# File Handling

Introduction:

- Till now, we were taking the input from the console and writing it back to the console to interact with the user.
- Sometimes, it is not enough to only display the data on the console. The data to be displayed may be very large, and only a limited amount of data can be displayed on the console, and since the memory is volatile, it is impossible to recover the programmatically generated data again and again.
- If the data is stored in a place, it is called a file.
- Each & every organization depends on its data for its day to day activities that has been accumulating since its inception.
- Data being generated over the years is vast and require measures to handle. This is the reason computers are used to handle data, especially for storing and retrieving. Later on, programs are developed to process the data stored in computers.
- In computers, we store data in the form of files.

**Ex:** We can stores details of employees (i.e employee data) working in a organization like employee number, name, contact info, salary & so on in a file in the computer.

- From the perspective of computer, a file is a collection of data that is available to a program for processing.
- However, if we need to do so, we may store it onto the local file system which is volatile and can be accessed every time. Here, comes the need of file handling.

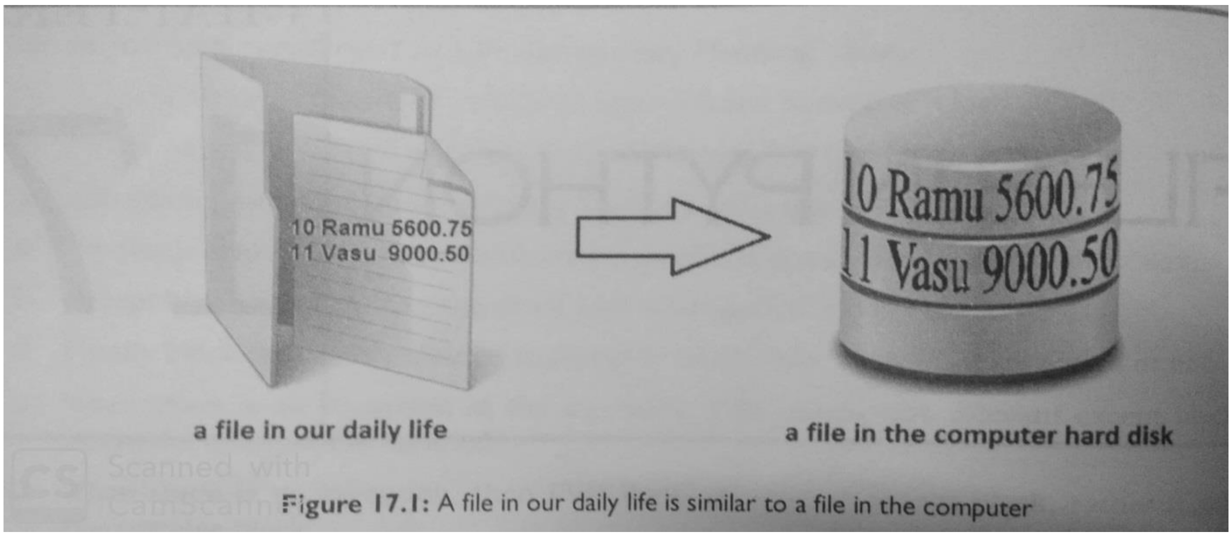


Figure 17.1: A file in our daily life is similar to a file in the computer

Advantages of storing data in a file.

1. Data can be stored permanently.
2. It enables the updating of the file data.
3. It enables the sharing of file across various programs.
4. Large amount of data can be stored

**<u>Types of Files:</u>**

There are 2 types of files

**1. Text Files**:

we can use text files to store character data.

eg: abc.txt

**2. Binary Files**: Binary files store entire data in the form of bytes, i.e a group of 8 bits each. Binary files can be used to store text, images, audio & video

**Opening a File:**

Before performing any operation (like read or write) on the file, first we have to open that file. For this we should use Python's inbuilt function open(). But at the time of open, we have to specify mode, which represents the purpose of opening file.

**f = open(filename, mode)**

- **open()** function accepts two arguments, file name and access mode in which the file is accessed. The function returns a file object which can be used to perform various operations like reading, writing, etc.
- **'file_name"** refers to name of the file which is to be opened in the "mode" specified using "open_mode" argument.

