(x)
b = np.sin(x)
plt.plot(x, a, 'b')
plt.plot(x, b, 'r')
plt.show()
```

Color code blue for ***ndarray a***

Color code red for ***ndarray b***

The output produced by above script is as shown below :



Please note that even if you skip the color information in *plot( )*, Python will plot multiple lines in the same plot with different colors but these colors are decided internally by Python.

NOTE

Even if you skip the color information in **plot( )**, Python will plot multiple lines in the same plot with different colors but these colors are decided internally by Python.

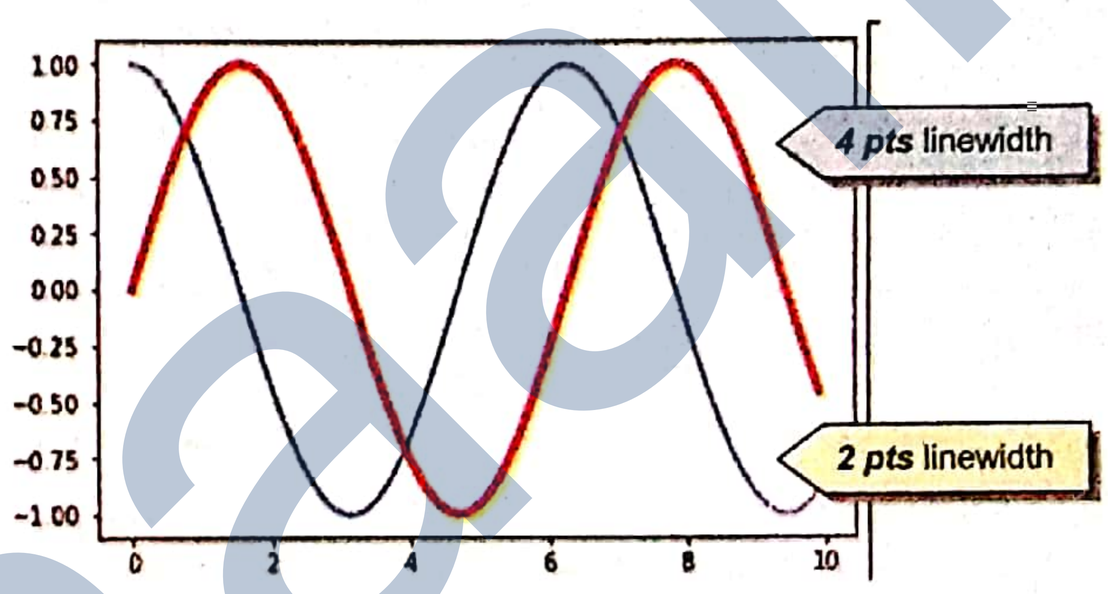
Table 8.2 *Different Color Codes*

character	color	character	color	character	color
'b'	blue	'm'	magenta	'c'	cyan
'g'	green	'y'	yellow	'w'	white
'r'	red	'k'	black		

In addition, you can also specify colors in many other ways, including full color names (*e.g.*, 'red', 'green' etc.), hex strings ('#008000') etc.

To change Line width, you can give additional argument in *plot( )* as **linewidth = <width>** where you have to specify the width value in points[2], *e.g.*,

```
: #x, a, b are same as earlier code
plt.plot(x, a, linewidth = 2)
plt.plot(x, b, linewidth = 4)
```



The result produced by above code is as shown here :

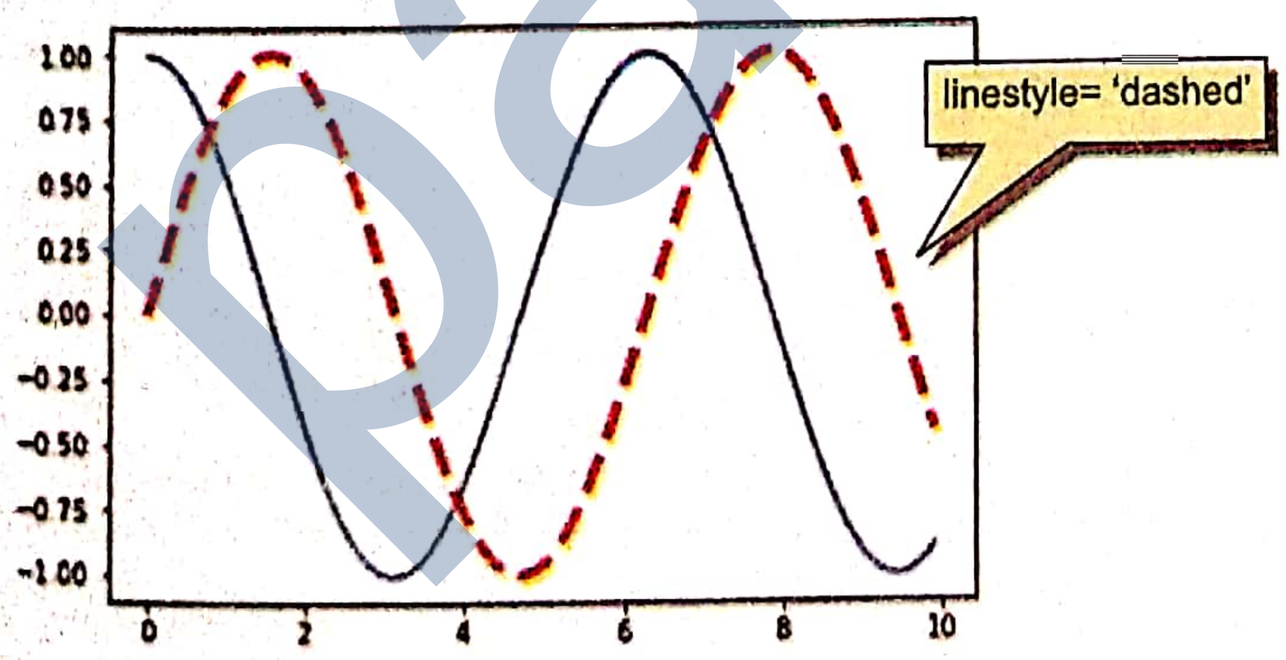
To change the line style, you can add following additional optional argument in *plot( )* function :

```
linestyle or ls = ['solid' | 'dashed', 'dashdot', 'dotted']
```

e.g.,

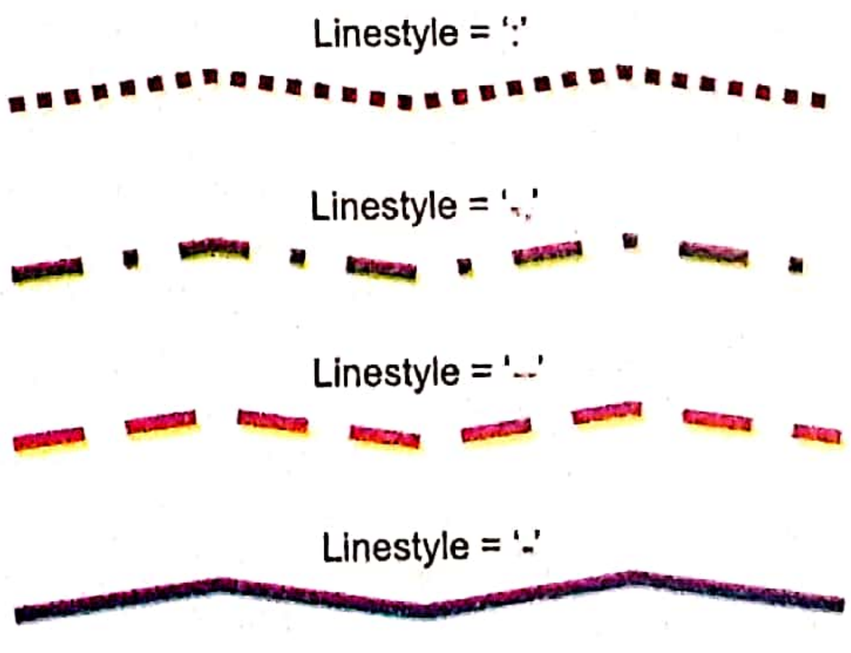
```
: #x, a, b are same as earlier code
plt.plot(x, a, linewidth = 2)
plt.plot(x, b, linewidth = 4, linestyle = 'dashed')
```

The result produced by above code is as shown here :



NOTE

Possible line styles are : **'solid'** for *solid line*, **'dashed'** for *dashed line*, **'dotted'** for *dotted line* and **'dashdot'** for *dashdotted line*.



8.3.1B Changing Marker Type, Size and Color

Recall that *data points* being plotted are called **markers**. To change *marker type*, its *size* and color, you can give following additional optional arguments in *plot( )* function :

```
marker = <valid marker type> , markersize = <in points>, markeredgecolor = <valid color>
```

2. Line width or thickness is measured in points e.g., 0.75 points, 2.0 points etc.

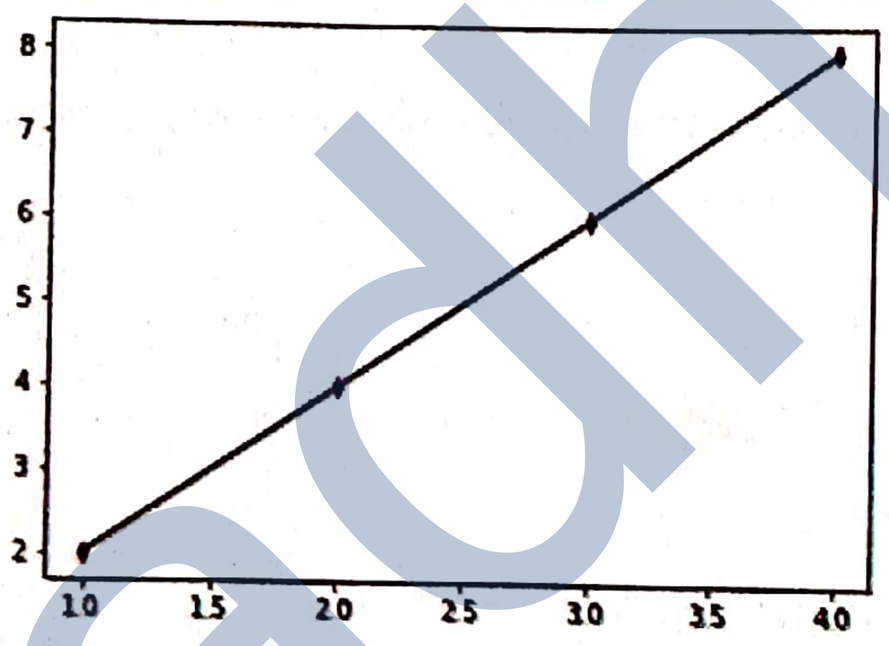
You can control the *type of marker i.e., dots or crosses or diamonds* etc. by specifying desired marker type from the table below. If you do not specify marker type, then data points will not be marked specifically on the line chart and its default type will be same as that of the line type. Also, the ***markersize*** is to be specified in points and ***markeredgecolor*** must be a valid color.

Table 8.3 *Marker Types for Plotting*

marker	description	marker	description	marker	description
'.'	point marker	's'	square marker	'3'	tri_left marker
','	pixel marker	'p'	pentagon marker	'4'	tri_right marker
'o'	circle marker	'*'	star marker	'v'	triangle_down marker
'+'	plus marker	'h'	hexagon1 marker	'^'	triangle_up marker
'x'	x marker	'H'	hexagon2 marker	'<'	triangle_left marker
'D'	diamond marker	'1'	tri_down marker	'>'	triangle_right marker
'd'	thin_diamond marker	'2'	tri_up marker	'|', '_'	vline, hline markers

Now consider following example where p = [1, 2, 3, 4] and q = [2, 4, 6, 8]

```
In [61]: plt.plot(p, q, 'k', marker='d', markersize = 5, markeredgecolor ='red')
Out[61]: [<matplotlib.lines.Line2D at 0xd09c310>]
```

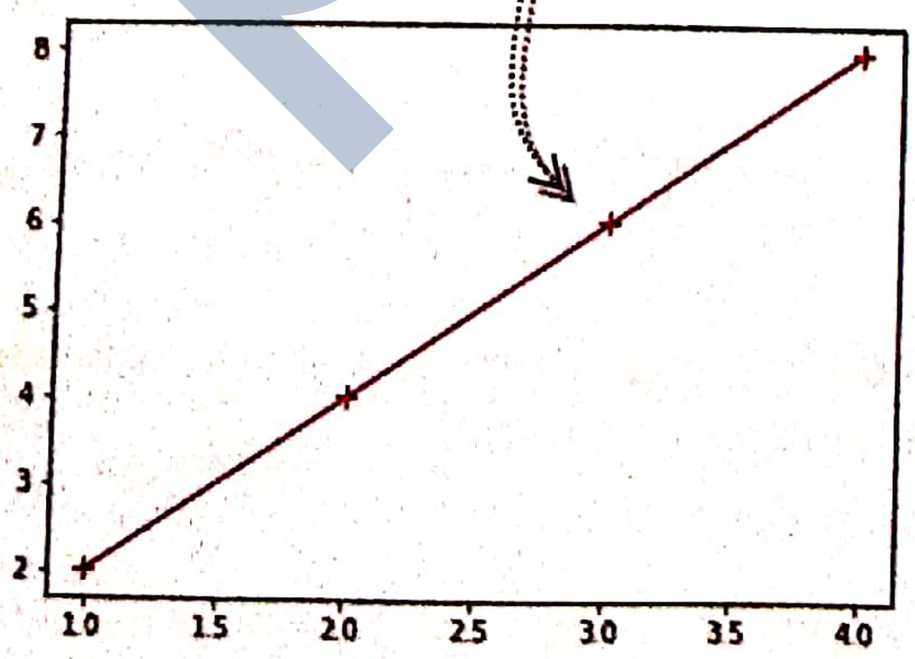


Line color is black (color 'k')
Marker type = small diamond ('d')
Marker size = 5 points
Marker color = 'red'

Note you can combine the marker type with color code, *e.g.*, 'r+' when given for ***line color*** marks the *color* as **'red'** and ***markertype*** as plus('+'). Consider two following examples combining these :

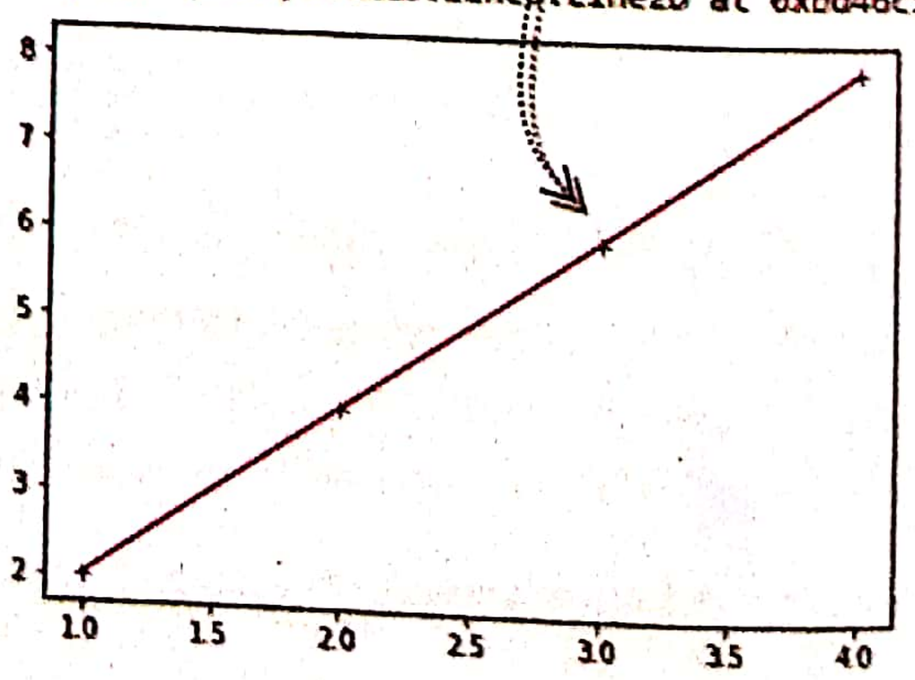
Line color and *marker style* combined so marker takes same color as line

```
In [75]: plt.plot(p, q, 'r+', linestyle = 'solid')
Out[75]: [<matplotlib.lines.Line2D at 0xbdfc470>]
```



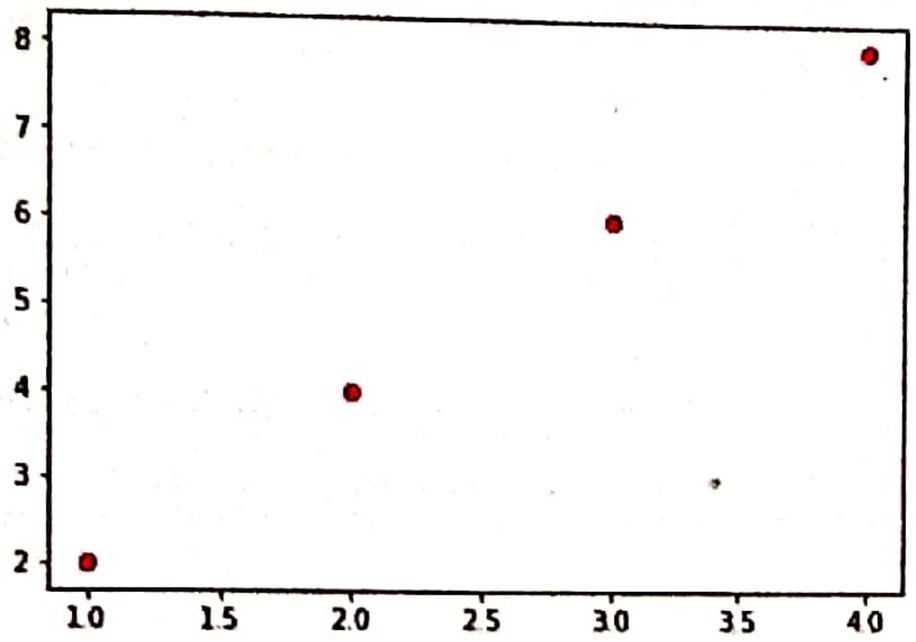
Here *marker color* separately specified

```
In [76]: plt.plot(p, q, 'r+', linestyle = 'solid',
    ...: markeredgecolor='b')
Out[76]: [<matplotlib.lines.Line2D at 0xbd46c50>]
```



See, when you do not specify **markeredgecolor** separately in **plot( )**, the marker takes the same color as the line. Also, if you do not specify the **linestyle** separately along with *linecolor- and-marketstyle-combination-string* (*e.g.*, 'r+' above), Python will only plot the markers and not the line, (see below). To get the line, specify **linestyle** argument also as shown above.

```
In [3]: plt.plot(p,q, 'ro')
Out[3]: [<matplotlib.lines.Line2D at 0x8834770>]
```



if you do not specify the **linestyle** argument separately along with linecolor-&-marketstyle-string (e.g., 'r+' or 'bo' etc.), Python will only plot the markers and not the line

So full useful syntax of **plot( )** can be summarized as :

```
plot( <data1> [,<data2>] [, … <colorcode and marker type>] [, <linewidth>]
      [, <linestyle>] [,<marker>][, <markersize>] [,<markeredgecolor>] )
```

where,

- ***data1, data2*** are sequences of values to be plotted on x-axis and y-axis respectively
- Rest of the arguments affect the look and format of the line chart as given below :
  - **color code with markertype** color and marker symbol for the line chart
  - **linewidth** width of the line in points (a float value)
  - **linestyle** can be ['solid' | 'dashed', 'dashdot', 'dotted' or take markertype string]
  - **marker** a symbol denoting valid marker style such as '–', 'o', 'x', '+', 'd', and others etc.
  - **markersize** size of marker in points (a float value)
  - **markeredgecolor** color for markers; should be a valid color identified by PyPlot

* *please note that this is still not the complete and full syntax of plot( ).*
Complete syntax coverage of plot( ) is beyond the scope of the book.

NOTE

If you are working on a Jupyter Notebook, you can make the plots appear inline in the notebook by using following magic function :

```
%matplotlib inline
```

If you don't use above function, the plots will appear in a pop up window as with other methods.

Example 8.1 *Create an array in the range 1 to 20 with values 1.25 apart. Another array contains the log values of the elements in first array.*

(*a*) *Simply plot the two arrays first vs second in a line chart.*

(*b*) *In the plot of first vs second array, specify the x-axis (containing first array's values) title as* ***'Random Values'*** *and y-axis title as* ***'Logarithm Values'****.*

(c) *Create a third array that stores the COS values of first array and then plot both the second and third arrays vs first array. The Cos values should be plotted with a* ***dashdotted line.***

(d) *Change the marker type as a circle with blue color in second array.*

(e) *Only mark the data points as this : second array data points as* ***blue small diamonds,*** *third array data points as* ***black circles.***

Note : Show commands and their results.

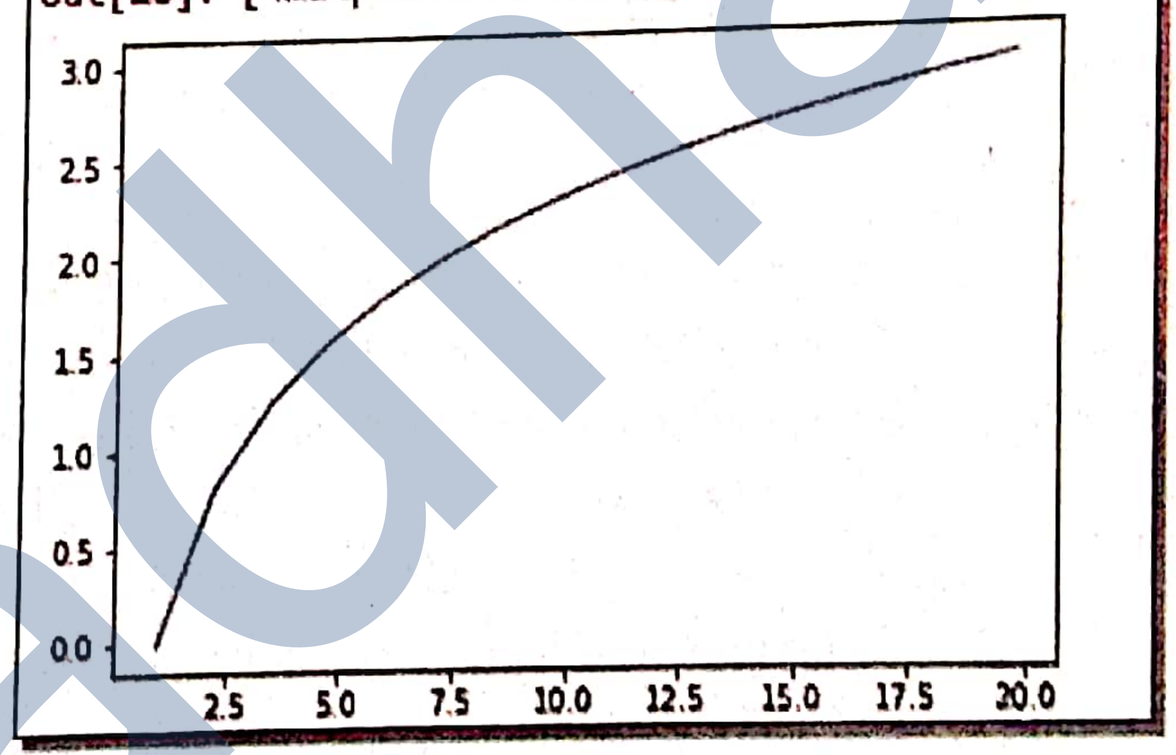
Solution.