The allowed modes in Python are

1. **r :**open an existing file for read operation. The file pointer is positioned at the beginning of the file. If the specified file does not exist then we will get FileNotFoundError. This is default mode.
2. **w:** open an existing file for write operation. If the file already contains some data then it will be overridden. If the specified file is not already available then this mode will create that file.
3. **a :** open an existing file for append operation. It won't override existing data. If the specified file is not already available then this mode will create a new file.
4. **r+**: To read and write data into the file. The previous data in the file will not be deleted. The file pointer is placed at the beginning of the file.
5. **w+:** To write and read data. It will override existing data.
6. **a+ :** To append and read data from the file. It won't override existing data.
7. **x :** To open a file in exclusive creation mode for write operation. If the file already exists then we will get FileExistsError.(compulsory file should not be available)

**Note**: All the above modes are applicable for text files. If the above modes suffixed with 'b' then these represents for binary files.

Eg: rb,wb,ab,rb+,wb+,ab+

f = open("abc.txt","wb")

We are opening abc.txt file for writing data.

1. **rb** :It opens the file to read only in binary format. The file pointer exists at the beginning of the file.
2. **rb+**:It opens the file to read and write both in binary format. The file pointer exists at the beginning of the file.
3. **wb**: It opens the file to write only in binary format. It overwrites the file if it exists previously or creates a new one if no file exists with the same name. The file pointer exists at the beginning of the file.
4. **wb+**:It opens the file to write and read both in binary format. The file pointer exists at the beginning of the file.
5. **ab:** It opens the file in the append mode in binary format. The pointer exists at the end of the previously written file. It creates a new file in binary format if no file exists with the same name.
6. **ab+:**It opens a file to append and read both in binary format. The file pointer remains at the end of the file.

**Closing a File:**

After completing our operations on the file, it is highly recommended to close the file.

For this we have to use close() function.

f.close()

**Properties of File Object:**

Once we opend a file and we got file object, we can get various details related to that file by using its properties.

**name :**Name of opened file

**mode:** Mode in which the file is opened

**closed**: Returns boolean value indicates that file is closed or not

**readable()**: Retruns boolean value indicates that whether file is readable or not

**writable():** Returns boolean value indicates that whether file is writable or not.

Eg:

```
f=open("abc.txt",'w')
print("File Name: ",f.name)
print("File Mode: ",f.mode)
print("Is File Readable: ",f.readable())
print("Is File Writable: ",f.writable())
print("Is File Closed : ",f.closed)
f.close()
print("Is File Closed : ",f.closed)
```

## Writing data to text files:

We can write character data to the text files by using the following 2 methods.

write(str)…….used to write single line of text

writelines(list of lines)…. Used to write multiple lines of text

Eg:

```
 f=open("abcd.txt",'w')
 f.write("naresh\n")
 f.write("shankar\n")
 f.write("dileep kumar\n")
 print("Data written to the file successfully")
 f.close()
```

**abcd.txt:**

```
naresh
shankar
dleep kumar
```

In the above program, data present in the file will be overridden everytime if we run the program. Instead of overriding if we want append operation then we should open the file as follows.

f = open("abcd.txt","a")

**eg:**

```
f=open("abcd.txt",'w')
 list=["sunny\n","bunny\n","vinny\n","chinny"]
 f.writelines(list)
 print("List of lines written to the file successfully")
f.close()
```

Note: while writing data by using write() methods, compulsory we have to provide line seperator(\n), otherwise total data should be written to a single line.

**Reading Character Data from text files:**

We can read character data from text file by using the following read methods.

**read():** To read total data from the file

**read(n):** To read 'n' characters from the file

**readline():**To read only one line

**readlines()**: To read all lines into a list

**abc.txt:**

```
naresh
shankar
dleep kumar
```

**To read total data from the file**

```
f=open("abc.txt",'r')
data=f.read()
print(data)
f.close()
```

**output:**

```
naresh
shankar
dleep kumar
```

**To read only first 10 characters:**

```
f=open("abc.txt",'r')
data=f.read(10)
print(data)
f.close()
```

**output:**

```
naresh
sha
```

**To read data line by line:**

```
f=open("abc.txt",'r')
line1=f.readline()
print(line1,end='')
line2=f.readline()
print(line2,end='')
f.close()
```

**output:**

```
naresh
shankar
```

**To read all lines into list:**

```
f=open("abc.txt",'r')
lines=f.readlines()
for line in lines:
        print(line,end='')
f.close()
```

**<u>with statement:</u>**

The with statement can be used while opening a file.We can use this to group file operation statements within a block.

The advantage of with statement is it will take care closing of file,after completing all operations automatically even in the case of exceptions also, and we are not required to close explicitly.

Eg:

```
with open("abc.txt","w") as f:
   f.write("naresh\n")
   f.write("shankar\n")
   f.write("dleep kumar\n")
print("Is File Closed: ",f.closed)
print("Is File Closed: ",f.closed)
```

**output:**

```
Is File Closed:  True
Is File Closed:  True
```

## seek() and tell() methods:

**tell():**We can use tell() method to return current position of the cursor(file pointer) from beginning of the file. [ can you please tell current cursor position]
The position(index) of first character in files is zero just like string index.

**Eg:**

```
f=open("abc.txt","r")
print(f.tell())
print(f.read(10))
print(f.tell())
print(f.read(3))
print(f.tell())
```

**abc.txt**

```
naresh
shankar
dleep kumar
```

**output:**

```
0
naresh
sha
11
nka
14
```

**seek():**We can use seek() method to move cursor(file pointer) to specified location.
[Can you please seek the cursor to a particular location]
**Syntax:f.seek(offset, fromwhere)**

- **offset** represents the number of positions

The allowed values for second attribute(from where) are
0---->From beginning of file(default value)
1---->From current position
2--->From end of the file
Note: Python 2 supports all 3 values but Python 3 supports only zero.
Ex:

```
data="welcome to tkr college "
f=open("abc.txt","w")
f.write(data)
with open("abc.txt","r+") as f:
   text=f.read()
   print(text)
   print("The Current Cursor Position: ",f.tell())
   f.seek(23)
   print("The Current Cursor Position: ",f.tell())
   f.write("students")
   f.seek(0)
   text=f.read()
   print("Data After Modification:")
   print(text)
```

**output:**

```
welcome to tkr college
The Current Cursor Position:  23
The Current Cursor Position:  23
Data After Modification:
welcome to tkr college students
```

**How to check a particular file exists or not?**

- We can use os library to get information about files in our computer.
- os module has path sub module,which contains isFile() function to check whether a particular file exists or not?

<u>**Syntax:**</u> **os.path.isfile(fname)**

## Program to print the number of lines, words and characters present in the given file?

```
import os,sys
fname=input("Enter File Name: ")
if os.path.isfile(fname):
   print("File exists:",fname)
   f=open(fname,"r")
else:
   print("File does not exist:",fname)
   sys.exit(0)
lcount=wcount=ccount=0
for line in f:
   lcount=lcount+1
   ccount=ccount+len(line)
   words=line.split()
   wcount=wcount+len(words)
print("The number of Lines:",lcount)
print("The number of Words:",wcount)
print("The number of Characters:",ccount)
```

**abc.txt:**

```
naresh goud
shiva shankar goud
R dileep kumar
```

**Output:**

```
Enter File Name: abc.txt
File exists: abc.txt
The number of Lines: 3
The number of Words: 8
The number of Characters: 45
```

**Handling Binary Data:**

It is very common requirement to read or write binary data like images, video files, audio files etc.

**Program to read image file and write to a new image file?**

```
f1=open("rossum.jpg","rb")
f2=open("newpic.jpg","wb")
bytes=f1.read()
f2.write(bytes)
print("New Image is available with the name: newpic.jpg")
```

**Directory in Python:**

- A directory or folder is a collection of files and sub directories
- very common requirement to perform operations for directories like
  1. To know current working directory
  2. To create a new directory
  3. To remove an existing directory
  4. To rename a directory
  5. To list contents of the directory

To perform these operations,Python provides **inbuilt module os**,

**OS Module**

The python OS module provides functions used for interacting with the operating system and also gets related information about it. The OS comes under Python's standard utility modules. This module offers a portable way of using operating system dependent functionality.

- The python OS module lets us work with the files and directories.
- There are some functions in the OS module which are given below:

**To Know Current Working Directory:** getcwd()

```
import os
cwd=os.getcwd()
print("Current Working Directory:",cwd)
```

**To create a sub directory in the current working directory:**

```
import os
os.mkdir("mysub")
print("mysub directory created in cwd")
```

**To create a sub directory in mysub directory:** mkdir()

```
cwd
   |-mysub
         |-mysub2

import os
os.mkdir("mysub/mysub2")
print("mysub2 created inside mysub")
```

Note: Assume mysub already present in cwd.

**To create multiple directories like sub1 in that sub2 in that sub3:** makedirs()

```
import os
os.makedirs("sub1/sub2/sub3")
print("sub1 and in that sub2 and in that sub3 directories created")
```

**To remove a directory:** rmdir()

```
import os
os.rmdir("mysub/mysub2")
print("mysub2 directory deleted")
```

**To remove multiple directories in the path:** removedirs()

```
import os
os.removedirs("sub1/sub2/sub3")
print("All 3 directories sub1,sub2 and sub3 removed")
```

**To rename a directory:** rename()

```
import os
os.rename("mysub","newdir")
print("mysub directory renamed to newdir")
```

**To know contents of directory:**

os module provides **listdir()** to list out the contents of the specified directory. It won't display the contents of sub directory.

```
import os
print(os.listdir("."))
```

**output:** it displays the all types of files in list format

- ➢ The above program display contents of current working directory but not contents of sub directories.
- ➢ If we want the contents of a directory including sub directories then we should go for walk() function.