```
import numpy as np
import matplotlib.pyplot as plt
a = np.arange(1, 20, 1.25)
b = np.log(a)
```

```
In [16]: print(a)
[ 1.    2.25  3.5   4.75  6.    7.25  8.5   9.75 11.   12.25 13.5  14.75
 16.   17.25 18.5  19.75]

In [17]: print(b)
[0.         0.81093022 1.25276297 1.55814462 1.79175947 1.98100147
 2.14006616 2.27726729 2.39789527 2.50552594 2.60268969 2.69124308
 2.77258872 2.84781214 2.91777073 2.98315349]
```

(a) `plt.plot(a, b)`

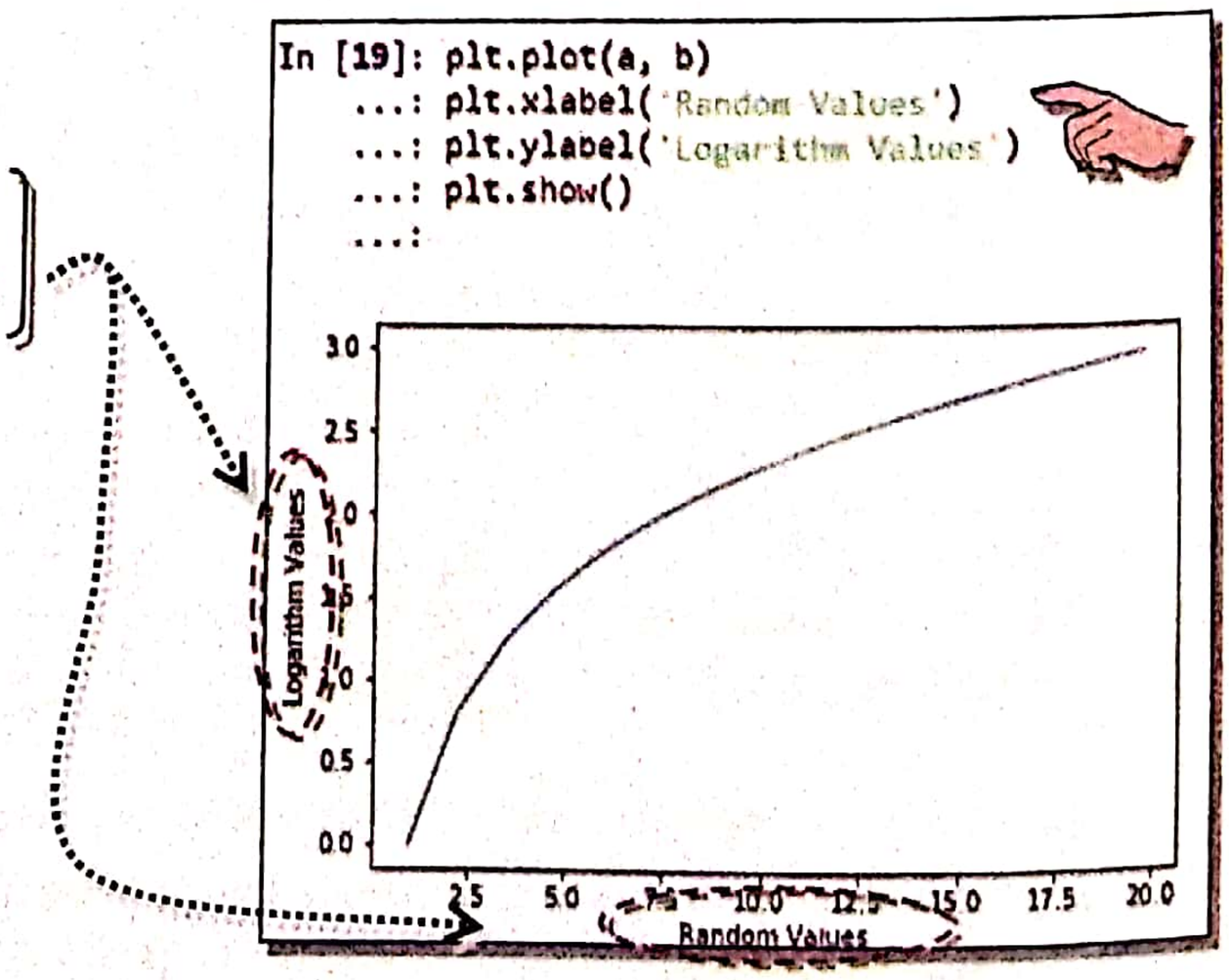
```
In [18]: plt.plot(a, b)
Out[18]: [<matplotlib.lines.Line2D at 0xca0e930>]
```



(b)

```
plt.plot(a, b)
plt.xlabel('Random Values')
plt.ylabel('Logarithm Values')
plt.show()
```

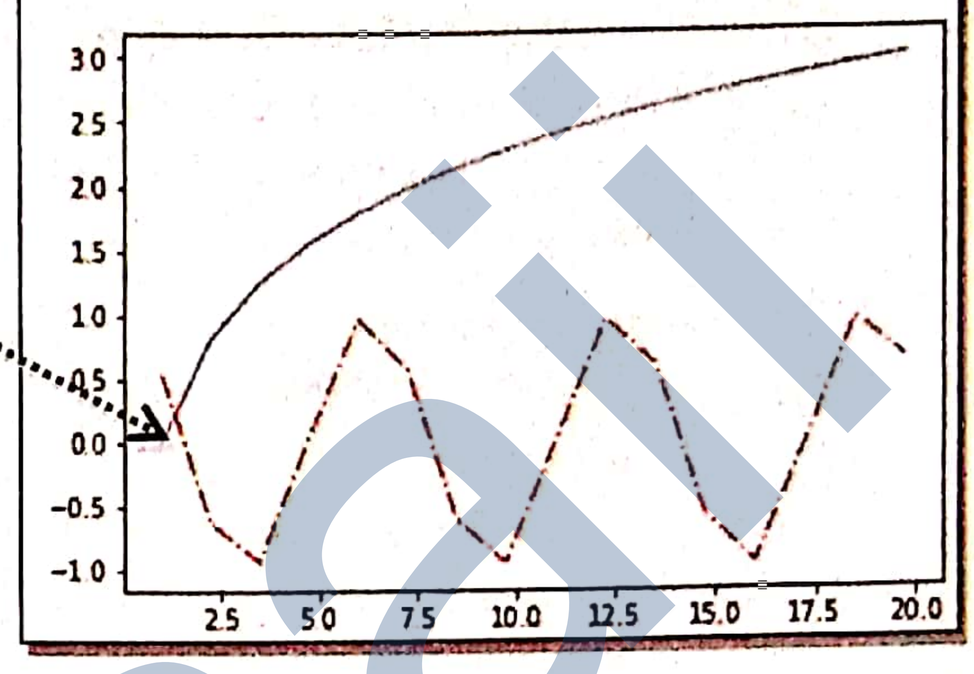
```
In [19]: plt.plot(a, b)
    ...: plt.xlabel('Random Values')
    ...: plt.ylabel('Logarithm Values')
    ...: plt.show()
    ...:
```



(c)

```
c = np.cos(a)
plt.plot(a, b)
plt.plot(a, c, linestyle = 'dashdot')
plt.show()
```

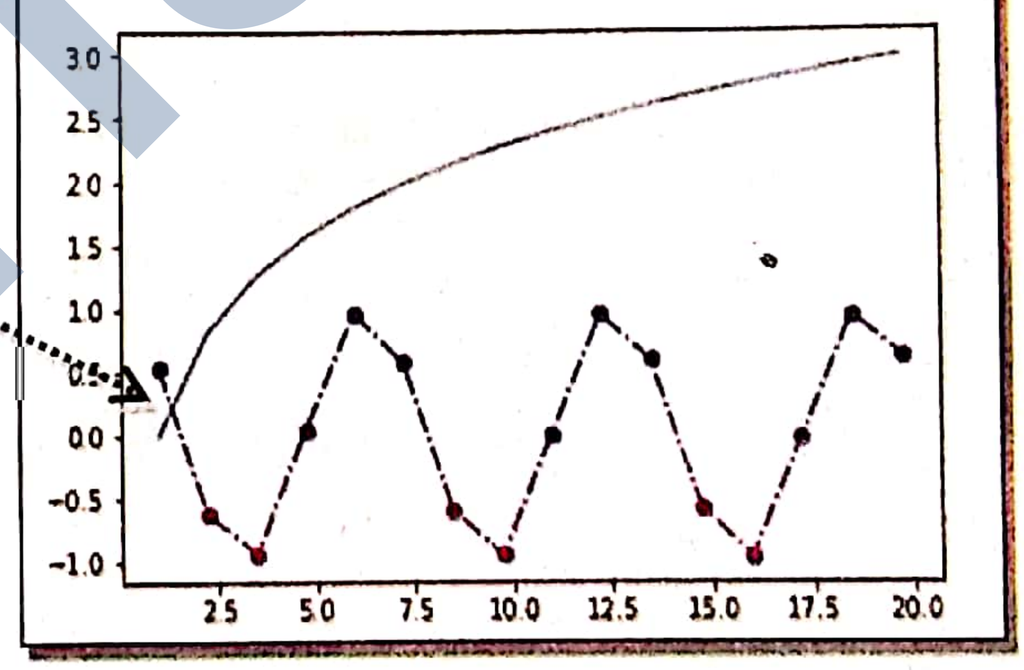
```
In [21]: c = np.cos(a)
    ...: plt.plot(a, b)
    ...: plt.plot(a, c, linestyle='dashdot')
    ...: plt.show()
    ...:
```



(d)

```
c = np.cos(a)
plt.plot(a, b)
plt.plot(a, c, 'bo', linestyle = 'dashdot')
plt.show()
```

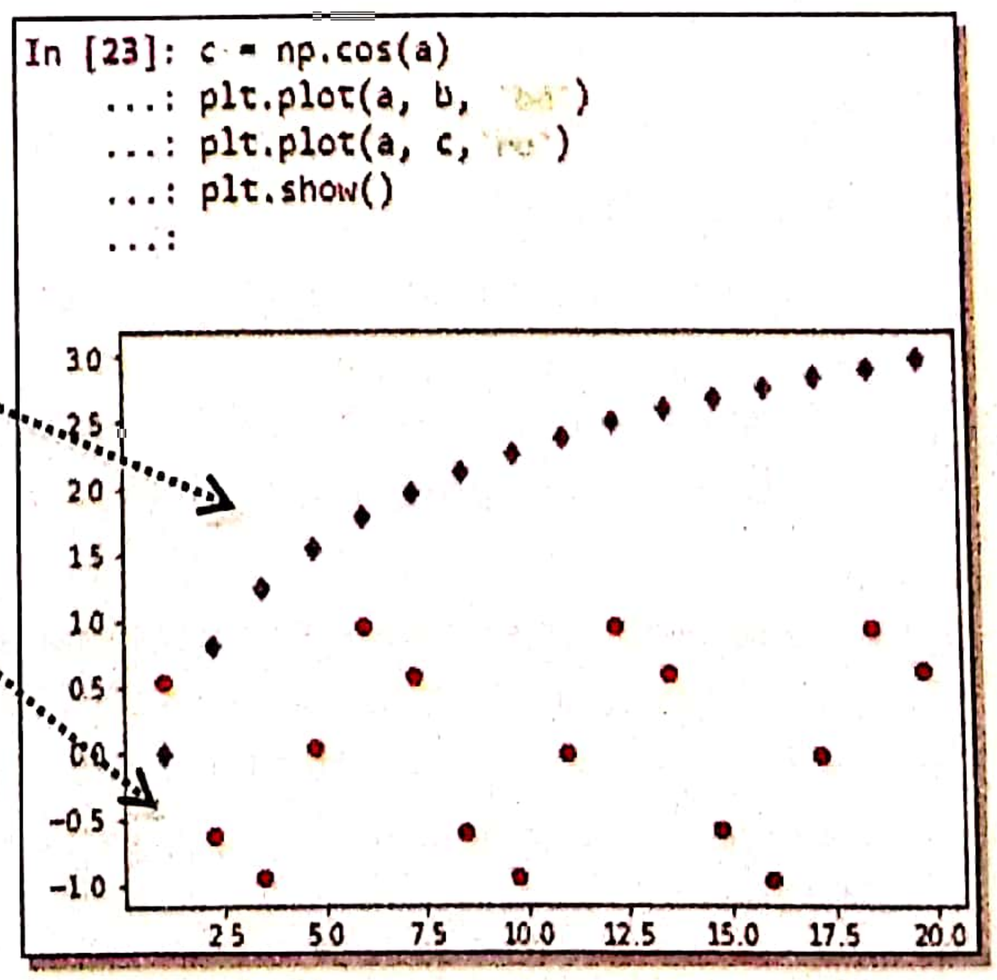
```
In [22]: c = np.cos(a)
    ...: plt.plot(a, b)
    ...: plt.plot(a, c, 'bo', linestyle='dashdot')
    ...: plt.show()
    ...:
```



(e)

```
c = np.cos(a)
plt.plot(a, b, 'bd')
plt.plot(a, c, 'ro')
plt.show()
```

```
In [23]: c = np.cos(a)
    ...: plt.plot(a, b, 'bd')
    ...: plt.plot(a, c, 'ro')
    ...: plt.show()
    ...:
```



8.3.2 Bar Chart

BAR GRAPH / CHART

A Bar Graph / Chart is a graphical display of data using bars of different heights.

A ***Bar Graph*** or a ***Bar Chart*** is a graphical display of data using bars of different heights. A bar chart can be drawn *vertically* or *horizontally* using rectangles or bars of different heights /widths PyPlot offers **bar( )** function to create a bar chart where you can specify the sequences for *x*-axis and corresponding sequence to be plotted on *y*-axis. Each *y* value is plotted as bar on corresponding *x*-value on *x*-axis.

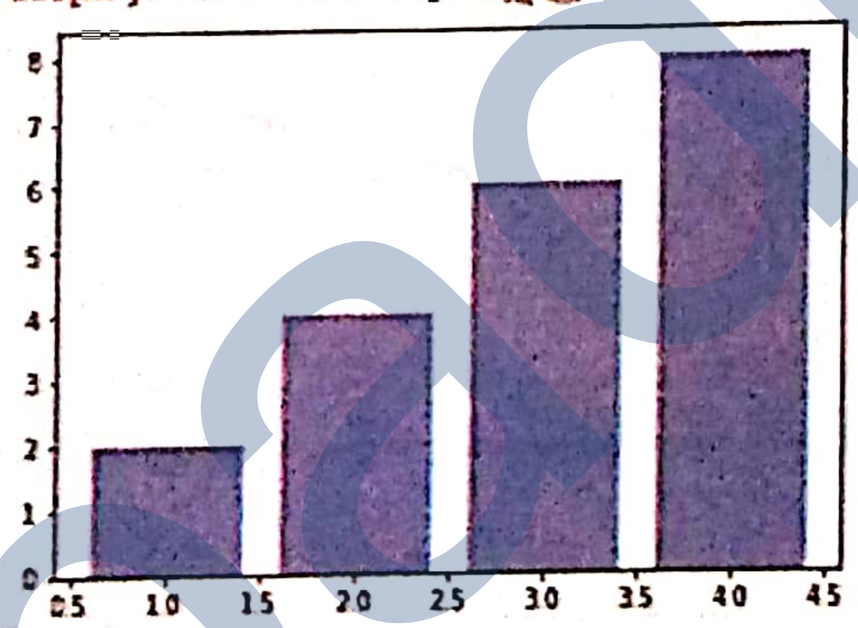
But before creating any bar chart, like you did in line charts, make sure to *import* ***matplotlib.pyplot*** *and also* ***numpy****, if you are using any numpy function. Likewise, if you want that multiple commands affect a common bar chart, then either store all the related statements in a Python script(.py file) with last statement being <matplotlib.pyplot>.show( ), or on iPython console, group related commands by pressing CTRL+ENTER; press ENTER only after the last command.*

Now consider the following code. If we have *three* sequences **a**, **b**, and **c** as :

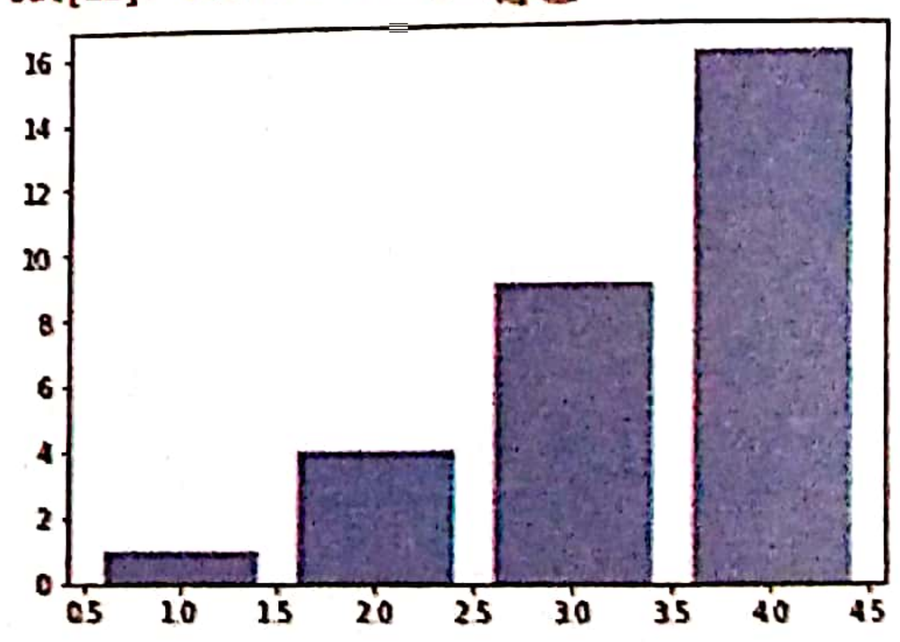
```
a, b, c = [1, 2, 3, 4], [2, 4, 6, 8], [1, 4, 9, 16]
```

Then commands **matplotlib.pyplot.bar(a.b)** and **matplotlib.pyplot.bar(a,c)** will produce result as :

```
In [10]: plt.bar(a,b)
Out[10]: <Container object of 4 artists>
```




```
In [11]: plt.bar(a,c)
Out[11]: <Container object of 4 artists>
```



Notice, first sequence given in the **bar( )** forms the *x-axis* and the second sequence values are plotted on y-axis.

As earlier, if you want to specify *x-axis label* and *y-axis label*, then you need to give commands :

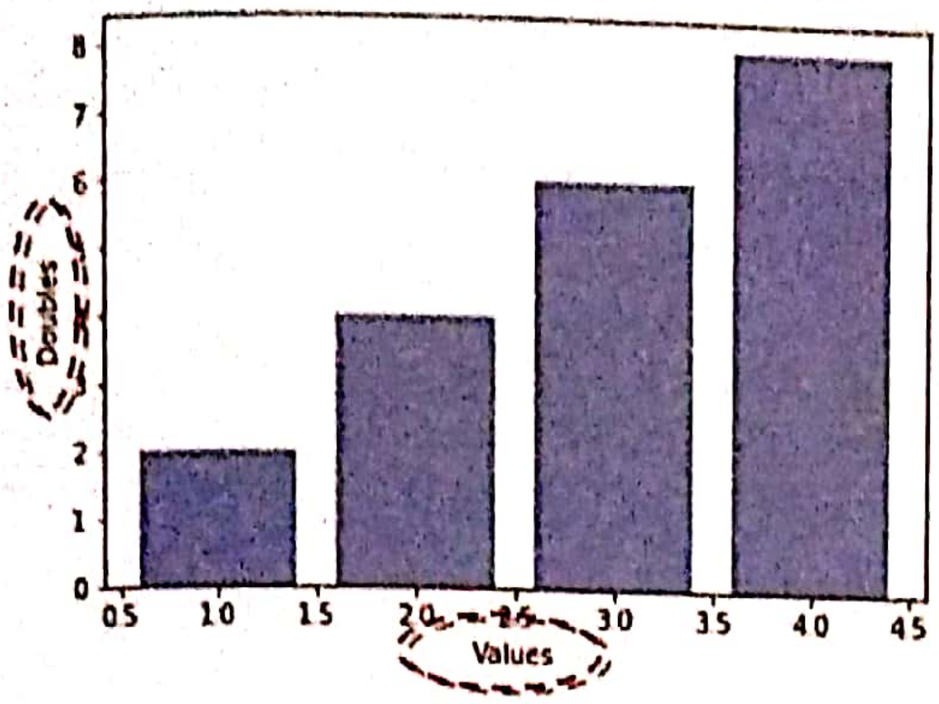
```
matplotlib.pyplot.xlabel(<label string>)
matplotlib.pyplot.ulabel(<label string>)
```

Also, you need to group the commands either in a script file (*.py* file) or by pressing CTRL+ENTER on the prompt for all commands except for the last command of the group (ENTER for the last command), as you can see yourself in the figures below.

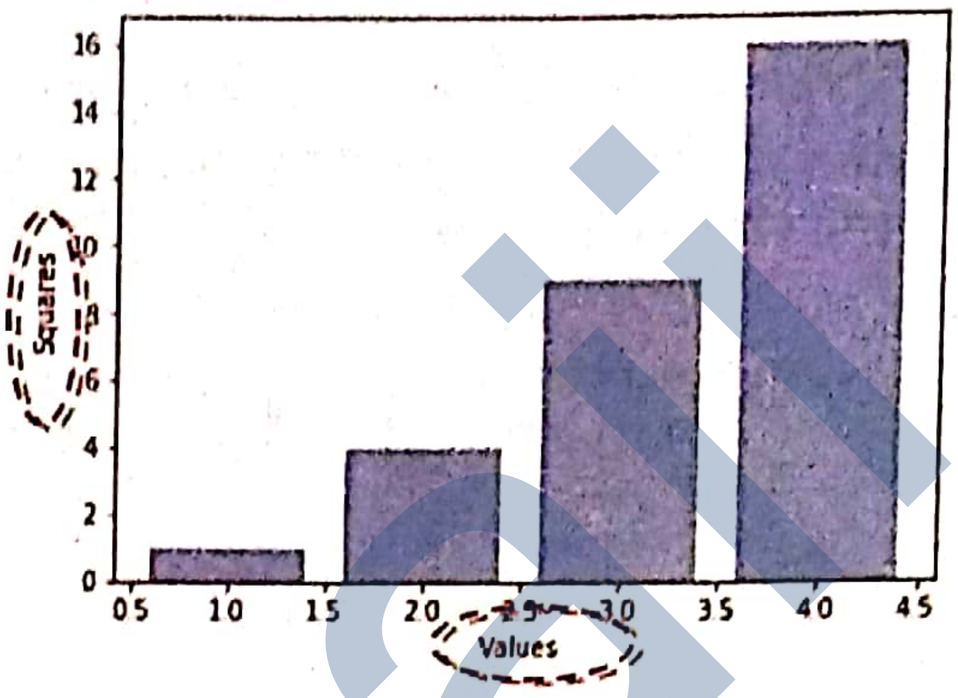
NOTE

The simple bar plot is best used when there is just one level of grouping to your variable.