**To know contents of directory including sub directories:**

We have to **use walk()** function

**os.walk(path,topdown=True,onerror=None,followlinks=False)**

It returns an Iterator object whose contents can be displayed by using for loop

path-->Directory path. (cwd means **'.'**)
topdown=True --->Travel from top to bottom
onerror=None --->on error detected which function has to execute.
followlinks=True -->To visit directories pointed by symbolic links

**To display all contents of Current working directory including sub directories:**

```
import os
for dirpath,dirnames,filenames in os.walk('.'):
        print("Current Directory Path:",dirpath)
        print("Directories:",dirnames)
        print("Files:",filenames)
        print()
```

**output:**

```
Current Directory Path: .
Directories: ['com', 'newdir', 'pack1', '__pycache__']
Files: ['abc.txt', 'abcd.txt', 'demo.py', 'math.py', 'emp.csv', 'file1.txt'
 , 'file2.txt', 'file3.txt', 'files.zip', 'log.txt', 'module1.py', 'mylog.txt', '
 newpic.jpg', 'rossum.jpg', 'test.py']

 Current Directory Path: .\com
 Directories: ['naresh', '__pycache__']
 Files: ['module1.py', '__init__.py']
```

- ➢ To display contents of particular directory, we have to provide that directory name

as argument to walk() function.

**os.walk("directoryname")**

**What is the difference between listdir() and walk() functions?**

**listdir():** we will get contents of specified directory but not sub directory contents.
**walk()** : we will get contents of specified directory and its sub directories also**.**

**To get information about a File:**

We can get statistics of a file like size, last accessed time, last modified time etc by **using stat() function of os module**.

```
stats = os.stat("abc.txt")
```

The statistics of a file includes the following parameters:
st_mode==>Protection Bits
st_ino==>Inode number
st_dev===>device
st_nlink===>no of hard links
st_uid===>userid of owner
st_gid==>group id of owner
st_size===>size of file in bytes
st_atime==>Time of most recent access
st_mtime==>Time of Most recent modification
st_ctime==> Time of Most recent meta data change

**Note:**st_atime, st_mtime and st_ctime returns the time as number of milli seconds since Jan 1st 1970 ,12:00AM. By using datetime module fromtimestamp() function,we can get exact date and time.

**To print all statistics of file abc.txt:**

```
import os
stats=os.stat("abc.txt")
print(stats)
```

**To print specified properties:**

```
import os
from datetime import *
stats=os.stat("abc.txt")
print("File Size in Bytes:",stats.st_size)
```

## Zipping and Unzipping Files:

**What is Zip File?**

Zip is an archive file format which supports the lossless data compression. The Zip file is a single file containing one or more compressed files.

**Uses for Zip File**

- Zip files help you to put all related files in one place.
- Zip files help to reduce the data size.
- Zip files transfer faster than the individual file over many connections.

To perform zip and unzip operations, Python contains one **in-bulit module zipfile**.
This module contains a **class : ZipFile.**

**To create Zip file:**

- We have to create ZipFile class object with name of the zipfile, mode and constant ZIP_DEFLATED. This constant represents we are creating zip file.
  **f = ZipFile("files.zip","w","ZIP_DEFLATED")**
- Once we create ZipFile object,we can add files by using **write()** method.
  **f.write(filename)**

Eg:
```
from zipfile import *
f=ZipFile("files.zip",'w',ZIP_DEFLATED)
f.write("file1.txt")
f.write("file2.txt")
f.write("file3.txt")
f.close()
print("files.zip file created successfully")
```
**output:**
```
files.zip file created successfully
```

**<u>To perform unzip operation</u>**:

- We have to create ZipFile object as follows
  **f = ZipFile("files.zip","r",ZIP_STORED)**
- ZIP_STORED constant represents unzip operation. This is default value and hence we are not required to specify.
- Once we created ZipFile class object for unzip operation, we can get all file names present in that zip file by using namelist() method.
  **names = f.namelist()**

Eg:
```
from zipfile import *
f=ZipFile("files.zip",'r',ZIP_STORED)
names=f.namelist()
for name in names:
        print( "File Name: ",name)
        print("The Content of this file is:")
        f1=open(name,'r')
        print(f1.read())
        print()
```
**Output:**
```
File Name:  file1.txt
The Content of this file is:
abc
xyz
mnp

File Name:  file2.txt
The Content of this file is:
abcd
wxyz
mnop

File Name:  abc.txt
The Content of this file is:
naresh goud
shiva shankar goud
R dleep kumar
```

**Write a Python program to combine each line from first file with the corresponding line in second file.**

```
with open('abc.txt') as f1, open('file2.txt') as f2:
    for line1, line2 in zip(f1, f2):
        # line1 from abc.txt, line2 from file2.txt
        print(line1+line2)
```

output:

```
            naresh goud
            abcd
            shiva shankar goud
            wxyz
            R dleep kumar
            Mnop
```

**Write a Python program to assess if a file is closed or not**

```
            f = open('abc.txt','r')
            print(f.closed)
            f.close()
            print(f.closed)
```

# Modules

A python module can be defined as a python program file which contains a python code including python functions, class, or variables. In other words, we can say that our python code file saved with the extension (.py) is treated as the module. We may have a runnable code inside the python module.

Modules in Python provide us the flexibility to organize the code in a logical way.

To use the functionality of one module into another, we must have to import the specific module

**Example**

In this example, we will create a module named as file.py which contains a function func that contains a code to print some message on the console.

Let's create the module named as **calc.py.**

```
x=888
def add(a,b):
        print("The Sum:",a+b)
def product(a,b):
        print("The Product:",a*b)
```

calc module contains one variable i.e ‘x’ and 2 functions are add(), product()

If we want to use members of module in our program then we should import that module

**Loading the module in our python code**

We need to load the module in our python code to use its functionality. Python provides two types of statements as defined below.

1. import statement
2. from-import statement

**1.import statement**

The import statement is used to import all the functionality of one module into another. Here, we must notice that we can use the functionality of any python source file by importing that file as the module into another python source file.

We can import multiple modules with a single import statement, but a module is loaded once regardless of the number of times, it has been imported into our file.

The syntax to use the import statement is

**import module1,module2,........ module n**

**We can access members by using module name.**

```
modulename.variable
modulename.function()
```

Ex:

```
import calc
print(calc.x)
calc.add(10,20)
calc.product(10,20)
```

**Output**

```
888
The Sum: 30
The Product: 200
```

Note: whenever we are using a module in our program, for that module compiled file will be generated and stored in the hard disk permanently.

2. **from-import statement**

Instead of importing the whole module into the namespace, python provides the flexibility to import only the specific attributes of a module. This can be done by using from import statement. The syntax to use the from-import statement is given below.

**from** < module-name> **import** <name 1>, <name 2>..,<name n>

The main advantage of this is we can access members directly without using module name.

Ex:

```
from calc import x,add
print(x)
add(10,20)
product(10,20)==> NameError: name 'product' is not defined
```

We can import all members of a module as follows

from calc import *

**test.py:**

```
from calc import *
print(x)
add(10,20)
product(10,20)
```

**<u>Renaming a module at the time of import (module aliasing):</u>**

Python provides us the flexibility to import some module with a specific name so that we can use this name to use that module in our python source file.

The syntax to rename a module is given below.

**import <module-name> as <specific-name>**

**Ex:**

```
import calc c
print(c.x)
c.add(10,20)
c.product(10,20)
```

**<u>Various possibilties of import:</u>**

import modulename…….**import single module**
import module1,module2,module3…..**import multiple modules**
import module1 as m……..**import single module with rename**
import module1 as m1,module2 as m2,module3**…import multiple modules with multiple renames**
from module import member……………. **Import single module member**
from module import member1,member2,memebr3…**import multiple members from single module**
from module import memeber1 as x… **..import member as alias name from single module**
from module import *…… **import all members form module**

**<u>member aliasing:</u>**

```
from calc import x as y, add as sum
print(y)
sum(10,20)
```

Once we defined as alias name,we should use alias name only and we should not use original name

**Eg:**

```
from calc import x as y
print(x)==>NameError: name 'x' is not defined
```

**Finding members of module by using dir() function:**

Python provides inbuilt function dir() to list out all members of current module or a specified module.