```
In [12]: plt.bar(a,b)
    ...: plt.xlabel('Values')
    ...: plt.ylabel('Doubles')
    ...:
Out[12]: Text(0,0.5,'Doubles')
```




```
In [13]: plt.bar(a,c)
    ...: plt.xlabel('Values')
    ...: plt.ylabel('Squares')
    ...:
Out[13]: Text(0,0.5,'Squares')
```



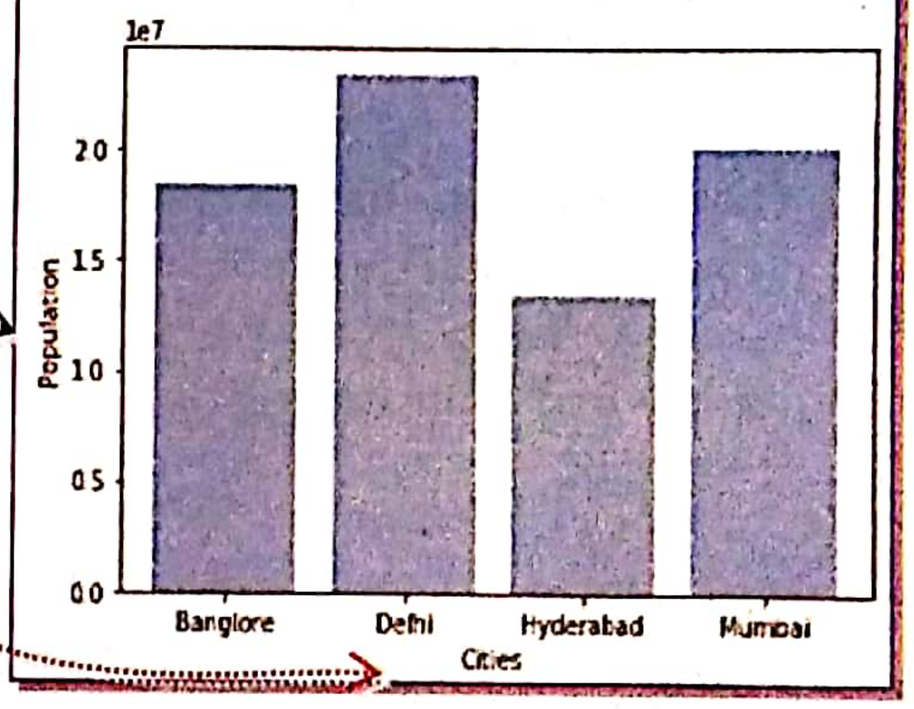
NOTE

The first sequence given in the **bar( )** forms the *x-axis* and the second sequence values are plotted on *y-axis*.

Consider another example :

```
Cities = ['Delhi', 'Mumbai', 'Banglore', 'Hyderabad']
Population = [23456123, 20083104, 18456123, 13411093]
plt.bar(Cities, Population)
plt.xlabel('Cities')
plt.ylabel('Population')
plt.show()
```

```
In [9]: plt.bar(Cities, Population)
   ...: plt.xlabel('Cities')
   ...: plt.ylabel('Population')
   ...: plt.show()
   ...:
```



NOTE

The order of bars plotted may be different from the order in actual data sequence.

Note that the order of bars plotted is different from the order in actual data sequence *e.g.*, the **Cities** sequence being plotted contains values in the order ("Delhi", "Mumbai", "Bangalore", "Hyderabad") but when plotted their order is different, *i.e.*, "Bangalore", "Delhi", "Hyderabad" and "Mumbai" – the sorted order.

8.3.2A Changing Widths of the Bars in a Bar Chart

By default, bar chart draws bars with equal widths and having a default width of 0.8 units on a bar chart. That is, all bars have the width same as the *default width*. But you can always change the widths of the bars.

- You can specify a different width (other than the default width) for all the bars of a bar chart
- You can also specify different widths for different bars of a bar chart

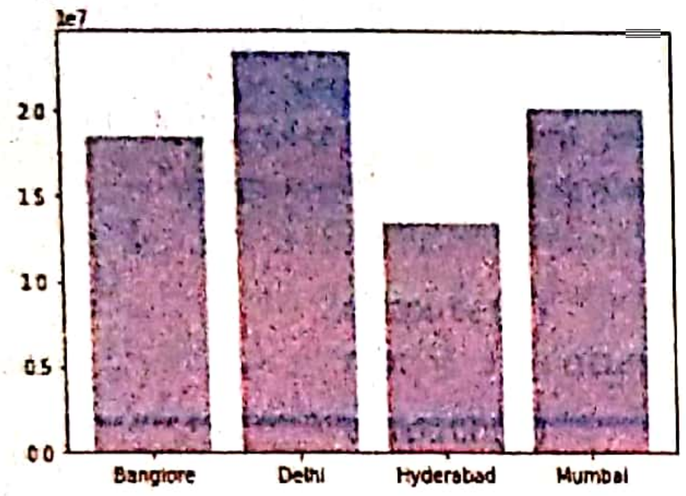
(*i*) **To specify common width (other than the default width) for all bars,** you can specify *width* **argument** having a **scalar float value** in the **bar( )** function, *i.e.*, as :

```
<matplotlib.pyplot>.bar(<x-sequence>, <y-sequence>, width = <float value>)
```

Consider the following three bar charts where the first chart has the bars with default *widths*, second chart with a *width as per expression 1/2* (*i.e.*, 0.5 units) and the third chart has *width specified as 0.3 units.*

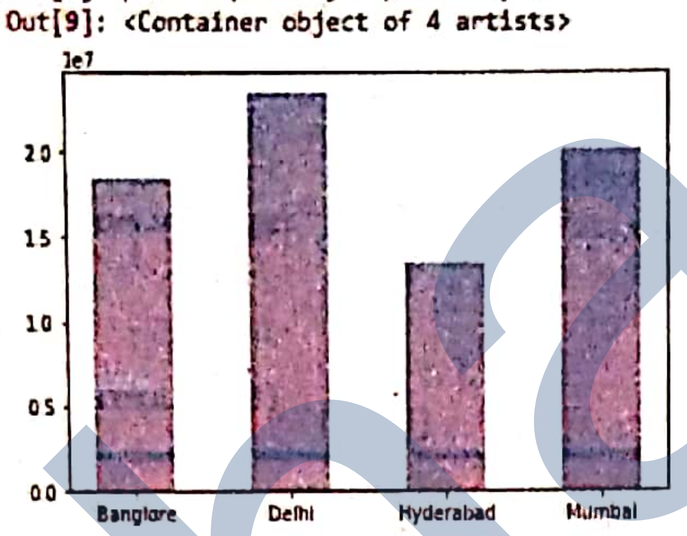
Default width for all bars
plt.bar(Cities, Population)

```
In [8]: plt.bar(Cities, Population)
   ...:
Out[8]: <Container object of 4 artists>
```



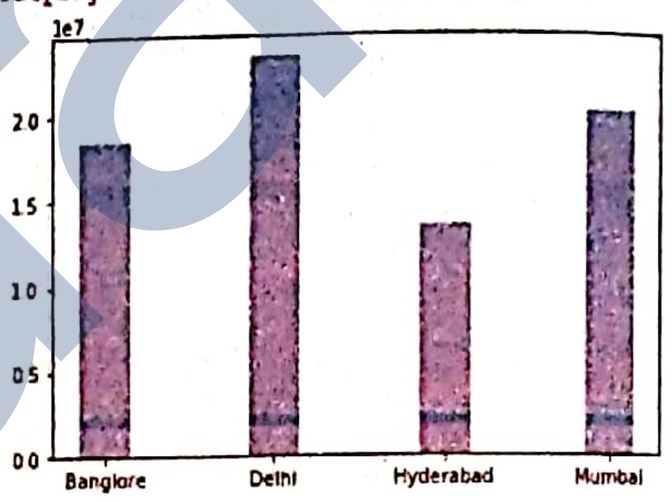
Width as 0.5 units for all bars
plt.bar(Cities, Population, **width = 1/2**)

```
In [9]: plt.bar(Cities, Population, width = 1/2)
Out[9]: <Container object of 4 artists>
```



Width as 0.3 units for all bars
plt.bar(Cities, Population, **width = 0.3**)

```
In [10]: plt.bar(Cities, Population, width = 0.3)
Out[10]: <Container object of 4 artists>
```



So, when you specify a scalar value *i.e.*, a single value for **width** argument, then that width is applied to all the bars of the bar chart.

NOTE

If you specify a scalar value (a single value) for **width** argument, then that width is applied to all the bars of the bar chart.

(*ii*) **To specify different widths for different bars of a bar chart,** you can specify *width* **argument** having a sequence (such as lists or tuples) containing widths for each of the bars, in the **bar( )** function, *i.e.*, as :

```
<matplotlib.pyplot>.bar(<x-sequence>, <y-sequence>, width = <width values sequence>)
```

The widths given in the sequence are applied from left to right, *i.e.*, the first value of the sequence specifies the width of first value of data sequence, second value specifies the width for the second value of data sequence, and so on. The width values are *float* values. Please note **that the width sequence must have widths for all the bars** (*i.e.*, its length must match the length of data sequences being plotted) otherwise Python will report an error, the [*ValueError: shape mismatch error*].

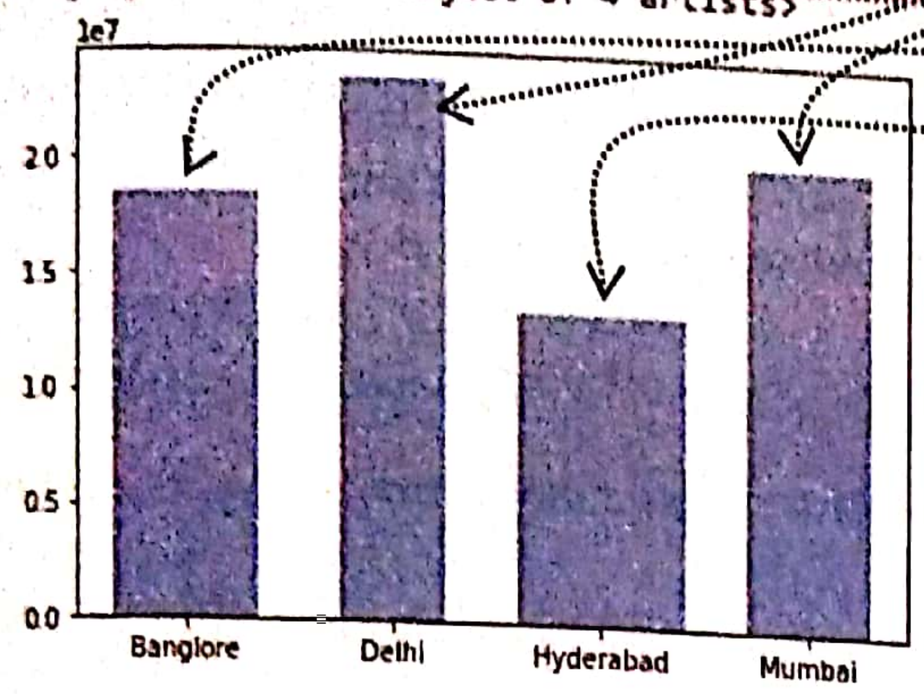
Consider the following bar chart :

```
plt.bar(Cities, Population, width =[0.5,.6, .7, .8])
```

So as per above **bar( )**, the widths 0.5, .6, .7, .8 are applicable to **Cities** values in order *i.e.*, "**Delhi**"'s bar will be 0.5 units wide; "**Mumbai**"'s bar will be 0.6 units wide ; "**Bangalore**"'s bar will be 0.7 units wide ; and "**Hyderabad**"'s bar will be 0.8 units wide.

The output produced by above command will be as shown below :

```
In [12]: plt.bar(Cities, Population, width =[0.5,.6, .7, .8])
Out[12]: <Container object of 4 artists>
```



See, the widths are applied to values of data sequence in *left to right order* but the bars appear in sorted order in the bar chart

The width values' sequence in a bar( ) must have widths for all the bars, *i.e.*, its length must match the length of data sequences being plotted, otherwise Python will report an error.

8.3.2B Changing Colors of the Bars in a Bar Chart

By default, a bar chart draws bars with same default color. But you can always change the color of the bars.

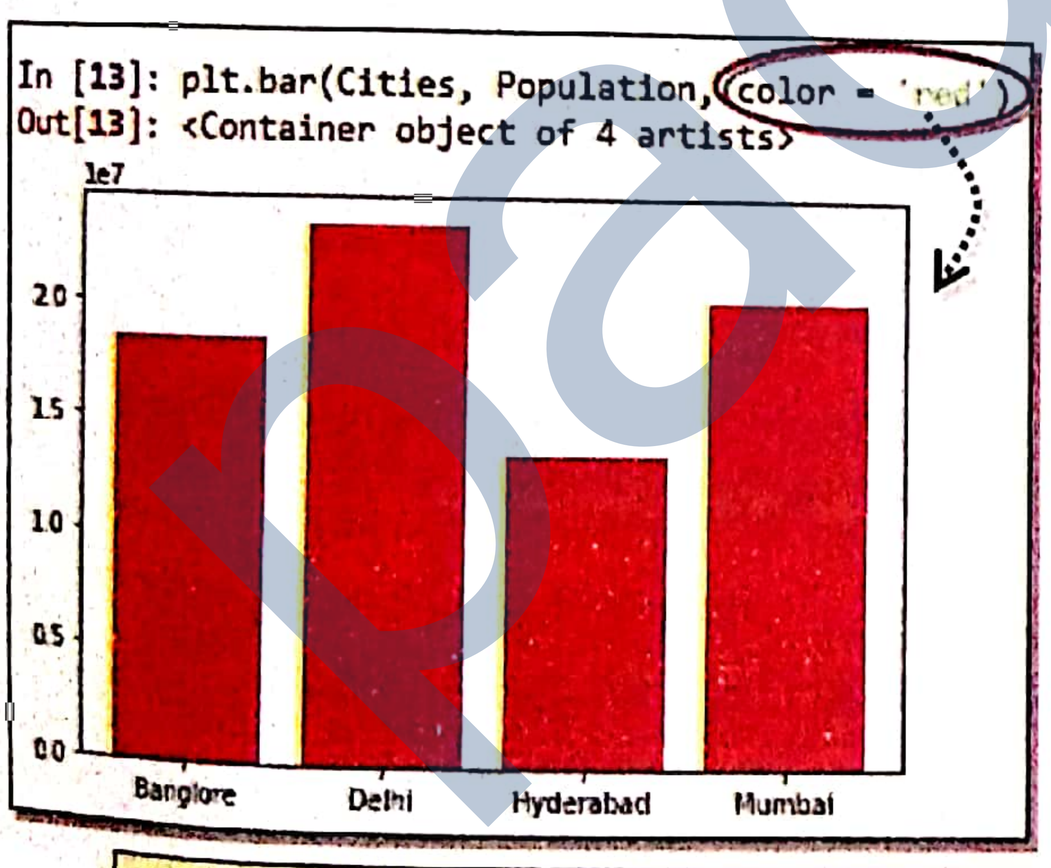
- You can specify a different color for all the bars of a bar chart.
- You can also specify different colors for different bars of a bar chart.

(*i*) **To specify common color (other than the default color) for all bars**, you can specify ***color* argument** having a **valid color code/name** in the **bar( )** function, *i.e.*, as :

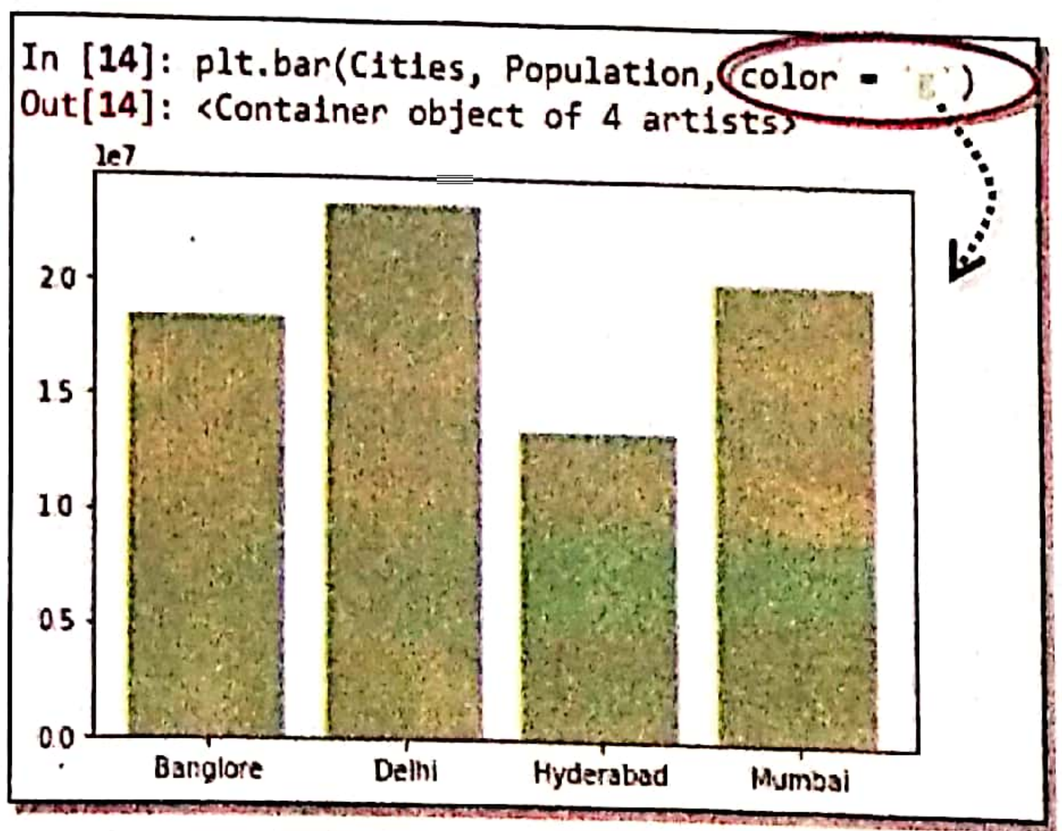
```
<matplotlib.pyplot>.bar(<x-sequence>, <y-sequence>, color = <color code/name>)
```

The *color* given with **color** argument will be applied to all the bars, *e.g.*, consider this :

```
In [13]: plt.bar(Cities, Population, color = 'red')
Out[13]: <Container object of 4 artists>
```




```
In [14]: plt.bar(Cities, Population, color = 'g')
Out[14]: <Container object of 4 artists>
```



When you specify single color name or single color code with colr argument of the bar() function, the specified color is applied to all the bars of the bar chart, i.e., all bars of the bar chart have the same common color.

(*ii*) **To specify different colors for different bars of a bar chart**, you can specify ***color*** argument having a sequence (such as lists or tuples) containing colors for each of the bars, in the **bar( )** function, *i.e.*, as :

```
<matplotlib.pyplot>.bar(<x-sequence>, <y-sequence>, color = <color names/codes sequence>)
```

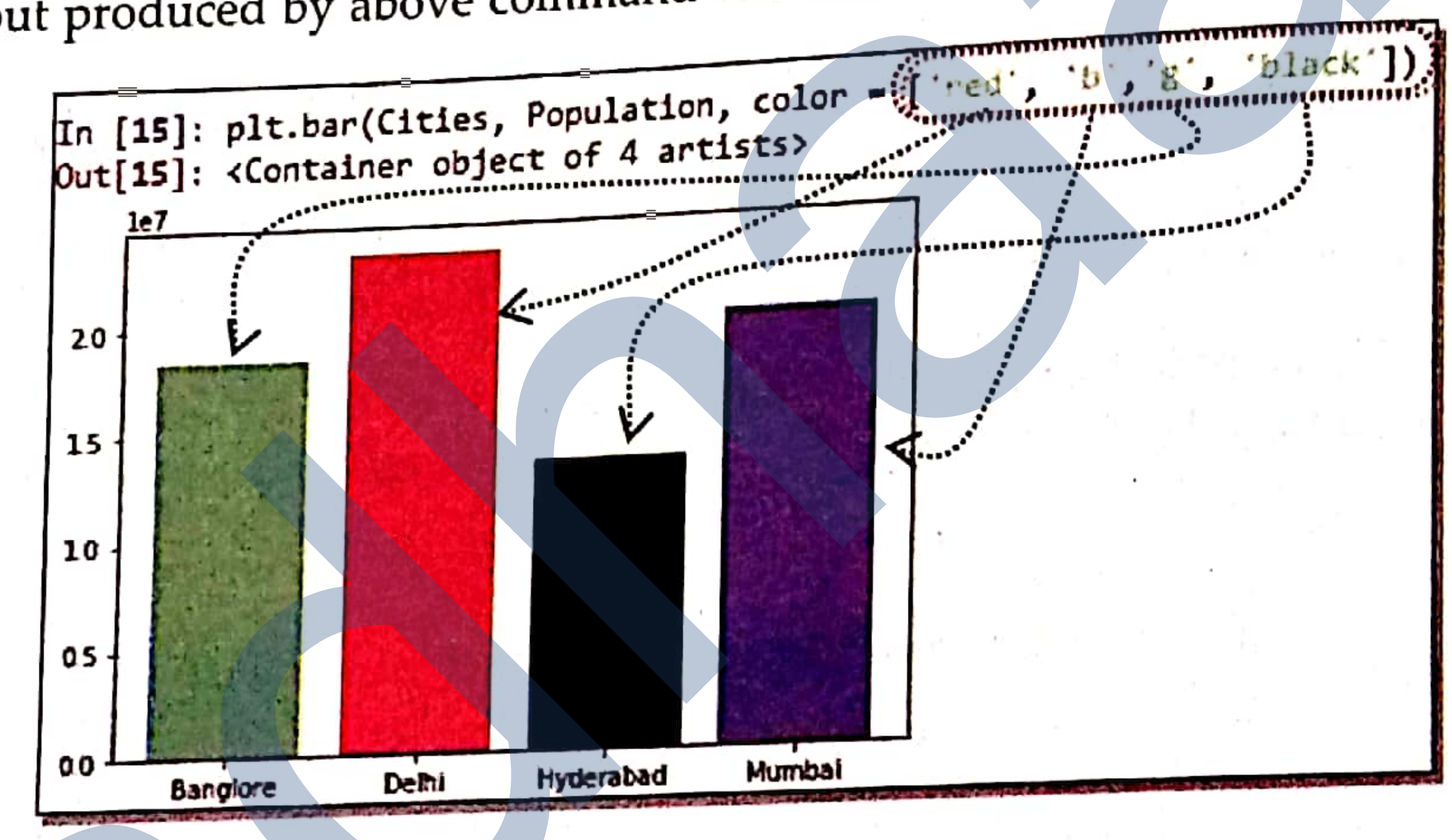
The colors given in the sequence are applied from left to right, *i.e.*, the first value of the sequence specifies the color of first value of data sequence, second value specifies the color for the second value of data sequence, and so on. The color values are *valid color codes or names*. Please note that **the color sequence must have colors for all the bars** (*i.e.*, its length must match the length of data sequences being plotted) otherwise Python will report an error, the [***ValueError: shape mismatch error***].

Consider the following bar chart :