**dir()** ===>To list out all members of current module
**dir(moduleName)**==>To list out all members of specified module

**calc.py.**

```
x=888
def add(a,b):
        print("The Sum:",a+b)
def product(a,b):
        print("The Product:",a*b)
```

**Ex1**:

```
import calc
print(dir())
```

**output:**

```
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__',
 '__package__', '__spec__', 'add',’calc’, 'product', 'x']
```

**Ex2:**

```
import calc
Print(dir(calc))
```

**Output:**

```
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__',
'__package__', '__spec__', 'add', 'product', 'x']
```

## math module:

- Python provides inbuilt module math.
- This module defines several functions which can be used for mathematical operations.
- The main important functions are

| **Function** | **Description** |
|---|---|
| **ceil(x)** | Returns the smallest integer **greater than or equal to x**. |
| **fabs(x)** | Returns the absolute value of x |
| **factorial(x)** | Returns the factorial of x |
| **floor(x)** | Returns the largest integer **less than or equal to x** |
| **fmod(x, y)** | Returns the remainder when x is divided by y |
| **frexp(x)** | Returns the mantissa and exponent of x as the pair (m, e) |
| **exp(x)** | Returns e**x |

| | |
|---|---|
| **log10(x)** | Returns the base-10 logarithm of x |
| **pow(x, y)** | Returns x raised to the power y |
| **sqrt(x)** | Returns the square root of x |
| **cos(x)** | Returns the cosine of x |
| **sin(x)** | Returns the sine of x |
| **tan(x)** | Returns the tangent of x |
| **degrees(x)** | Converts angle x from radians to degrees |
| **pi** | Mathematical constant, the ratio of circumference of a circle to it's diameter (3.14159...) |
| **e** | mathematical constant e (2.71828...) |

**Eg:**

```
from math import *
print(sqrt(4))……….2.0
print(int(sqrt(4)))…..2
print(ceil(10.5))…….11
print(floor(10.9))……10
print(fabs(-10.6))……10.6
print(fabs(10.6))…….10.6
print(pow(2,3))……..8.0
print(int(pow(2,3)))….8
print(log2(10))…… 3.321928094887362
print(log10(10))……1.0
print(pi)……….. 3.141592653589793
print(e)………....2.718281828459045
```

**Note:** We can find help for any module by using help() function

**Eg:**

```
import math
help(math)
```

## random module:

- This module defines several functions to generate random numbers.
- We can use these functions while developing games,in cryptography and to generate random numbers on fly for authentication

**1. random() function:**

This function always generate some float value between 0 and 1 ( not inclusive) i.e **0<x<1**

**Ex:**

```
from random import *
for i in range(5):
    print(random())
```

**output:**

```
0.8693926429398271
0.5721387451624846
```

```
0.232012787268973
0.7169434818780743
0.05520566642846614
```

**2. randint() function:**

To generate random integer beween two given numbers(inclusive**)**

**Ex:**

```
from random import *
for i in range(4):
    print(randint(1,100))
```

**output:**

```
40
74
50
71
```

**3. uniform():**

It returns random float values between 2 given numbers (not inclusive)

**Ex:**

```
from random import *
for i in range(4):
        print(uniform(1,100))
```

**output:**

```
43.47042788627279
39.45908588010893
8.467302075336807
87.69016431511457
```

**4. randrange([start],stop,[step])**

Returns a random number from range

start<= x < stop

start argument is optional and default value is 0

step argument is optional and default value is 1

randrange(10)-->generates a number from 0 to 9
randrange(1,15)-->generates a number from 1 to 14
randrange(1,11,2)-->generates a number from 1,3,5,7,9

**Ex1:**

```
from random import *
for i in range(4):
    print(randrange(1000,9999))
```

**output:**

```
4397
2429
8286
2122
```

**Ex2:**

```
from random import *
for i in range(4):
    print(randrange(1,10,3))
```

**output:**

```
1
4
1
7
```

**5. choice() function:**

- It won’t return random number.
- It will return a random object from the given list or tuple.

**Ex:**

```
from random import *
list=['a','abc',23,45,'xyz']
for i in range(6):
    print(choice(list),end=” “)
```

**output:** 45 xyz abc 23 abc a  .

**Reloading a Module:**

By default module will be loaded only once even though we are importing multiple times.

Ex**: module1.py:**

```
print("This is from module1")
```

**test.py**

```
import module1
import module1
import module1
import module1
print("This is test module")
```

**Output:**

```
This is from module1
This is test module
```

**Note**:In the above program test module will be loaded only once even though we are importing multiple times.

- The problem in this approach is after loading a module if it is updated outside then updated version of module1 is not available to our program.
- We can solve this problem by reloading module explicitly based on our requirement. We can reload by using reload() function of imp module.