```
plt.bar(Cities, Population, color = ['red', 'b', 'g', 'black'])
```

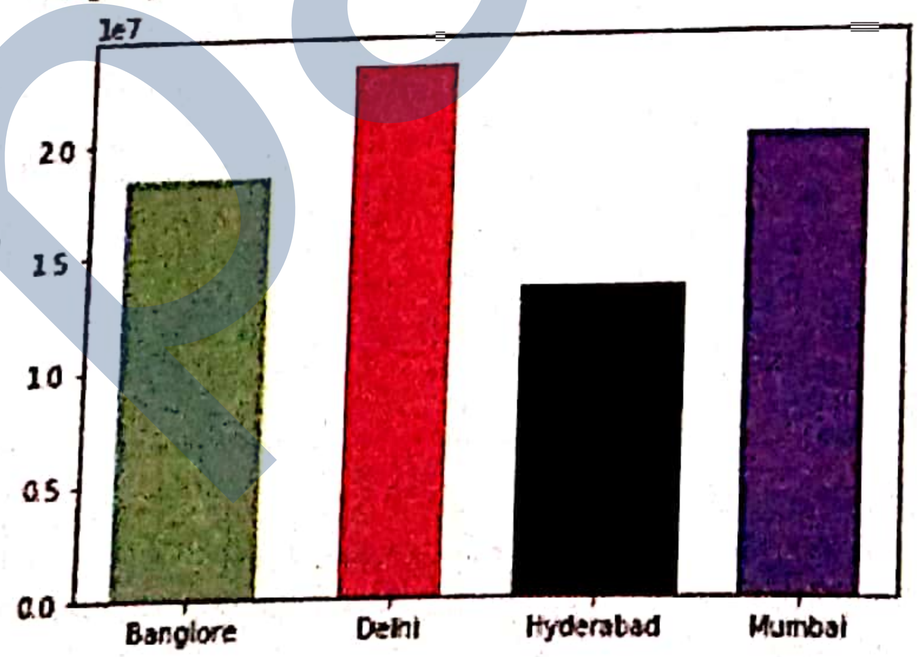
So as per above **bar( )**, the colors *'red', 'b', 'g', 'black'* are applicable to **Cities** values in order *i.e.*, "**Delhi**"'s bar will have 'red' color; "**Mumbai**"'s bar will have blue color (color code'b'); "**Bangalore**"'s bar will have 'green' color (color code 'g') ; and "**Hyderabad**"'s bar will have 'black' color.

The output produced by above command will be as shown below :



Both *width* and *color* arguments are optional and you can combine them both also, *e.g.*,

```
In [16]: plt.bar(Cities, Population, width =[0.5,.6, .7, .8], color = ['red', 'b','g', 'black'])
Out[16]: <Container object of 4 artists>
```



8.3.2C Creating Multiple Bars chart

Often in real life, you may need to plot multiple data ranges on the same bar chart creating multiple bars. As such PyPlot does not provide a specific function for this, BUT you can always create one exploiting the ***width*** and ***color*** arguments of **bar( )** that you learnt above.

Let's see how.

(*i*) **Decide number of X points.** Firstly, you need to determine how many X points you will need. For this, calculate the number of entries in the ranges being plotted. Say, you are plotting two sequences **A** and **B**, so length of sequences **A** or **B** being plotted will determine the number of X points in our case. You can create a sequence with this length of A or B using ***range( )*** or **numpy.arange( )** functions.
Do note that both the sequences A and B must have similar shape *i.e.*, same length (number of elements).

(*ii*) **Decide thickness of each bar and accordingly adjust X points on X-axis.** So, if you want to plot two bars per X point, then carefully think of the thickness of each bar. Say you want the thickness of each bar as .3 , then for first range keep it as X and for the second data range, make the X point as X + 0.3. (see below)

(*iii*) **Give different color to different data ranges** using color argument of bar( ) function.

(*iv*) **The *width* argument remains the same for all ranges being plotted.**

(*v*) **Plot using bar( ) for each range separately,** *i.e.*, the number of bars decide the number of **bar( )** functions you will be using to plot different bars on same plot.

Following example will make it clear.

Say we want to plot ranges $A = [2, 4, 6, 8]$ and $B = [2.8, 3.5, 6.5, 7.7]$

(*i*) Deciding X points and thickness. Say, we want the thickness of each bar as 0.35, then **for the first range, X point will be X,** and **for the second range,** the X will shift by first bar's thickness, *i.e.*, X + 0.35. (If there were three bars, then X points would have been X, X + 0.35 *i.e.*, X + thick of one bar on its left, and X + 0.70 , *i.e.*, X + thickness of two bars on its left.)

(*ii*) **Deciding colors.** Say we want *red* color for the first range and *blue* color for the second range.

(*iii*) The **width** argument will take value as 0.35 in this case.

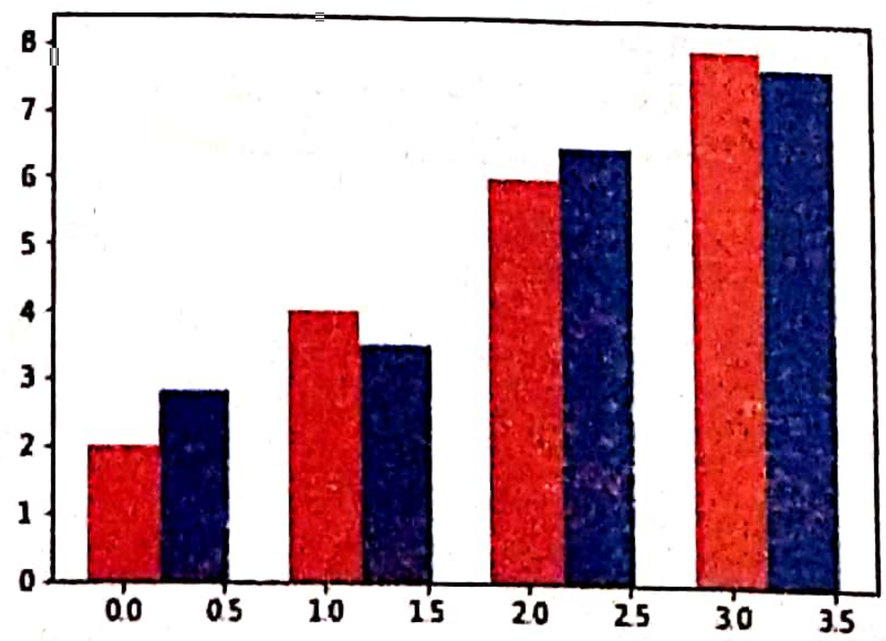
(*iv*) Plot using multiple bar( ) functions.

As there are two ranges to be plotted, we shall need to use two bar( ) functions one for plotting X *vs.* A and the second one for plotting X + width *vs.* B, as depicted below :

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

In [2]: A = [2, 4, 6, 8]
   ...: B = [2.8, 3.5, 6.5, 7.7]
   ...:

In [3]: X = np.arange( len(A) )

In [4]: plt.bar(X, A, color='red', width = 0.35)
   ...: plt.bar(X+0.35, B, color= 'blue', width = 0.35)
   ...: plt.show()
   ...:
```

Sequence created for X points using lengths of sequences A or B being plotted

See, X in first bar() is X and in second bar(), it is X +0.35. Also, first bar() plots X vs A and second bar() plots X+width vs B



Following example also creates a multi-bar chart plotting three different data ranges.

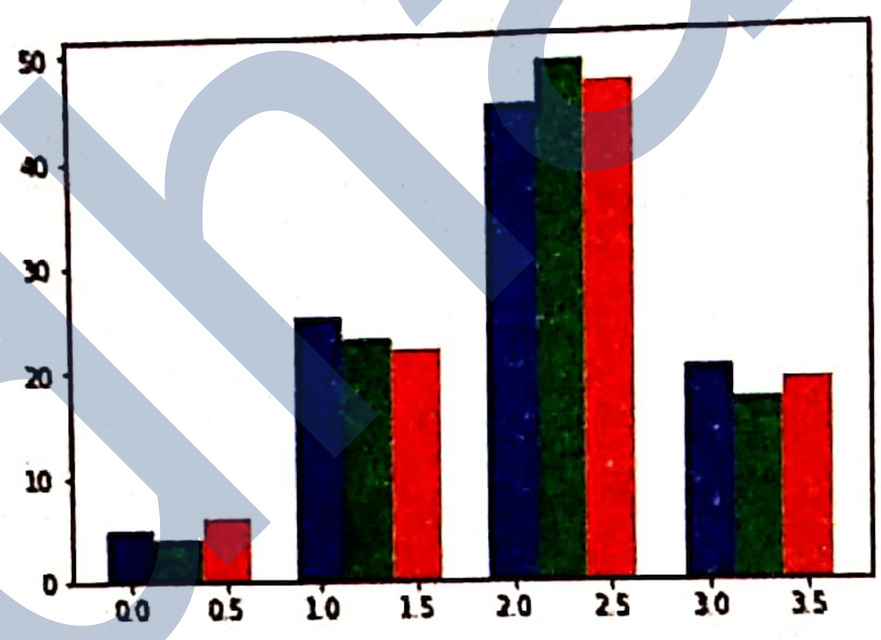
Example 8.2 *Val is a list having three lists inside it. It contains summarized data of three different trials conducted by company A. Create a bar chart that plots these three sublists of Val in a single chart. Keep the width of each bar as 0.25.*

Solution.

```
import numpy as np
import matplotlib.pyplot as plt

Val = [[5., 25., 45., 20.], [4., 23., 49., 17.], [6., 22., 47., 19.]]

X = np.arange(4)
plt.bar(X + 0.00, Val[0], color = 'b', width = 0.25)
plt.bar(X + 0.25, Val[1], color = 'g', width = 0.25)
plt.bar(X + 0.50, Val[2], color = 'r', width = 0.25)

plt.show()
```

See, the X range for the third bar(), is deviated with X+ widths of two bars on the left of it. i.e., X+0.50 as width of one bar is 0.25

The output produced by above code is :

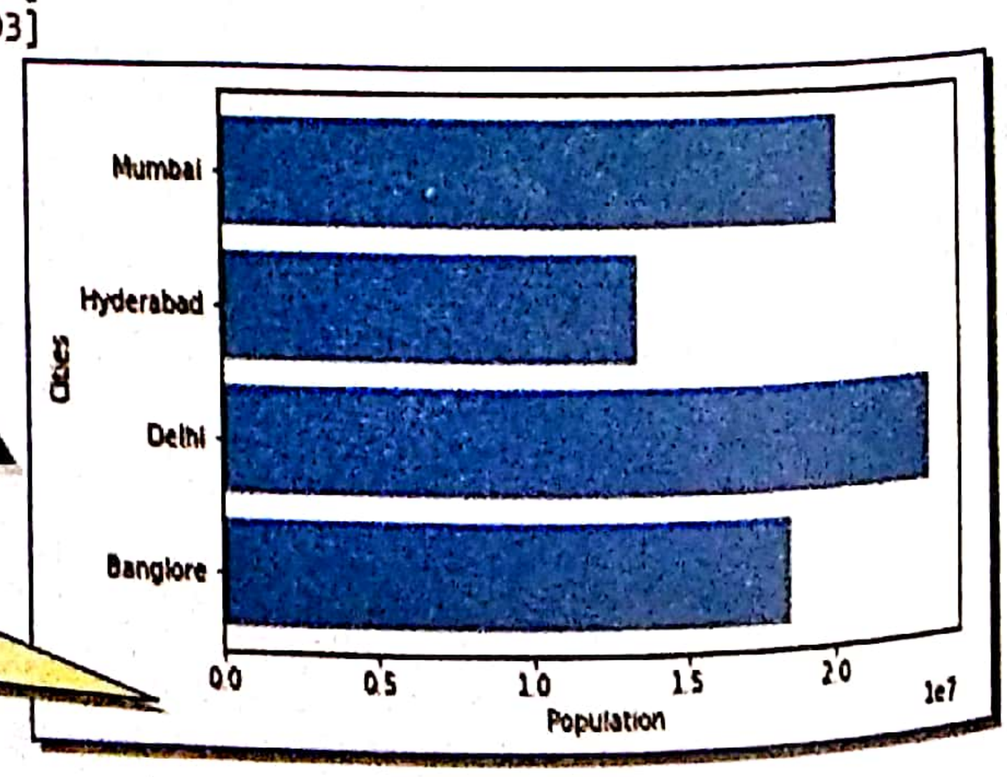


8.3.2D Creating a Horizontal Bar Chart

To create a horizontal bar chart, you need to use **barh( )** function (bar horizontal), in place of bar( ). Also, you need to give x and y axis labels carefully – the label that you gave to x axis in bar( ), will become y-axis' label in barh( ) and vice versa.

For instance, consider the following code where we have replaced bar( ) with barh( ) in an earlier code :

```
In [2]: Cities =['Delhi', 'Mumbai', 'Banglore', 'Hyderabad']
   ...: Population= [23456123, 20083104, 18456123, 13411093]
   ...: plt.barh(Cities, Population)
   ...: plt.ylabel('Cities')
   ...: plt.xlabel('Population')
   ...: plt.show()
```



Also, notice that the labels of x and y axis have been swapped for **barh()** when compared to **bar()**

8.3.3 The Pie Chart

Recall that a pie chart (or a circle chart) is a circular statistical graphic, which is divided into slices to illustrate numerical proportion. Typically, a Pie Chart is used to show parts to the whole, and often a % share. The PyPlot interface offers **pie( )** function for creating a pie chart.

PIE CHART

The **pie chart** is a type of graph in which a circle is divided into sectors that each represent a proportion of the whole.

Before we proceed with pie( ) function's examples, it is important to know two things :

(*i*) The pie( ) function, **plots a single data range only.** It will calculate the share of individual elements of the data range being plotted *vs.* the whole of the data range.

(*ii*) The default shape of a pie chart is oval but you can always change to circle by using **axis( )** of pyplot, sending "equal" as argument to it. That is, issue following command before you use **pie( )** function :

```
matplotlib.pyplot.axis("equal")
```

*In place of **matplotlib.pyplot**, you may also use its alias name, the name you have given to it while importing*

Now consider following example code to understand working of pie( ).

Your school houses have collected amount for PM charity fund. To plot a pie chart from this data, you can do the following :

```
contri = [17, 8.8, 12.75, 14]          # contribution in thousands
plt.pie(contri)
```

The data range being plotted in pie( )

The output produced by above code will be :

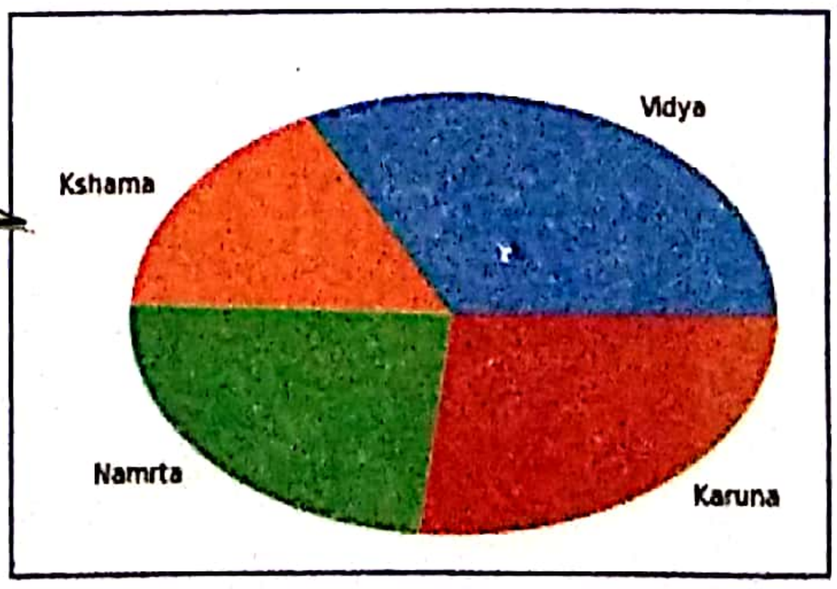
The adjacent chart shows the values of **contri** list as slices of a pie.

8.3.3A Labels of Slices of Pie

The above shown pie chart is just incomplete. You cannot make out which slice belongs to what. So, there must be labels next to the slices to make more readable and understandable. For this purpose, you need to create a sequence containing the labels and then specify this sequence as value for labels argument of pie() function. The first label is given to first value; second label to second value and so on, *e.g.*, see below :

```
contri = [17, 8.8, 12.75, 14]                    # contribution in thousands
houses = ['Vidya', 'Kshama', 'Namrta', 'Karuna']
plt.pie(contri, labels = houses)
```

Now the labels for the slices will be taken from the sequence namely houses.

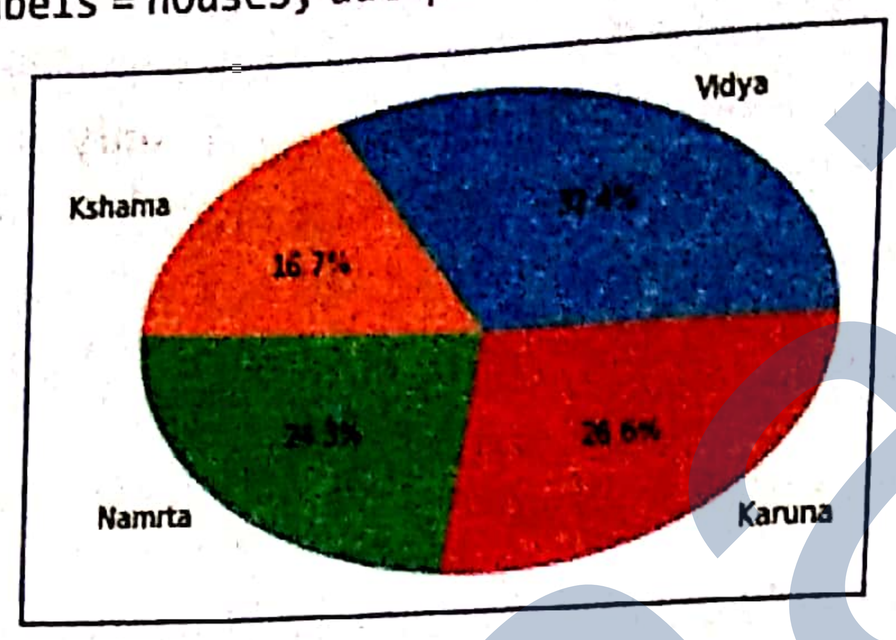


Now the output looks like :

8.3.3B Adding Formatted Slice Percentages to Pie

To view percentage of share in a pie chart, you need to add an argument **autopct** with a format string, such as **"%1.1F%%"**. It will show the percent share of each slice to the whole, formatted in a certain way, *e.g.*, see below :

```
plt.pie(contri, labels = houses, autopct ="%1.1f%%")
```



The percentage of each value plotted is calculated as : *single value/(sum of all values) * 100* and this also determines the size of the slice, *i.e.*, more the value bigger the slice.

For instance, if a list being plotted contains values as [30, 50, 10, 6], then

(*i*) First of internally sum of all these is calculated which is 30 + 50 + 10 + 6 = **96**.

(*ii*) Next each value's percentage share and its slice size is calculated as :

Slice 1 : 30/96 = 31.25%
Slice 2 : 50/96 = 52.08%
Slice 3 : 10/96 = 10.42%
Slice 4 : 6/96 = 6.25%

The format string used with **autopct** will determine the format of the percentage being displayed. The format string begins with the "%" operator, which specifies the format of the percentage value being displayed.

The general syntax for a format placeholder is

```
"%[flags][width][.precision]type"
```

- The first % symbol is a special character that signifies that it is a special string which will determine the format of the values to be displayed.
- The **width** determines the total number of characters to be displayed (digits before and after decimal point + 1 for decimal point). If the value being displayed has lesser number of digits than the width specified, then the value is padded as per the ***flag*** specified. If however, the value has more digits than the ***width*** specifies, then the full value is displayed. The examples below will make it clear.
- In pie charts, the most useful **flag** is 0 (zero), which when specified will pad the value being displayed with preceding zeros if the digits of value is less the ***width***.
- The ***precision*** is specified with digits and it specifies the number of digits after decimal point.

- The *type* specifies the type of value : *d* or *i* means integer type and *f* or *F* means float type.
- To print a % sign, given %% (two percentage signs) in the format string. The 2 %% signs are needed to print a percentage sign, otherwise single % is interpreted as a special character with a special meaning and thus used accordingly. By doubling %, *i.e.*, %%, you suppress its special meaning and thus a percentage sign is printed.

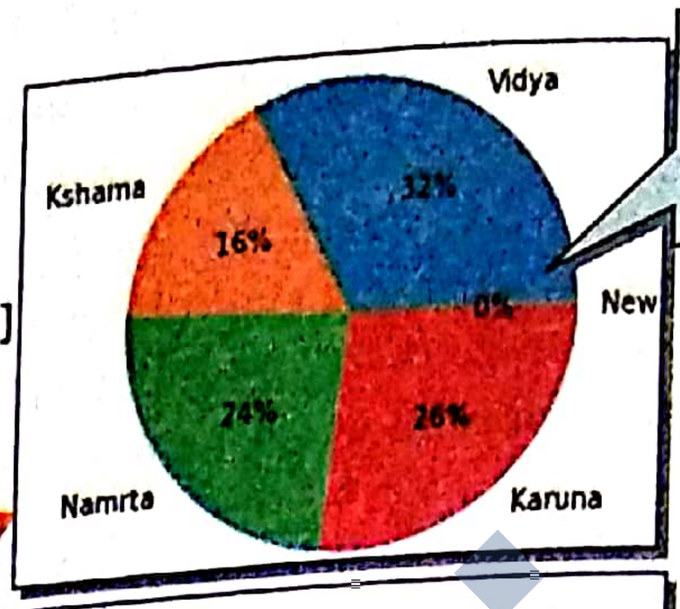
Now consider following examples of format strings as given below.

<table>
<tr><th>Format string</th><th>Description</th></tr>
<tr><td>"%5d" , "%5i"</td><td>width = 5, type = d or i (integer type)
Print the value with 5 characters e.g., if the value being printed is 123 then it will be printed as __123, because 123 consists only of 3 digits, the output is padded with 2 leading blanks.</td></tr>
<tr><td>"%05d", "%05i"</td><td>flag = 0, width = 5, type = d or i (integer type i.e., no decimal point)
Print the value with 5 characters and pad with leading zeros if necessary. E.g., if the value being printed is 123, then it will be printed as 00123, because value 123 consists only of 3 digits, the output is padded with 2 leading zeros to make it have width of 5 digits.</td></tr>
<tr><td>"%03d %%",
"%03i %%"</td><td>width = 3, type = d or i (integer type), percentage sign in the end
Print the value with 3 characters, pad with leading zeros if necessary and end with a % sign. E.g., if the value being printed is 123 then it will be printed as 123 % ; no leading zeros as the width 3 is fully filled.</td></tr>
<tr><td>"%6.2f",
"%6.2F"</td><td>width = 6, precision = 2, type = f (float type)
Print the value with space of 6 characters; after decimal point, keep the precision of 2 digits (rounding off may take place for this precision); pad with leading blank if necessary. Consider following examples :
<table>
<tr><th>Value</th><th>Formatted As</th><th>Remarks</th></tr>
<tr><td>12.679</td><td>_12.68</td><td>Rounded off for 2 precise digits ; 1 leading blank</td></tr>
<tr><td>0.018</td><td>__0.02</td><td>Rounded off for 2 precise digits ; 2 leading blanks</td></tr>
<tr><td>211.2</td><td>211.20</td><td>1 trailing zero added for 2 precise digits.</td></tr>
<tr><td>7</td><td>__7.00</td><td>2 trailing zeros added for 2 precise digits ; 2 leading blanks.</td></tr>
<tr><td>1009.27</td><td>1009.27</td><td>This time 7 characters will get printed (more than the width) as the number before the decimal point cannot be reduced and also, precision digits cannot be reduced as .27 is already precise to 2 digits.</td></tr>
<tr><td>1009.277</td><td>1009.28</td><td>7 characters will get printed as the number before the decimal point cannot be reduced and for 2 precision digits, 0.277 is rounded off as .28.</td></tr>
</table></td></tr>
<tr><td>"%6.2f%%",
"%6.2F%%"</td><td>Same as previous but with a percentage sign in the end.</td></tr>
</table>

Now consider some examples below where we added one new value each to lists **contri** and **houses** used above.

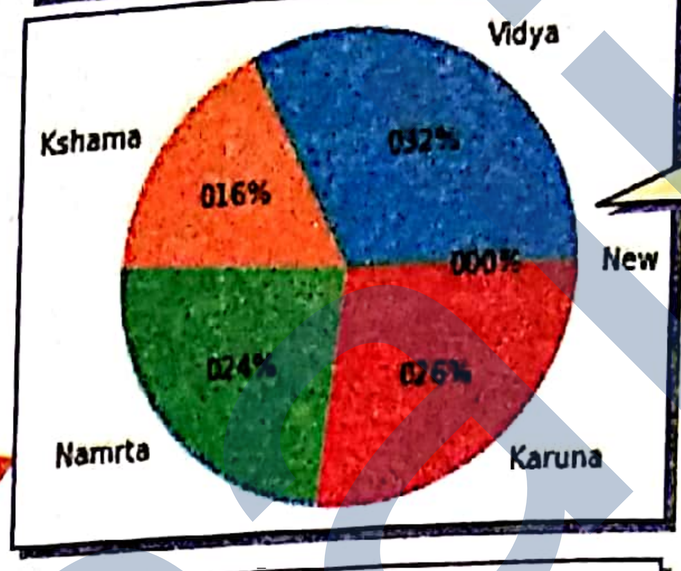
```
In [30]: contri
Out[30]: [17, 8.8, 12.75, 14, 0.1]

In [31]: houses
Out[31]: ['Vidya', 'Kshama', 'Namrta', 'Karuna', 'New']
```

```
plt.axis("equal")      # for circle shape
plt.pie(contri, labels = houses, autopct ="%3d%%" )
```



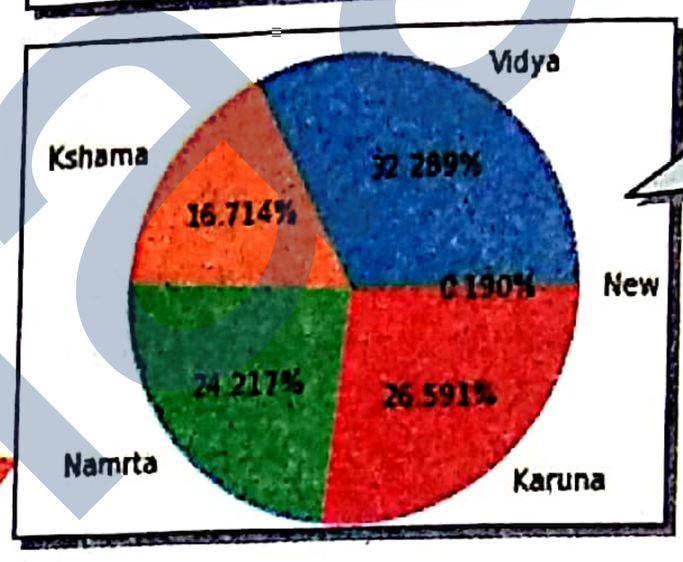
See, 'New' house's contri value 0.1 is depicted as 0% in d (Integer format)

```
plt.axis("equal")      # for circle shape
plt.pie(contri, labels = houses, autopct ="%03d%%")
```



See, leading zeros have been padded this time to make it 3 characters wide

```
plt.axis("equal")      # for circle shape
plt.pie(contri, labels = houses,  autopct ="%05.3f%%")
```



See, trailing zeros have been added this time to make it 3 digits precise

8.3.3C Changing Colors of the Slices

By default, the **pie( )** function shows each slice with a different color. If you are not satisfied with the default colors, you can specify own colors for the slices. For this purpose, you need to create a sequence containing the color codes or names for each slice and then specify this sequence as a value for **colors** argument of **pie( )** function. The first color is given to first value; second color to second value and so on, *e.g.*, see below (considering the same **contri** and **houses** lists containing five values each).

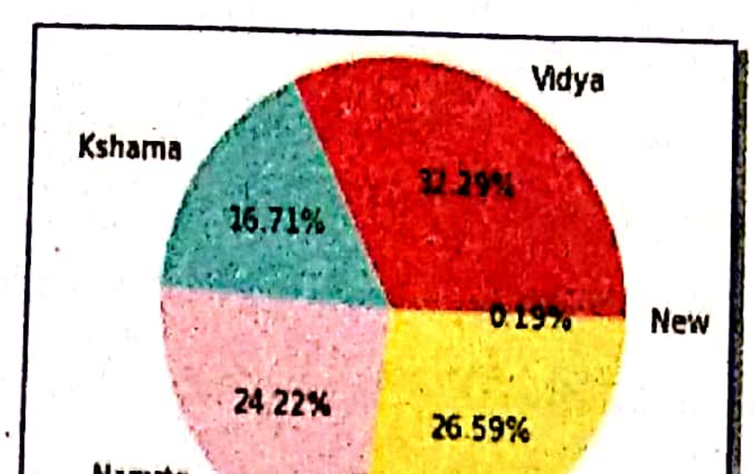
We need a sequence containing five colors for five values of list **contri**, so we created a sequence **colr** containing five color names (you may also give color codes such as 'r', 'b', 'g', etc.)

```
colr = ['red', 'cyan', 'pink', 'yellow', 'silver']
```

Now we issued followed command :

```
plt.pie(contri, labels = houses, colors = colr, autopct ="%2.2f%%" )
```

And the result we got was :



See, the colors to each slices have been given from **colr** sequence in order (1st color to 1st value,, 2nd color to 2nd value and so on)

8.3.3D Exploding a Slice

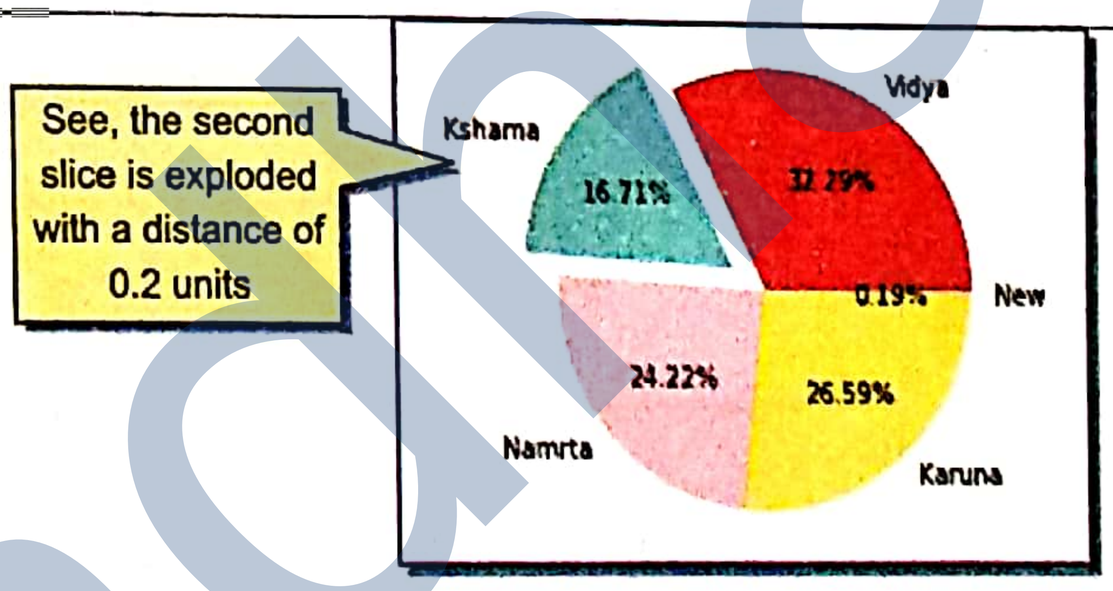
Sometimes you want to emphasize on one or more slices and show them little pulled out. This feature is called **explode** in pie charts. You can provide explode values for the slices in the form of a sequence, *e.g.*,

- if you want to **explode first slice** out of five slices being plotted with a distance of 0.2 units then your sequence for explode will be like : [0.2, 0, 0, 0,0]
- To **explode 3rd and 5th** slices by a distance of 0.15 and 0.3 units out of five slices, the explode sequence will be [0, 0, 0.15, 0, 0.3]

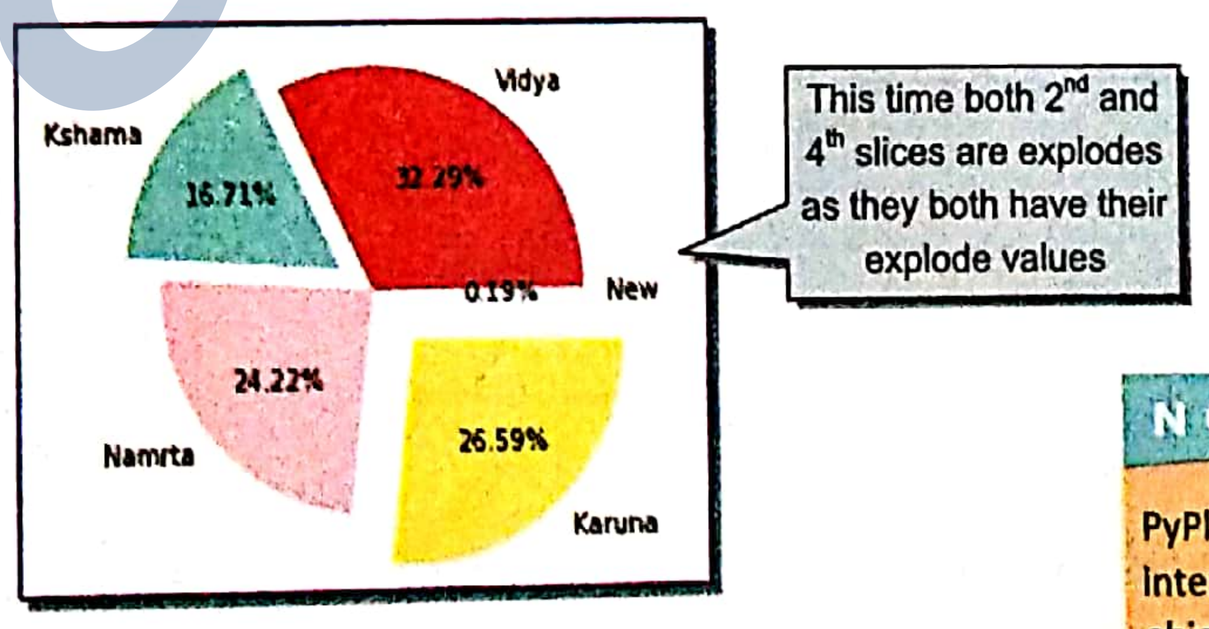
You need to specify this explode sequence as value to explode argument of **pie( )** function :

```
expl = [0, 0.2, 0, 0, 0]
plt.pie(contri, labels = houses, explode = expl, colors = colr, autopct ="%2.2f%%" )
```

```
In [46]: expl = [0, 0.2, 0, 0, 0]

In [47]: plt.axis("equal")
    ...: plt.pie(contri, labels = houses, explode = expl, colors = colr, autopct ="%2.2f%%" )
    ...:
Out[47]:
```



If you change the **expl** sequence to [0, 0.2, 0, 0.3, 0] where there are explode values for 2nd and 4th slices, the pie chart will look like :



NOTE

PyPlot provides a procedural interface to the matplotlib object-oriented plotting library. It is modeled closely after Matlab™.

Thus, we can summarize that the complete form of **pie( )** function is as shown below :

```
<matplotlib.pyplot>.pie(data [, labels = <labels sequence>] [, explode = <explode sequence>] [, colors = <color sequence>] [, autopct = <format string> ])
```

8.4 CUSTOMIZING THE PLOT

Till now you have learnt to create line charts, bar charts and pie charts using pyplot interface. But here, let me ask you a question – are you satisfied with the outcome or the plots that you created using **plot( )**, **bar( )** and **pie( )** functions? Hmm, even I am not satisfied. These plots do not show any details like what a chart is showing *i.e.*, its title, legends, X and Y axes' details etc. are not shown and so on.

As mentioned earlier that '*Data-visualization-needs*' demand much more from a graph/plot. The graph or plot should have a proper *title*, X and Y limits defined, *labels*, *legends* etc. All this makes understanding the plot and taking the decisions easier. In the following lines, we are covering how you can customize your plots but before that you must know what all makes a plot and what all you can customize, *i.e.*, the anatomy of a chart.

8.4.1 Anatomy of a Chart

Any graph or chart that you create using **matplotlib's PyPlot** interface, is created as per a specific structure of a plot or shall we say a *specific anatomy*. As ***anatomy***, generally refers to study of bodily structure (or parts) of something, here we shall discuss various parts of a plot that you can create using PyPlot.

PyPlot charts have hierarchical structures or in simple words they are actually like containers containing multiple items/things inside it. Look at the figure given below carefully and then go through the terms given below it describe it.

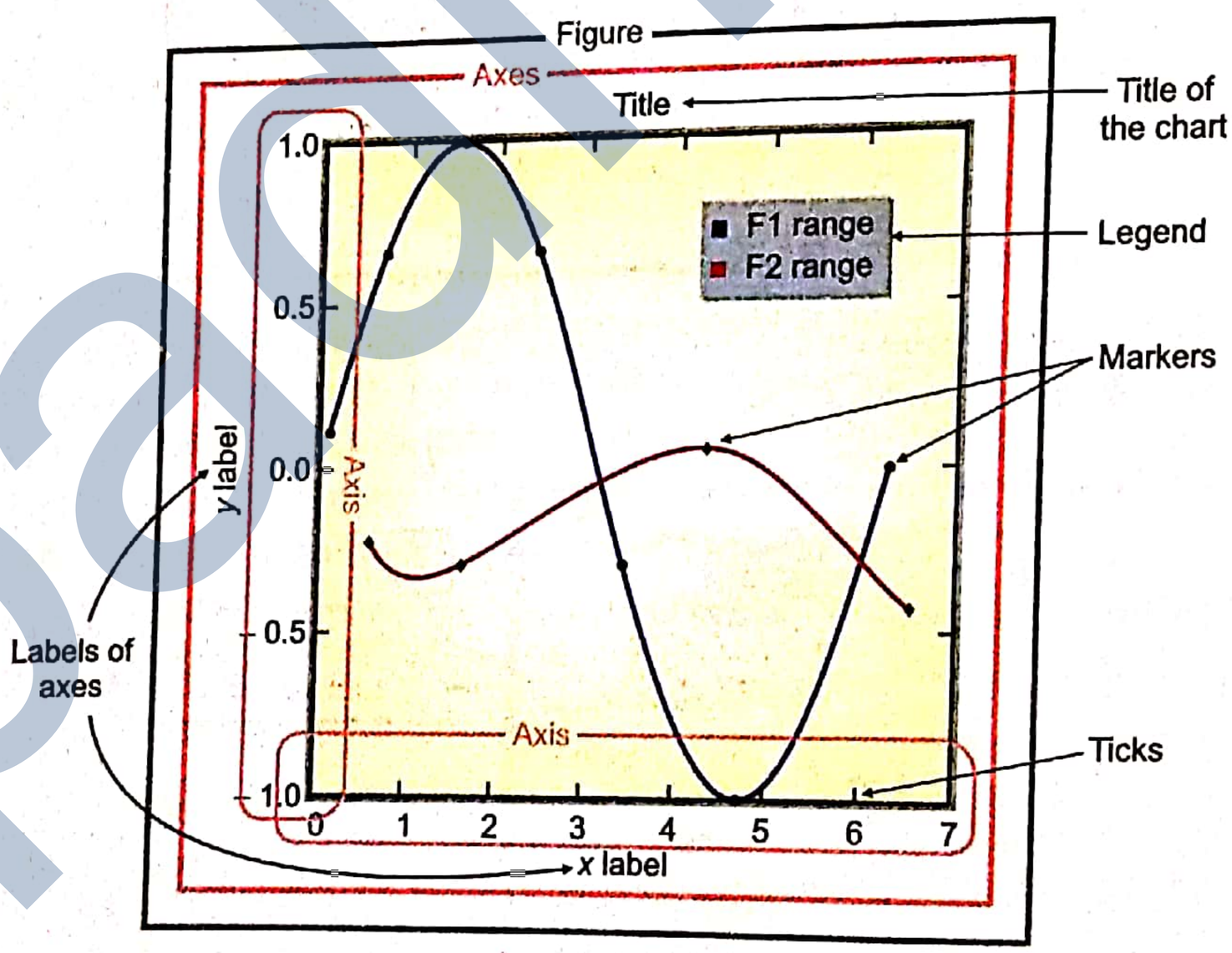


Figure 8.2 Anatomy of a Chart.

Let us now talk about various parts of a chart as shown in Fig. 8.2 above.

- **Figure.** PyPlot by default plots every chart into an area called *Figure*. A figure contains other elements of the plot in it.

- **Axes.** The axes define the area (mostly rectangular in shape for simple plots) on which actual plot (line or bar or graph etc.) will appear. Axes have properties like ***label, limits*** and ***tick marks*** on them.

  There are two axes in a plot : (*i*) **X–axis,** the horizontal axis , (*ii*) **Y-axis,** the vertical axis.

  - **Axis label.** It defines the name for an axis. It is individually defined for X-axis and Y-axis each.
  - **Limits.** These define the range of values and number of values marked on X-axis and Y-axis.
  - **Tick_Marks.** The tick marks are individual points marked on the X-axis or Y-axis.

- **Title.** This is the text that appears on the top of the plot. It defines what the chart is about.
- **Legends.** These are the different colors that identify different sets of data plotted on the plot. The legends are shown in a corner of the plot.

Now that you know what components a plot can have, let us learn how you can customize these and create informative, user friendly charts using PyPlot.

8.4.2 Adding a Title

To add a title to your plot, you need to call function title( ) before you show your plot. The syntax of title( ) function is as :

```
<matplotlib.pyplot>.title(<title string>)
```

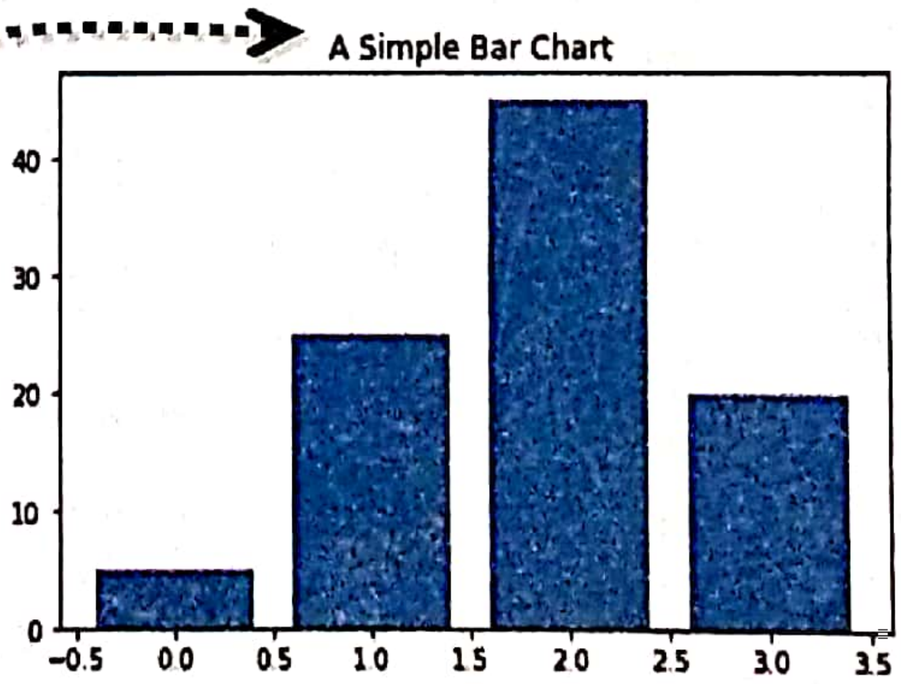
E.g.,

```
plt.title( "A Bar chart")
```

will add a title string "A Bar Chart" to your plot at the top, outside the axes area (assuming that the *matplotlib.pyplot* has been imported with alias name as **plt**).

Consider the following example :

```
X = np.arange(4)        #[0, 1, 2, 3]
Y = [5., 25., 45., 20.]
plt.bar(X, Y)
plt.title("A Simple Bar Chart")
plt.show()
```



You can use **title( )** function for all types of plots *i.e.,* for *plot( ),* for *bar( )* and for *pie( )* as well.

8.4.3 Setting X and Y Labels, Limits and Ticks

You already know about setting labels for X and Y axes as we have done it earlier. Recall that functions **xlabel( )** and **ylabel( )** can be used to set labels for X and Y axes respectively. Thus, we shall not cover them here again, but we shall talk about setting of *limits* and *ticks*.