**import imp**
**imp.reload(module_name)**
**or**
**from imp import reload**
**reload(module_name)**

**test.py:**

```
import module1
import module1
from imp import reload
reload(module1)
reload(module1)
reload(module1)
print("This is test module")
```

In the above program module1 will be loaded 4 times in that 1 time by default and 3 times explicitly. In this case output is

```
This is from module1
This is from module1
This is from module1
This is from module1
This is test module
```

The main advantage of explicit module reloading is we can ensure that updated version is always available to our program.

**The Special variable __name__:**

- For every Python program , a special variable __name__ will be added internally.
- This variable stores information regarding whether the program is executed as an individual program or as a module.
- If the program executed as an individual program then the value of this variable is __main__
- If the program executed as a module from some other program then the value of this variable is the name of module where it is defined.
- Hence by using this __name__ variable we can identify whether the program executed directly or as a module

Ex**: module1.py**

```
def f1():
    if __name__=='__main__':
        print("The code executed as a program")
    else:
        print("The code executed as a module from some other program")
f1()
```

**test.py**

```
import module1
module1.f1()
```

**output: py module1.py**

```
The code executed as a program
```

**py test.py**

```
The code executed as a module from some other program
The code executed as a module from some other program
```

**<u>Datetime module:</u>**

**date** – Manipulate just date ( year, month, day)
**time** – Time independent of the day (Hour, minute, second, microsecond, tzinfo)
**datetime** – Combination of time and date (year, month, day, hour, minute, second, microsecond, tzinfo)

**Get Current Date and Time**

```
import datetime
datetime_object = datetime.datetime.now()
print(datetime_object)
```

**output:** 2020-03-03 12:22:31.383936

- Here, we have imported **datetime** module using **import datetime** statement.
- **now()** method is used to create a datetime object containing the current local date and time.

**Get Current Date**

```
import datetime
date_object = datetime.date.today()
print(date_object)
```

**output:** 2020-03-03

- In above program, we have used **today()** method defined in the **date class** to get a **date object** containing the current local date.

**Date object to represent a date**

```
import datetime
d = datetime.date(2020, 4, 13)
print(d)
```

**output:** 2020-04-13

We can **only import date class** from the datetime module

```
from datetime import date
a = date(2020, 4, 13)
print(a)
```

**output**: 2020-04-13

**Print today's year, month and day**

```
from datetime import date
# date object of today's date
today = date.today()
print("Current year:", today.year)
print("Current month:", today.month)
print("Current day:", today.day)
```

**output**:
```
Current year: 2020
Current month: 3
Current day: 3
```

**Print hour, minute, second and microsecond**

```
from datetime import time
a = time(11, 34, 56)
print("hour =", a.hour)
print("minute =", a.minute)
print("second =", a.second)
print("microsecond =", a.microsecond)
```

**output:**
```
hour = 11
minute = 34
second = 56
microsecond = 0
```

**Python program to calculate number of days between two dates.**

```
from datetime import date
f_date = date(1991, 6, 25)
l_date = date(2020, 3, 3)
diff = l_date - f_date
print(diff)………………… 10479 days, 0:00:00
print(diff.days)…………… 10479
```

**or**

```
import datetime
x = input("Enter first date in Y,m,d format:")
y = input("Enter second date in Y,m,d format:")
a = datetime.datetime.strptime(x,"%Y,%m,%d")
b = datetime.datetime.strptime(y,"%Y,%m,%d")
print((a-b).days)
```