8.4.3A Setting Xlimits and Ylimits

When you specify and X and Y ranges for plotting, PyPlot automatically tries to find best fitting range for X-axis and Y-axis depending on the data being plotted. But sometimes, you may need to have own limits specified for X and Y axes. For this, you can use **xlim( )** and **ylim( )** functions to set limits for x-axis and y-axis respectively. Make sure to set the limits keeping in mind the data being plotted.

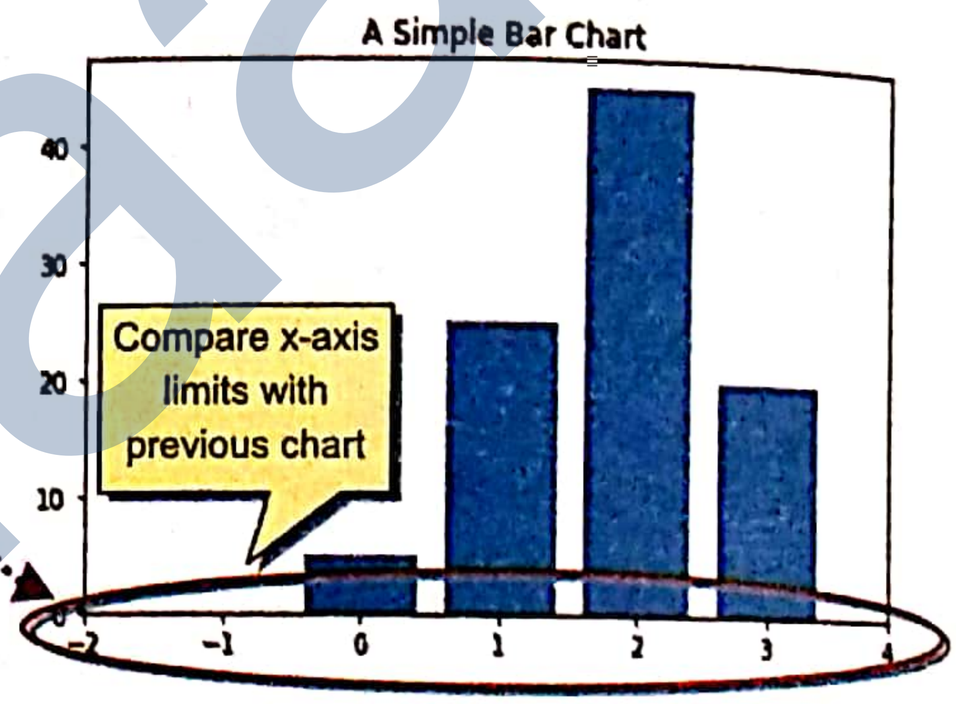
Both **xlim( )** and **ylim( )** are used as per following format :

```
<matplotlib.pyplot>.xlim(<xmin>, <xmax>)      # set the X-axis limits as xmin to xmax
<matplotlib.pyplot>.ylim(<ymin>, <ymax>)      # set the Y-axis limits as ymin to ymax
```

Let us reconsider the above example where we added a title to a bar chart in section 8.4.2 ; let us change the x limits for above chart :

```
X = np.arange(4)
Y = [5., 25., 45., 20.]
plt.xlim(-2.0, 4.0)
plt.bar(X, Y)
plt.title("A Simple Bar Chart")
plt.show()
```

X-axis limits changed

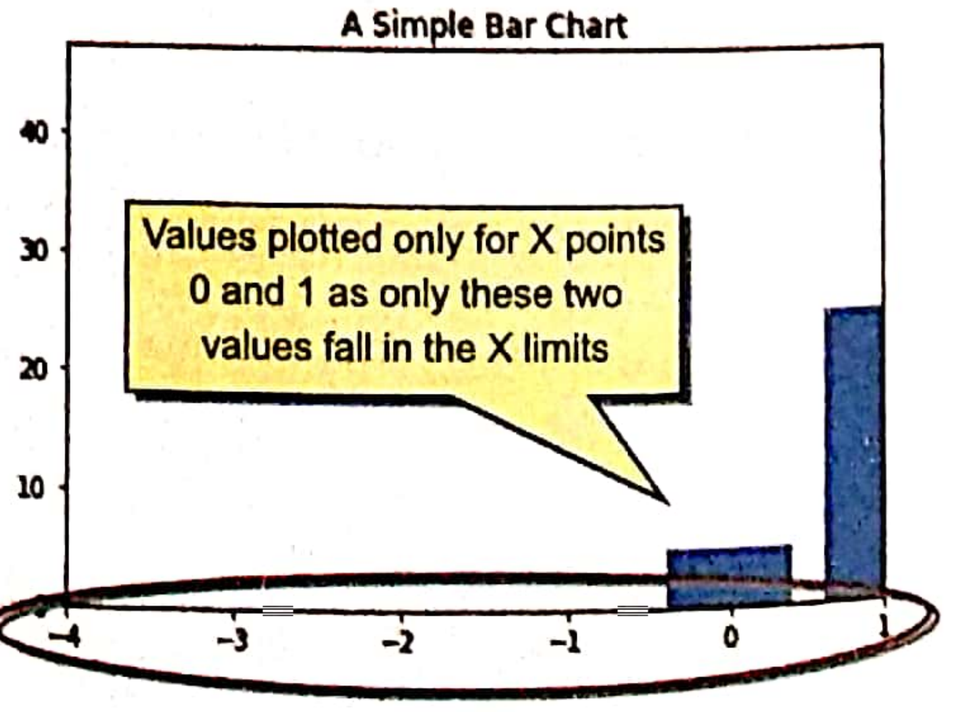


Important

While setting up the limits for axes, you must keep in mind that only the data that falls into the limits of *X* and *Y*-axis will be plotted; rest of the data will not show in the plot. In other words, if you have set *X*-axis or *Y*-axis limits which are not compatible with the data values being plotted, you may either get incomplete plot or the data being plotted is not visible at all.

Let us reconsider the previous chart and change the *xlimits* – carefully go through the code below and its implications :

```
X = np.arange(4)        #[0, 1, 2, 3]
Y = [5., 25., 45., 20.]
plt.xlim(-4.0, 1.0)
plt.bar(X, Y)
plt.title("A Simple Bar Chart")
plt.show()
```



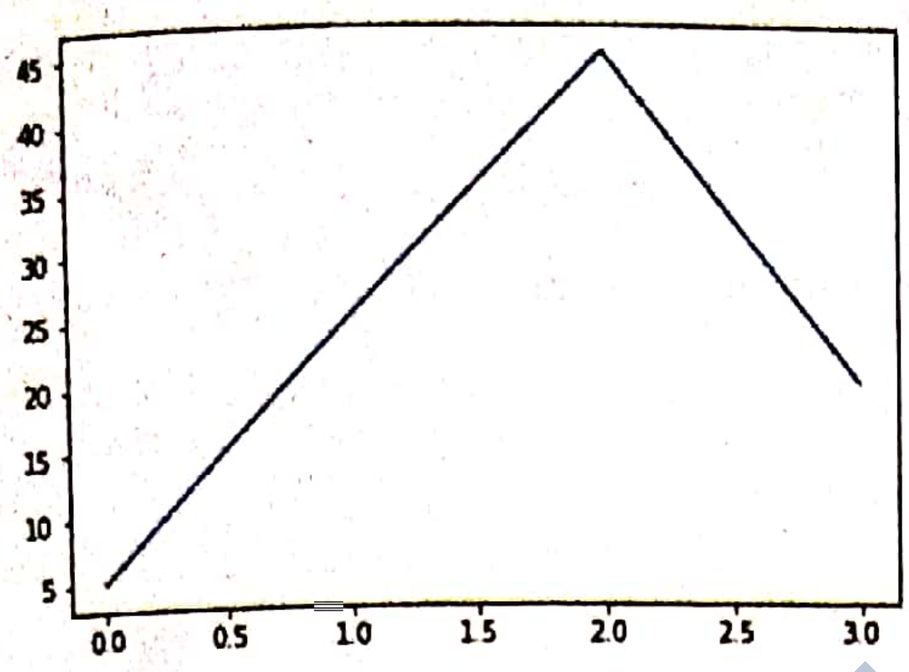
NOTE

Only the data values mapping on xlimits and ylimits will get plotted. If no data values maps to the xlimits or ylimits, nothing will show on the plot.

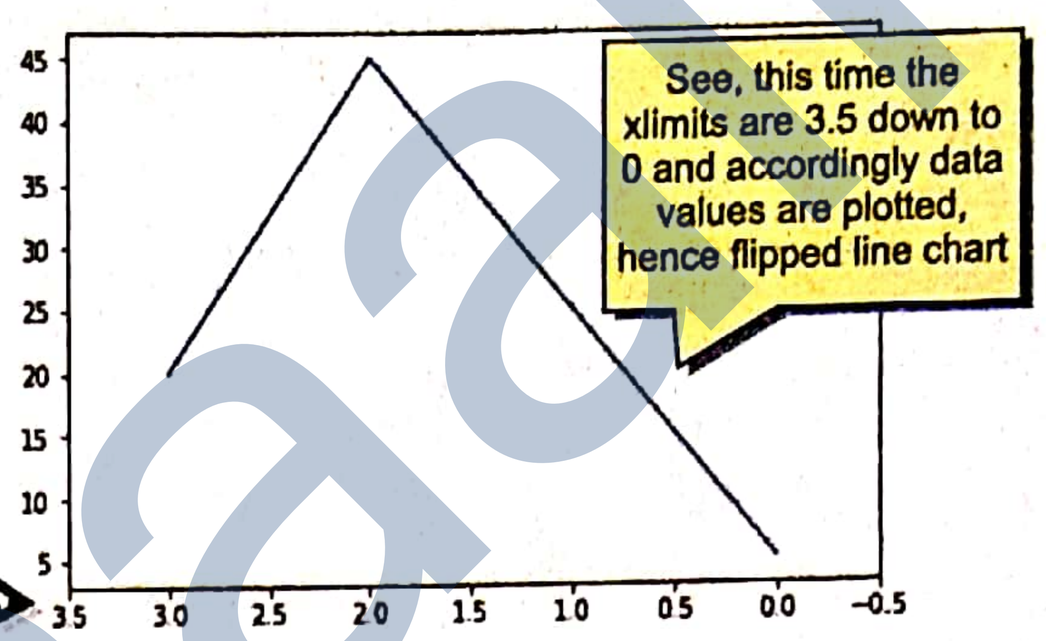
Interesting

You can use decreasing axes by flipping the normal order of the axis limits *i.e.*, if you swap the limits *(min, max)* as *(max, min)*, then the plot gets flipped, *e.g.*, see below :

```
In [11]: X = [0, 1, 2, 3]
    ...: Y = [5., 25., 45., 20.]
    ...: plt.plot(X, Y)
    ...: plt.show()
    ...:
```




```
In [15]: X = [0, 1, 2, 3]
    ...: Y = [5., 25., 45., 20.]
    ...: plt.xlim(3.5, -0.5)
    ...: plt.plot(X, Y)
    ...: plt.show()
    ...:
```



The *ylim( )* function works just the same as *xlim( )*, only thing is that it works on *Y*-axis.

NOTE

If you do not specify *x* or *y* limits, PyPlot will automatically decide the limits for *X* and *Y* axes as per the values being plotted.

8.4.3A Setting Ticks for Axes

By default, PyPlot will automatically decide which data points will have ticks on the axes, but you can also decide which data points will have tick marks on *X* and *Y* axes.

To set own tick marks :

- for X-axis, you can use **xticks( )** function as per format :

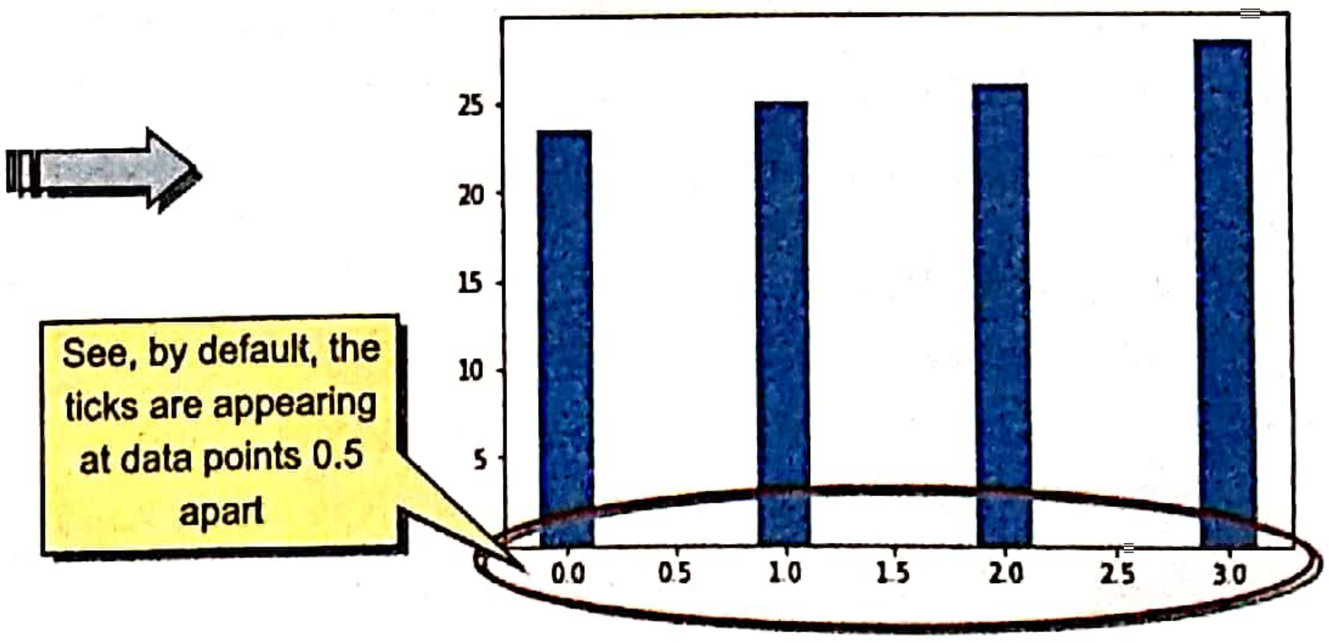
```
xticks(<sequence containing tick data points>,
       [<Optional sequence containing tick labels>])
```

- for Y axis, you can use **yticks( )** function as per format :

```
yticks(<sequence containing tick data points>,
       [<Optional sequence containing tick labels>])
```

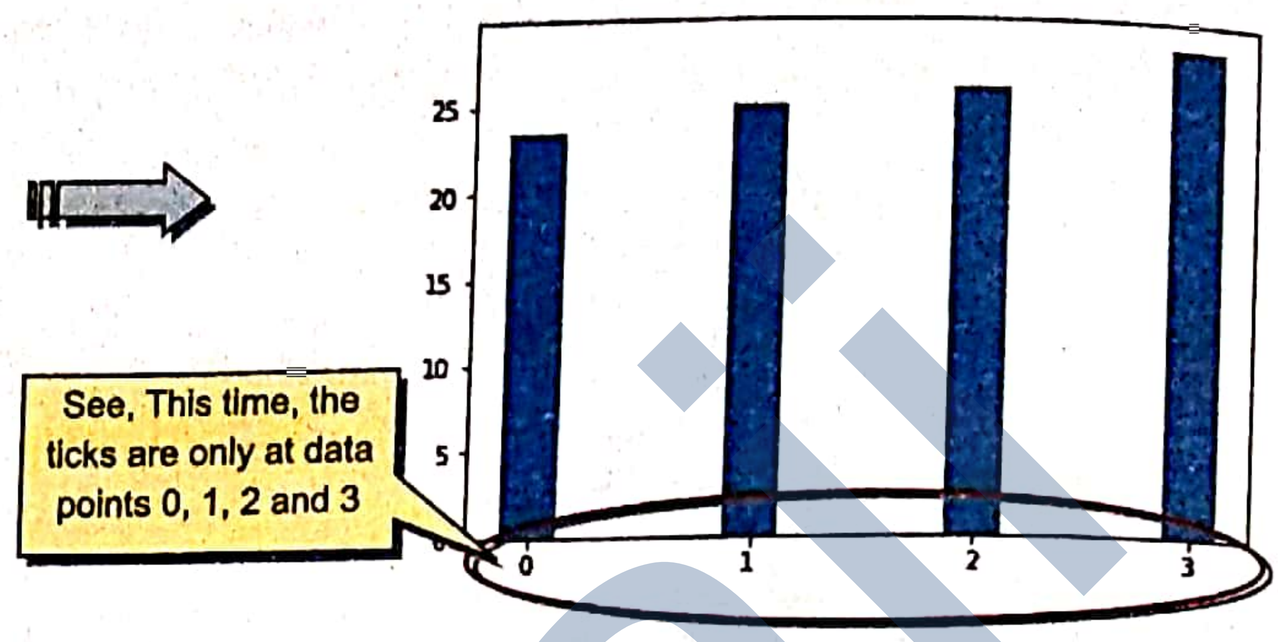
Let us understand this with the help of one example :

```
q = range(4)
s = [23.5, 25, 26, 28.5]
plt.bar(q, s, width = 0.25)
```



If you want that tick marks should appear only at data points 0, 1, 2 and 3, you will need to give code as :

```
q = range(4)
s = [23.5, 25, 26, 28.5]
plt.xticks([0,1, 2, 3])
plt.bar(q, s, width = 0.25)
```

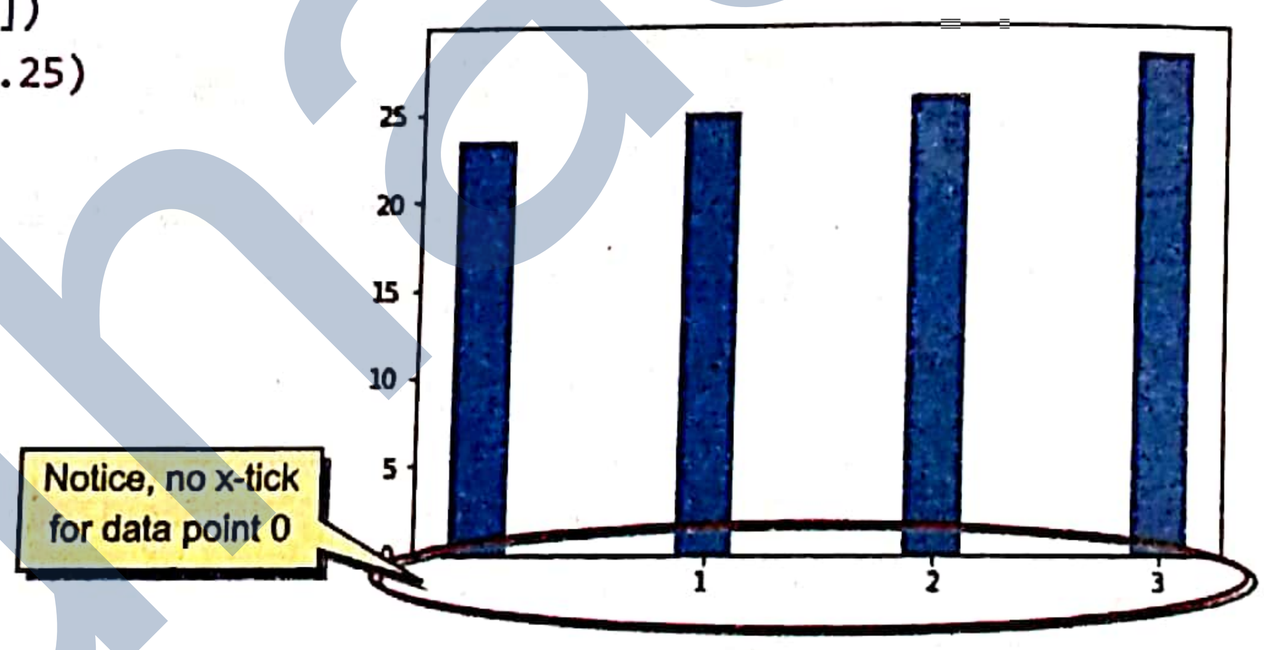


Now carefully go through the following code (it is very similar to the code above) :

```
q = range(4)
s = [23.5, 25, 26, 28.5]
plt.xticks([1, 2, 3, 4])
plt.bar(q,s, width = 0.25)
```

Can you figure out how the resultant plot will look like ?

Hmm. Well, you guessed it right – this time x-ticks will appear at data points 1, 2, 3.. but not on data point 0, *i.e.*, as :



You can specify own labels for tick points too by specifying an additional sequence containing labels for ticks. To understand this, consider following example.

Example 8.3 *TSS school celebrated volunteering week where each section of class XI dedicated a day for collecting amount for charity being supported by the school. Section A volunteered on Monday, B on Tuesday, C on Wednesday and so on. There are six sections in class XI. Amount collected by sections A to F are 8000, 12000, 9800, 11200, 15500, 7300.*

(*a*) *Create a bar chart showing collection amount.*

(*b*) *Plot the collected amount vs days using a bar chart.*

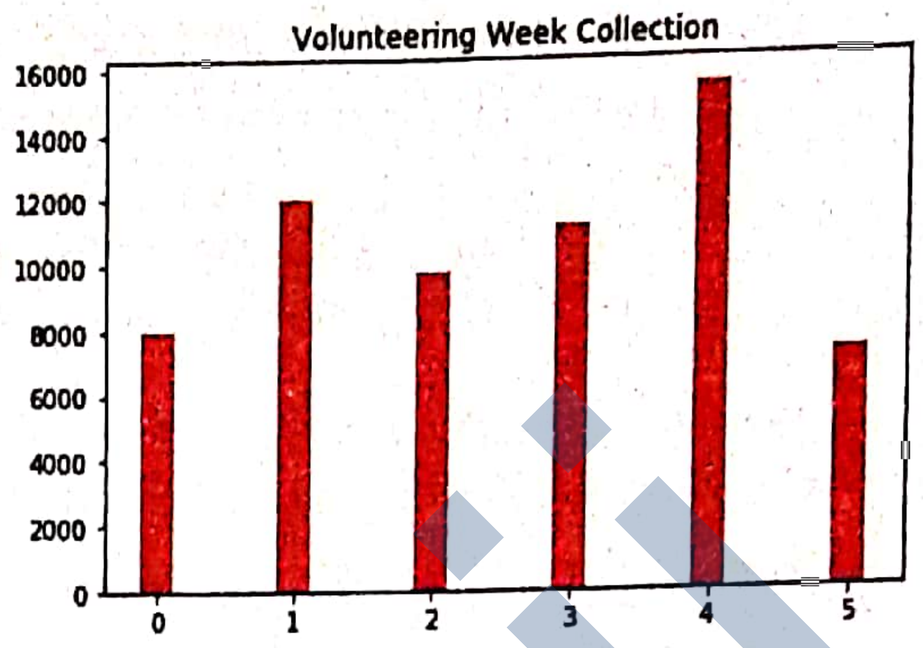
(*c*) *Plot the collected amount vs sections using a bar chart.*

Solution. Code common to all (*a*), (*b*) and (*c*) parts :

```
import numpy as np
import matplotlib.pyplot as plt
Col = [8000, 12000, 9800, 11200, 15500, 7300]
X = np.arange(6)                          # range to signify 6 days
plt.title("Volunteering Week Collection")
```

(*a*) Additional code for part (*a*) :

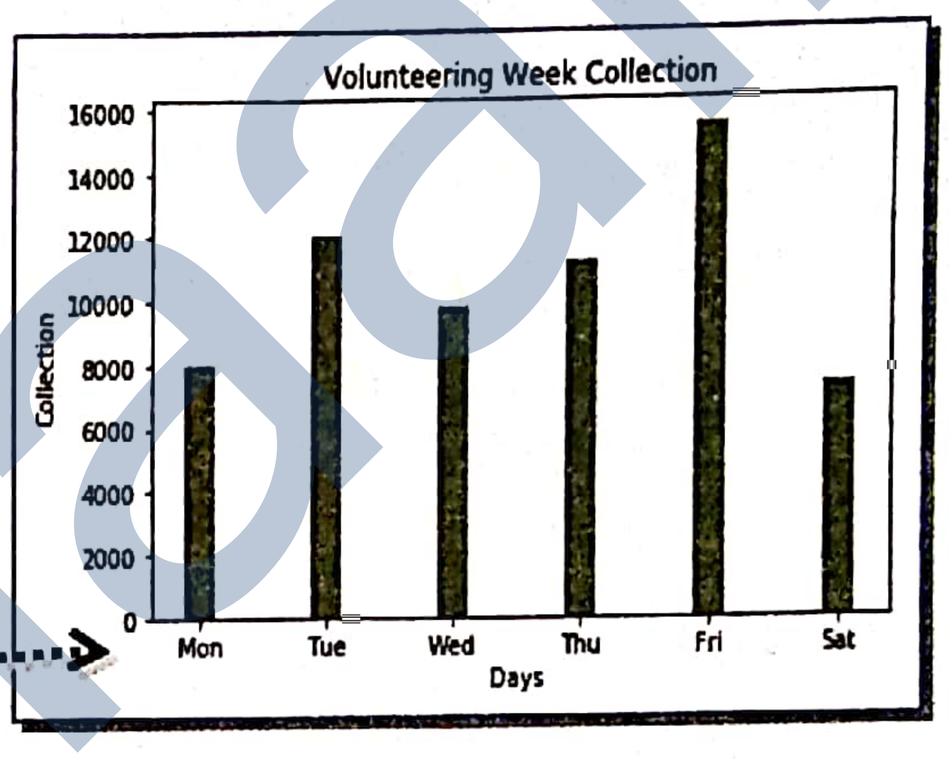
```
plt.bar(X, Col, color = 'r', width = 0.25)
plt.show()
```



(*b*) Additional code for part (*b*) :

```
plt.bar(X , Col, color = 'olive', width = 0.25)
plt.xticks( X, ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat'])
plt.xlabel("Days")
plt.ylabel("Collection")
plt.show()
```

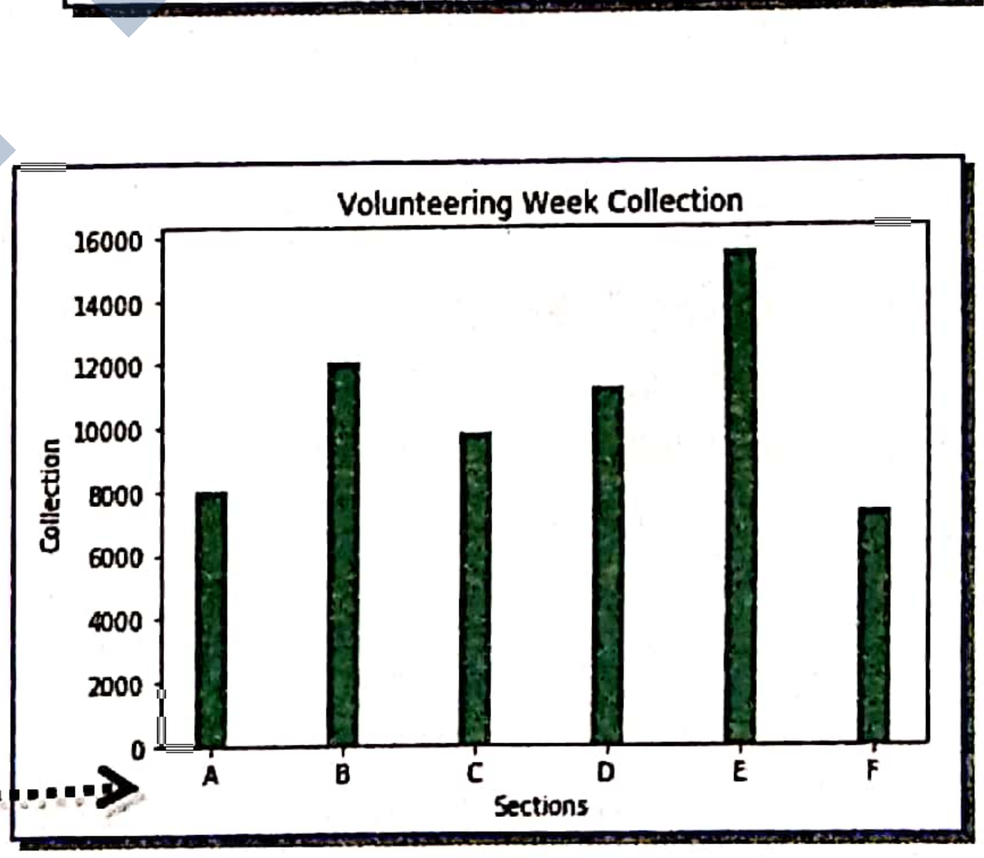
See, tick labels specified with additional sequence in xticks()



(*c*) Additional code for part (c) :

```
plt.bar(X , Col, color = 'g', width = 0.25)
plt.xticks(X, ['A', 'B', 'C', 'D', 'E', 'F'])
plt.xlabel("Sections")
plt.ylabel("Collection")
plt.show()
```

See, tick labels specified with additional sequence in xticks()



Please note that we have explained ticks using xticks( ) function above, yticks( ) also works in similar fashion – only difference being that it works for Y-axis.

8.4.4 Adding Legends

When we plot multiple ranges on a single plot, it becomes necessary that legends are specified. Recall that a legend is a color or mark linked to a specific data range plotted. To plot a legend, you need to do two things :

(*i*) In the plotting functions like plot( ), bar( ) etc., give a specific label to data range using argument **label**

(*ii*) Add legend to the plot using legend( ) as per format :

```
<matplotlib.pyplot>.legend(loc = <position number or string>)
```

The **loc** argument can either take values 1, 2, 3, 4 signifying the position strings *'upper right'*, *'upper left'*, *'lower left'*, *'lower right'* respectively. Default position is *'upper right'* or 1.

Let us understand this process with an example. Let us reconsider example 8.2 covered above, where we plotted three sublists in a bar chart. Let us add legends to this graph.

Code :

```
import numpy as np
import matplotlib.pyplot as plt

Val = [[5., 25., 45., 20.], [4., 23., 49., 17.], [6., 22., 47., 19.]]
X = np.arange(4)
```

#Step1 : specify label for each range being plotted using label argument, *i.e.*, :

```
plt.bar(X + 0.00, Val[0], color = 'b', width = 0.25, label = 'range1')
plt.bar(X + 0.25, Val[1], color = 'g', width = 0.25, label = 'range2')
plt.bar(X + 0.50, Val[2], color = 'r', width = 0.25, label = 'range3')
```

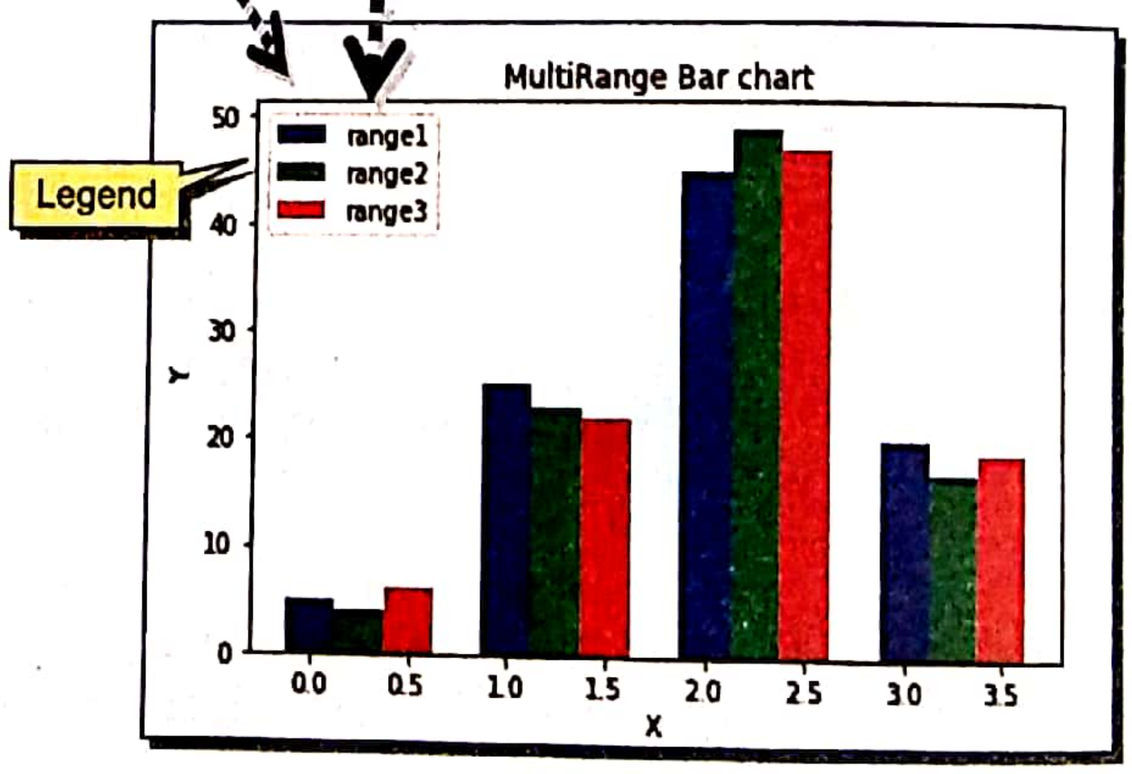
Step2 : add legend, *i.e.*, :

```
plt.legend(loc = 'upper left')
```

additional and optional chart formatting :

```
plt.title("MultiRange Bar chart")
plt.xlabel('X')
plt.ylabel('Y')
plt.show()
```

The output produced by above code will be :



Compare it with the output of example 8.2 and see the difference.

You can use ***label*** argument and ***legend( )*** function with *plot( )* function too for multiline charts. Consider following example.

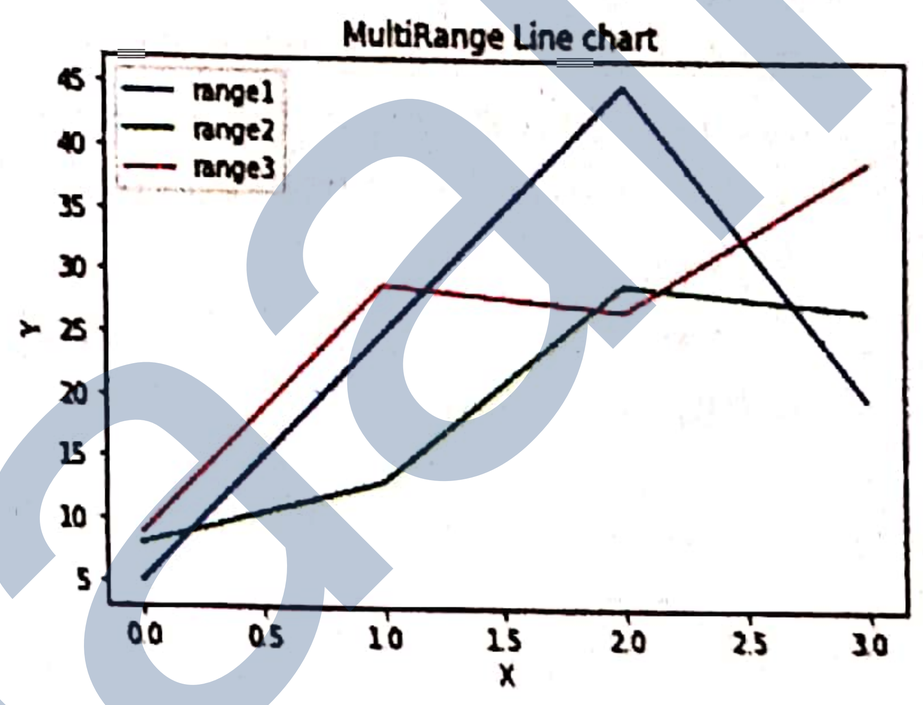
Example 8.4 *Create multiple line charts on common plot where three data ranges are plotted on same chart. The data range(s) to be plotted is/are :*

```
Data = [[5., 25., 45., 20.], [8., 13., 29., 27.], [9., 29., 27., 39.]]
```

Solution.

```
import numpy as np
import matplotlib.pyplot as plt
Data = [[5., 25., 45., 20.], [8., 13., 29., 27.], [9., 29., 27., 39.]]
X = np.arange(4)
plt.plot(X , Data[0], color = 'b', label = 'range1')
plt.plot(X, Data[1], color = 'g', label = 'range2')
plt.plot(X, Data[2], color = 'r', label = 'range3')
plt.legend(loc = 'upper left')
plt.title("MultiRange Line chart")
plt.xlabel('X')
plt.ylabel('Y')
plt.show()
```

The output produced by above code is :



8.4.5 Saving a Figure

If you want to save a plot created using pyplot functions for later use or for keeping records, you can use savefig( ) to save the plot.

You can use the pyplot's savefig( ) as per format :

```
<matplotlib.pyplot>.savefig(<string with filename and path>)
```

- You can save figures in popular formats like *.pdf, .png, .eps* etc. Specify the format as file extension.
- Also, while specifying the path, use double slashes to suppress special meaning of single slash character.

Consider the following examples :

```
plt.savefig("multibar.pdf")                 # save the plot in current directory
plt.savefig("c:\\data\\multibar.pdf")       # save the plot at the given path
plt.savefig("c:\\data\\multibar.png")       # save the plot at the given path in png format
```

You can open the folder to check yourself. After running the code, your plot will be saved there in the specified file format.

IMPORTANT

Please note that we have kept our discussion of pyplot functions very simple. Pyplot functions are capable of doing much more. Covering complete functionality of these functions is way beyond the scope of the book. However, if you want to learn and try more, then you may refer to matplotlib documentation.

8.5 COMPARING CHART TYPES

You have learnt to create line chart, bar charts and pie charts in above lines. But how will you decide the chart type while plotting data ranges. In the following lines, we shall briefly discuss when to use what ?

When to use a Line Graph ?

Line graphs are used to track changes or see trends over short and long periods of time. For smaller changes in data values, line graphs are better to use than bar graphs. In fact, line graphs are powerful visual tools that illustrate trends in data over a period of time or a particular correlation. For example, one axis of the graph might represent a variable value, while the other axis often displays a timeline, *e.g.*, the popularity of various social-media networks over the course of a year can be visually compared with ease through the use of a line graph.

Line graphs can also be used to compare changes over the same period of time for more than one group.

When to use a Pie Chart ?

Pie charts are best to use when you are trying to compare parts of a whole. They do not show changes over time. For example, a pie chart can quickly and effectively compare various budget allocations, population segments or market-research question responses.

Marketing content designers frequently rely on pie charts to compare the size of market segments. For example, a simple pie graph can clearly illustrate how the most popular mobile-phone manufacturers compare based on the sizes of their user-bases.

When to use a Bar Graph?

Bar graphs are used to compare things between different groups or to track changes over time. However, when trying to measure change over time, bar graphs are best when the changes are larger. For example, if you were showing the results for a class representative (CR) election in school, each candidate would have his or her own bar on the *x*-axis and the values on the *y*-axis would be number of votes the candidate received.

Similarly, if you were showing the revenues of various companies, you could use a bar chart with a bar for each company and the length corresponding to its revenue in rupees.

With this, we have come to the end of this chapter. Let us quickly revise what we have covered in the chapter.

Check Point 8.1

1. What is data visualization ?
2. Name the Python library generally used for data visualization.
3. Is Pyplot a Python library ? What is it ?
4. Which of the following are not a valid plotting function of pyplot ?
   (*a*) plot( ) (*b*) bar( )
   (*c*) line( ) (*d*) pie( )
5. Which of the following plotting functions does not plot multiple data services ?
   (*a*) plot( ) (*b*) bar( )
   (*c*) pie( ) (*d*) barh( )
6. Name the function you will use to create a horizontal bar chart.
7. Which argument will you provide to change the following in a line chart ?
   (*i*) width of the line
   (*b*) colour of the line
8. What is a marker ? How can you change the marker type and colour in a plot ?
9. What is a legend ? What is its usage ?
10. What are ticks ? What for are they used ?

DATA VISUALIZATION WITH MATPLOTLIB'S PYPLOT INTERFACE

PriP Progress In Python 8.1

The pyplot interface is a popular procedure interface of matplotlib library that is used for mostly 2d plotting with MATLAB like functions. In this PriP session, you shall work with pyplot to explore the plotting capabilities of matplotlib.

⋮

Please check the practical component-book – Progress in Computer Science with Python and fill it there in PriP 8.1 under Chapter 8 after practically doing it on the computer.

>>>❖<<<

LET US REVISE

- *Data visualization basically refers to the graphical or visual representation of information and data using visual elements like charts, graphs, and maps etc.*
- *The **matplotlib** is a Python library that provides many interfaces and functionality for 2D-graphics similar to MATLAB's in various forms.*
- *PyPlot is a collection of methods within matplotlib which allow user to construct 2D plots easily and interactively. PyPlot essentially reproduces plotting functions and behavior of MATLAB.*
- *In order to use pyplot on your computers for data visualization, you need to first import it in your Python environment by issuing **import** command for **matplotlib.pyplot**.*
- *NumPy is a Python library that offer many functions for creating and manipulating arrays, which come handy while plotting.*
- *You need to import numpy before using any of its functions.*
- *Numpy arrays are also called ndarrays.*
- *Some commonly used chart types are line chart, bar chart, pie chart, scatter chart etc.*
- *A line chart or line graph is a type of chart which displays information as a series of data points called 'markers' connected by straight line segments.*
- *You can create line charts by using pyplot's **plot( )** function.*
- *You can change line color, width, line-style, marker-type, marker-color, marker-size in plot( ) function.*
- *Possible line styles are : 'solid' for solid line, 'dashed' for dashed line, 'dotted' for dotted line and 'dashdot' for dashdotted line.*
- *A Bar Graph/Chart is a graphical display of data using bars of different heights.*
- *You can create bar chart using pyplot's bar( ) function.*
- *You can change colors of the bars, widths of the bars in bar( ) function.*
- *Use barh( ) function to create horizontal bar chart.*
- *The pie chart is a type of graph in which a circle is divided into sectors that each represent a proportion of the whole.*
- *You can create a pie chart using pie( ) function. The pie( ) chart plots a single data range.*
- *By default, the pie chart is oval in shape but you can change to circular shape by giving command **<matplotlib.pyplot>.axis("equal")**.*
- *The plot area is known as figure and every other element of chart is contained in it.*

- *The axes can be labelled using* ***xlabel( )*** *and* ***ylabel( )*** *functions.*
- *The limits of axes can be defined using* ***xlim( )*** *and* ***ylim( )*** *functions.*
- *The tick marks for axes values can be defined using* ***xticks( )*** *and* ***yticks( )*** *functions.*
- *The* ***title( )*** *function adds title to the plot.*
- *Using* ***legend( )*** *function, one can add legends to a plot where multiple data ranges have been plotted, but before that the data ranges must have their* ***label*** *argument defined in plot( ) or bar( ) function.*
- *The* ***loc*** *argument of legend( ) provides the location for legend, which by default is 1 or "upper right".*

Solved Problems

1. *What is data visualization ? What is its significance ?*

   Solution. Data visualization is a general term that describes any effort to help people understand the significance of data by placing it in a visual context. In simple words, Data visualization is the process of displaying data/information in graphical charts, figures and bars.

   Patterns, trends and correlations that might go undetected in text-based data can be exposed and recognized easier with data visualization techniques or tools such as line chart, bar chart, pie chart, histogram, scatter chart etc. Thus with data visualization tools, information can be processed in efficient manner and hence better decisions can be made.

2. *What is Python's support for Data visualization ?*

   Solution. Python support data visualizations by providing some useful libraries for visualization. Most commonly used data visualization library is **matplotlib**.

   Matplotlib is a Python library, also sometimes known as the plotting library. The matplotlib library offers very extensive range of 2D plot types and output formats. It offers complete 2D support along with limited 3D graphic support. It is useful in producing publication quality figures in interactive environment across platforms. It can also be used for animations as well.

   There are many other libraries of Python that can be used for data visualization but matplotlib is very popular for 2D plotting.

3. *What is pyplot ? Is it a Python library ?*

   Solution. The *pyplot* is one of the interfaces of *matplotlib* library of Python. This interface offers simple MATLAB style functions that can be used for plotting various types of charts using underlying *matplotlib* library's functionality.

   Pyplot is an interface, *i.e.*, a collection of methods for accessing and using underlying functionality of a library, not a library. The *matplotlib* library has may other interfaces too, along with *pyplot* interface.

4. *Name some commonly used chart types.*

   Solution. Some commonly used chart types are line chart, bar chart, pie chart, scatter chart etc.

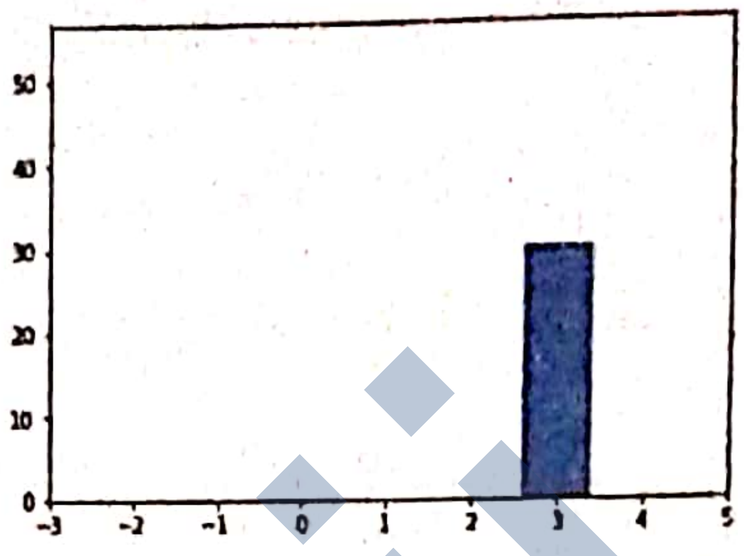
5. *Name the functions you will use to create a (i) line chart, (ii) bar chart, (iii) pie chart ?*

   Solution.

   (*i*) matplotlib.pyplot.plot( )
   (*ii*) matplotlib.pyplot.bar( )
   (*iii*) matplotlib.pyplot.pie( )

6. *Conisider the code given below (all required libraries are imported) and the output produced by it. Why is the chart showing one bar only while we are plotting four values on the chart ?*

```
:
a = [3, 6, 9, 12]
b = [30, 48, 54, 48]
plt.xlim(-3, 5)
plt.bar(a,b)
plt.show()
```

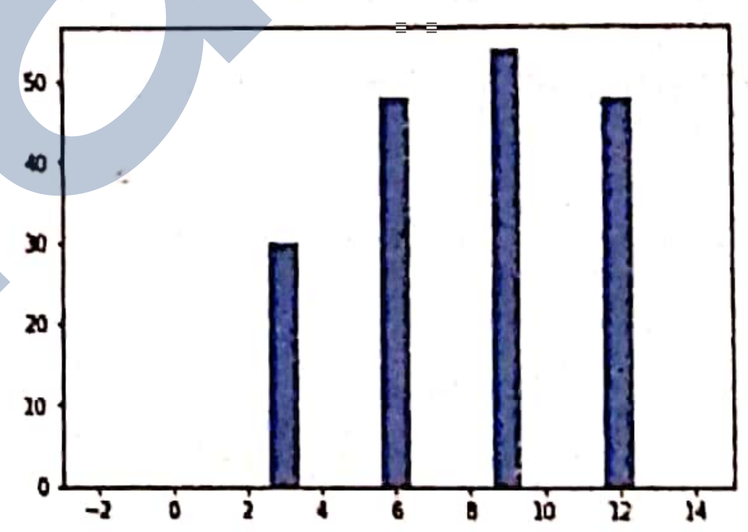


Solution. The given chart is showing a single bar as the limits of x axis have been set as –3 to 5. On this range, only one value from the data range being plotted falls *i.e.*, only a[0] and b[0] fall on this range. Thus only a single value b[0] *i.e.*, 30 is plotted against a[0] *i.e.*, 3.

7. *What changes will you make to the code of previous question so that the bars are visible for all four points. But do keep in mind that the x axis must begin from the point –3.*

Solution. If we change the limits of X-axis so that all the points being plotted fall in the range of limits, all values will show. Thus, we have changes the limits from –3 to 15, in place of –3 to 5.

```
plt.xlim(-3, 15)
plt.bar(a,b)
plt.show()
```



8. *Why is following code not producing any result ? Why is it giving errors ?*

*(**Note.** all required libraries have been imported and are available)*

```
a = range(10, 50, 12)
b = range(90, 200, 20)
matplotlib.pyplot.plot(a, b)
```

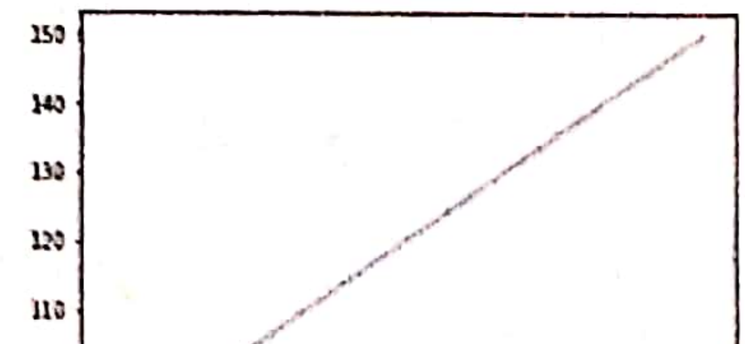
Solution. The above code is producing error because the two sequences being plotted *i.e.*, **a** and **b** do not match in shape. While sequence '**a**' contains 4 elements, sequence '**b**' contains 6 elements. For plotting, it is necessary that the two sequences being plotted must match in their shape.

9. *What changes will you recommend to rectify the error in previous question's code ?*

Solution. Since both the sequences being plotted must match in their shape, we can achieve this either by adding two elements to sequence a so that it has the same shape as sequence **b** (*i.e.*, 6 elements) or by removing two elements from sequence **b** so that it matches the shape of sequence **a** (*i.e.*, 4 elements).

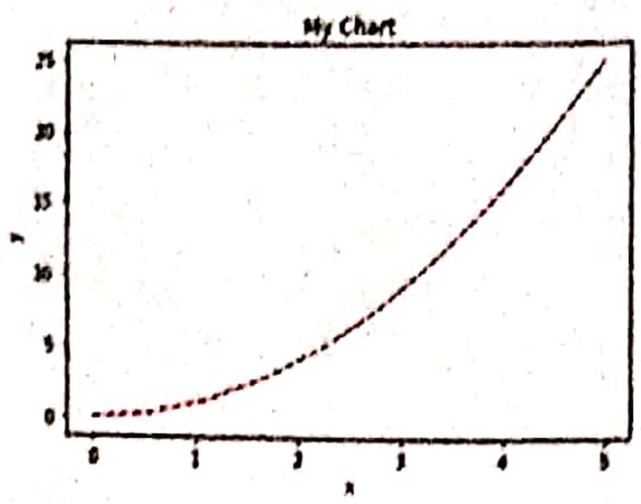
For instance,

```
a = range(10, 50, 12)
b = range(90, 160, 20)
matplotlib.pyplot.plot(a, b)
```

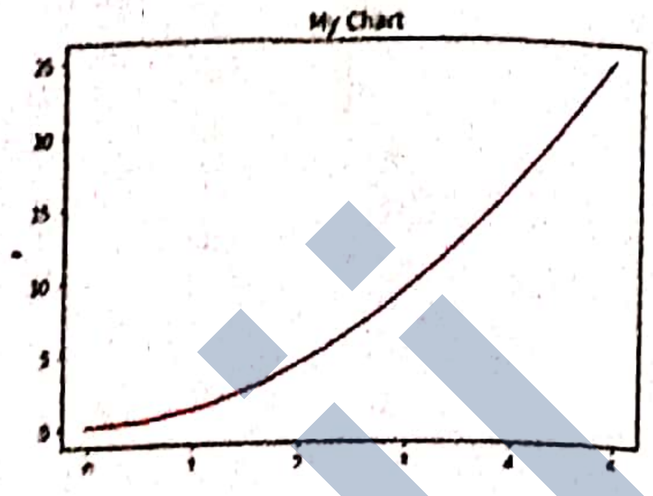


10. *Consider the two charts given below :*

(*a*)



(*b*)



Which of these charts has been produced by following code ?

```
plt.plot(x, y, 'r--')
plt.xlabel('x')
plt.ylabel('y')
plt.title('My Chart')
```

Solution. Chart (*a*) has been produced by above code as the linestyle is "--"

11. *Write code to produce the chart shown in q 10 b.*

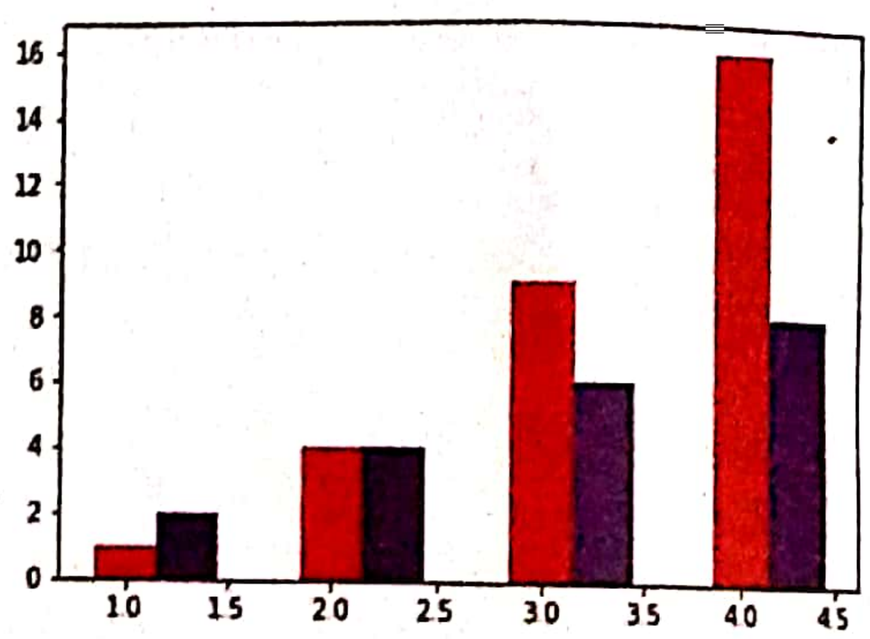
Solution.

```
plt.plot(x, y, 'r-')
plt.xlabel('x')
plt.ylabel('y')
plt.title('My Chart')
```

12. *Given an ndarray p as ([1, 2, 3, 4]). Write code to plot a bar chart having bars for p and p**2 (with red color) and another bar for p vs p*2 (with blue color). (assume that libraries have been imported)*

Solution.

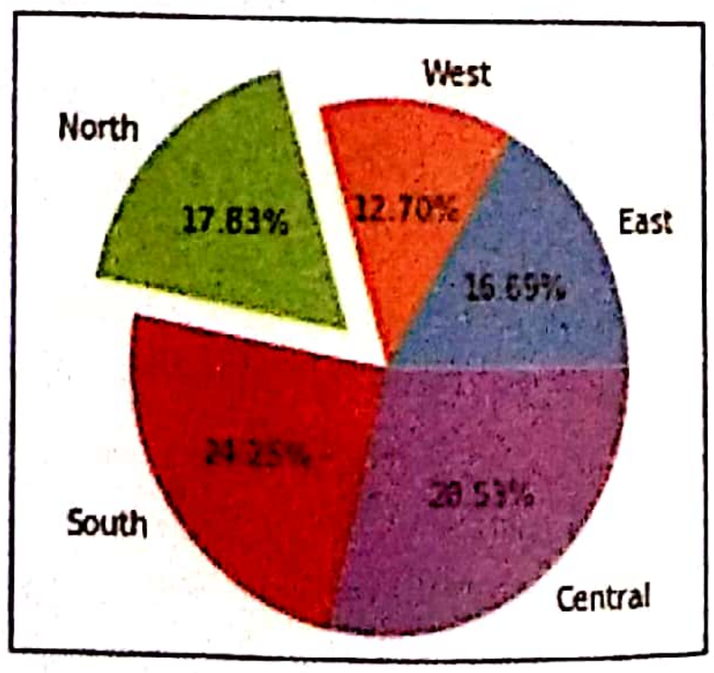
```
plt.bar(p, p**2, color = 'r', width = 0.3)
plt.bar(p+0.3, p*2, color = 'b', width = 0.3)
```



13. *Write to create a pie for sequence con = [23.4, 17.8, 25, 34, 40] for Zones = ['East', 'West', 'North', 'South', 'Central'].*

- Show North zone's value exploded
- Show % contribution for each zone
- The pie chart should be circular.

Solution.




```
import matplotlib.pyplot as plt
con = [23.4, 17.8, 25, 34, 40]
Zones = ['East', 'West', 'North', 'South', 'Central']
plt.axis("equal")
plt.pie(con, labels = Zones, explode = [0, 0, 0.2, 0, 0], autopct = "%1.2f%%")
plt.show()
```

GLOSSARY

Bar chart	A graphical display of data using bars of different heights.
Bar graph	Bar chart
Data visualization	The graphical or visual representation of information and data using visual elements like charts, graphs, and maps etc.
Line chart	A type of chart which displays information as a series of data points called 'markers' connected by straight line segments.
Line graph	Line chart
Pie chart	A type of graph in which a circle is divided into sectors that each represent a proportion of the whole.

Assignment

Type A : Short Answer Questions/Conceptual Questions

1. What is the significance of data visualization ?
2. How does Python support data visualization ?
3. What is the use of matplotlib and pyplot ?
4. What are the popular ways of plotting data ?
5. What are ndarrays ? How are they different from Python lists ?
6. Compare bar( ) and barh( ) functions.
7. What is the role of legends in a graph/chart ?
8. What will happen if you use legend( ) without providing any label for the data series being plotted ?
9. What do you understand by xlimit and ylimit ? How are these linked to data being plotted ?
10. When should you use (*i*) a line chart, (*ii*) a bar chart, (*iii*) a pie chart ?
11. Write code in Python to create a lists L1 having even elements between 1 – 10. Create another list L2 that stores the cubes of elements of list L2.
12. Repeat the previous question but this time use numpy functions to create the two sequences.
13. A list temp contains average temperatures for seven days of last week. You want to see how the temperature changed in last seven days. Which chart type will you plot for the same and why ?
14. Write code to practically produce a chart for question 8.
15. A company has three offices across India : Delhi, Mumbai, Kolkata. Each of the offices have sent a compiled list sharing sales in quarter1, quarter2, quarter3 and quarter4.
    - Suggest the best chart type to plot all of the above data ?
    - Why did you choose that specific chart type ?
    - Can you create a pie chart from above data ?
16. Write code to produce a chart plotting all the data given in question 10.

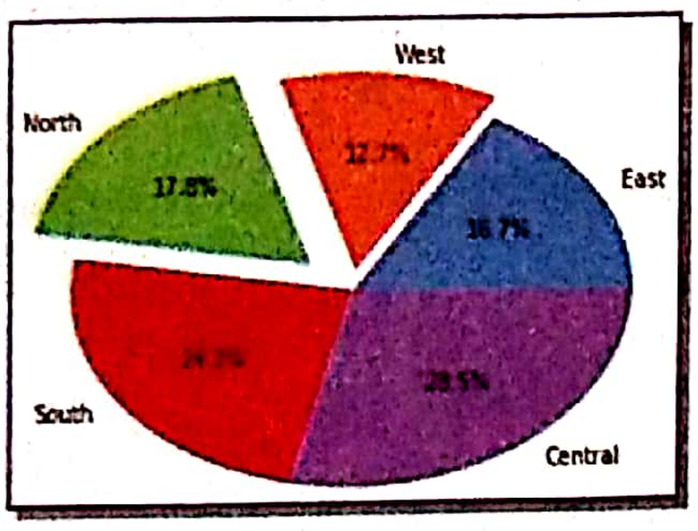
Type B : Application Based Questions

1. Consider the code below :

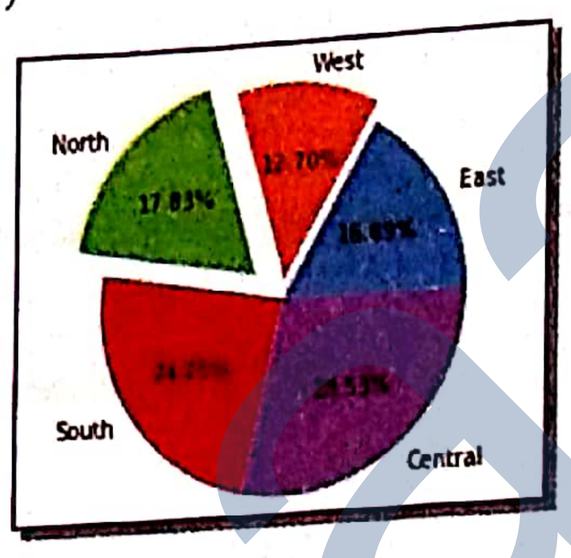
```
import matplotlib.pyplot as plt
con = [23.4, 17.8, 25, 34, 40]
Zones = ['East', 'West', 'North', 'South', 'Central']
plt.axis("equal")
plt.pie(con, labels = Zones, explode= [0, 0.1, 0.2, 0, 0], autopct = "%1.1f%%")
```

Which of the following pie charts are created by above code ?

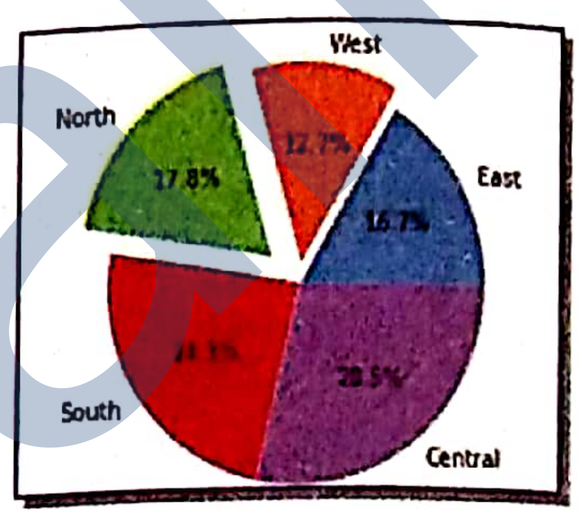
(*a*)



(*b*)



(*c*)



2. Execute the following codes and find out what happens ? (Libraries have been imported already ; plt is the alias name for matplotlib.pyplot)

(*a*)

```
A = np.arange(2, 20, 2)
B = np.log(A)
plt.plot(A, B)
```

(*b*)

```
A = np.arange(2, 20, 2)
B = np.log(A)
plt.bar(A, B)
```

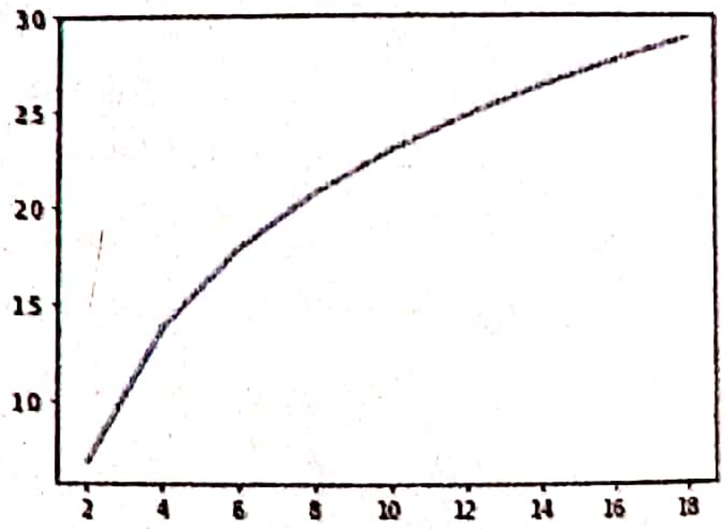
(*c*)

```
A = np.arange(2, 20, 2)
B = np.log(A)
plt.pie(A, B)
```

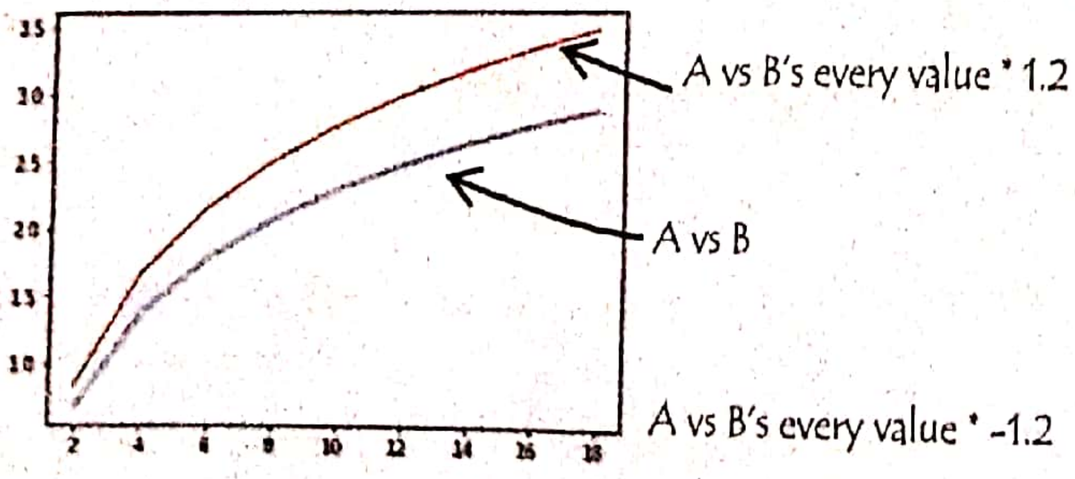
Will any code produce error ? Why ? Why not ?

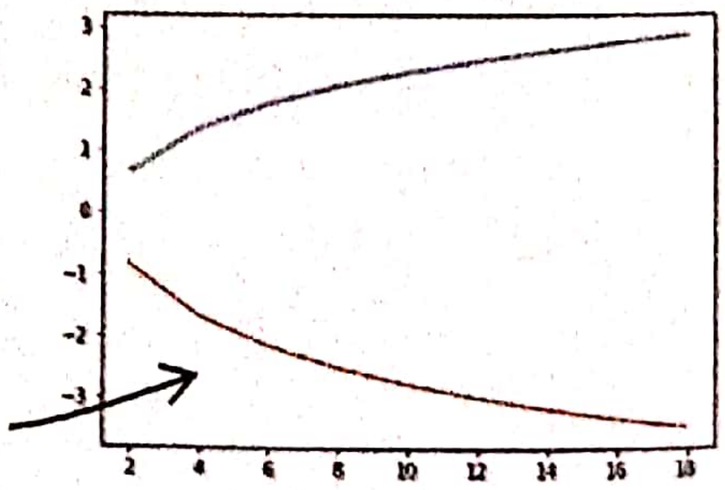
3. Add titles for the *X*-axis, *y*-axis and for the whole chart in above codes.

4. **plt.plot(A, B)** produces (A and B are the sequences same as created in question 2) chart as :



Write codes to produce charts as shown below :





5. Why is following command producing error ? (plt is the alias name of imported library matplotlib.pyplot and A is an *ndarray* created in question 2)

```
plt.pie(A, A+2, A+3, A+4)
```

Type C : Practical/Knowledge Based Questions

1. Consider the data given below. Create sequences required from the data below.

Rainfall in mm

Zones	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
North	140	130	130	190	160	200	150	170	190	170	150	120
South	160	200	130	200	200	170	110	160	130	140	170	200
East	140	180	150	170	190	140	170	180	190	150	140	170
West	180	150	200	120	180	140	110	130	150	190	110	140
Central	110	160	130	110	120	170	130	200	150	160	170	130

Write code to :

(*a*) Create bar charts to see the distribution of rainfall from Jan – Dec for all the zones.

(*b*) Create a pie chart to check the amount of rainfall in Jan separately.

(*c*) Create a line chart to observe any trends from Jan to Dec.

2. Consider the data given below :

App Name	App Price in Rs	Total Downloads
Angry Bird	75	197000
Teen Titan	120	209000
Marvel Comics	190	414000
ColorMe	245	196000
Fun Run	550	272000
Crazy Taxi	55	311000
Igram Pro	175	213000
WApp Pro	75	455000
Maths Formulas	140	278000

Using the above data, plot the following :

(*a*) A line chart depicting the prices of the apps.

(*b*) A bar chart depicting the downloads of the apps.

(*c*) Convert the *Est downloads* sequence that has each download value divided by 1000. Now create a bar chart that plots multiple bars for prices as well est downloads.

(*d*) The charts should have proper titles for the charts, axes, legends etc.