**output**:
```
Enter first date in Y,m,d format:2020,03,03
Enter second date in Y,m,d format:1991,06,25
10479
```

**Write a Python program to print the calendar of a given month and year.**

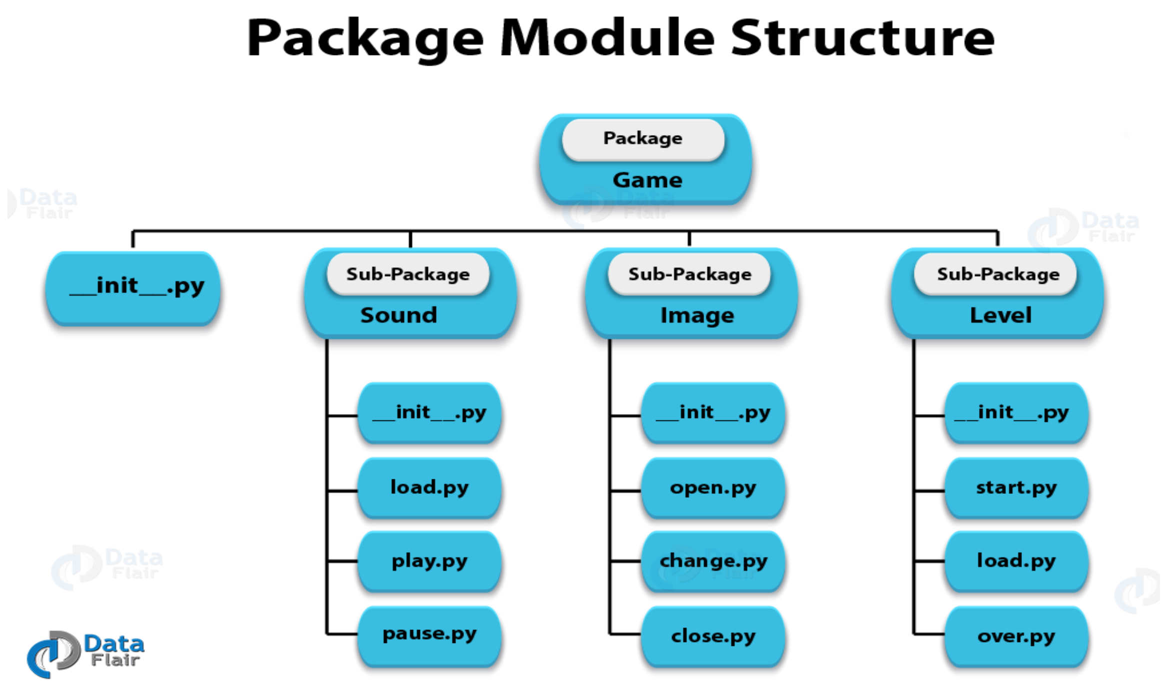
```
import calendar
y = int(input("Input the year : "))
m = int(input("Input the month : "))
print(calendar.month(y, m))
```

**output:**

```
Input the year : 2020
Input the month : 03
     March 2020
Mo Tu We Th Fr Sa Su
                   1
 2  3  4  5  6  7  8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30 31
```

**packages :**

- package is nothing but folder or directory which represents collection of Python modules.
- Any folder or directory contains __init__.py file, is considered as a Python package. This __init__.py file can be empty.
- A package can contains sub packages also.
- Packages are a way of structuring Python's module namespace by using "dotted module names". A.B stands for a submodule named B in a package named A. Two different packages like P1 and P2 can both have modules with the same name, let's say A, for example. The submodule A of the package P1 and the submodule A of the package P2 can be totally different.
- A package is imported like a "normal" module



**How to create package:**

1. Create directory
   Ex: import os
   os.mkdir("mysub")
   run above program its create a directory or folder with name "mysub"
2. Add modules to above directory

**__init__.py**

**module1.py**

```
def func1():
    print("package-----module1.....function1")
def func2():
    print("package-----module1.....function2")
def func3():
    print("package-----module1.....function3")
```

**module2.py**

```
def func1():
    print("package-----module2.....function1")
def func2():
    print("package-----module2.....function2")
def func3():
    print("package-----module2.....function3")
```

**How to access package members in another python program i.e saved in same drive.**

- Same as module access we can access the package
  **Syntax** for importing package module
  **import package_name.module_name**

**<u>Ex: test.py</u>**

```
import mysub.module1 as m1, mysub.module2 as m2
m1.func1()
m1.func2()
m2.func1()
m2.func2()
```

**output:**

```
package-----module1.....function1
package-----module1.....function2
package-----module2.....function1
package-----module2.....function2
```

**How to access package members in another python program i.e saved in another drive.**

1. Import predefined **sys** module in a new python program i.e
   **import sys**
2. Append user defind packge path to system path by using following syntax
   **sys.path.append("user defined package path")**
3. Finally import the user defined package modules

**Ex: test.py**

```
import sys
sys.path.append("C:/Users/hi/Desktop/mysub")
import module1, module2
module1.func1()
module2.func1()
module1.func2()
module2.func2()
```

**output:**

```
package-----module1.....function1
package-----module2.....function1
package-----module1.....function2
package-----module2.....function2
```

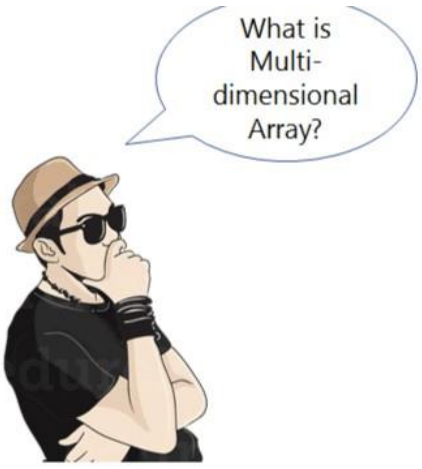
**What is a Python NumPy?**

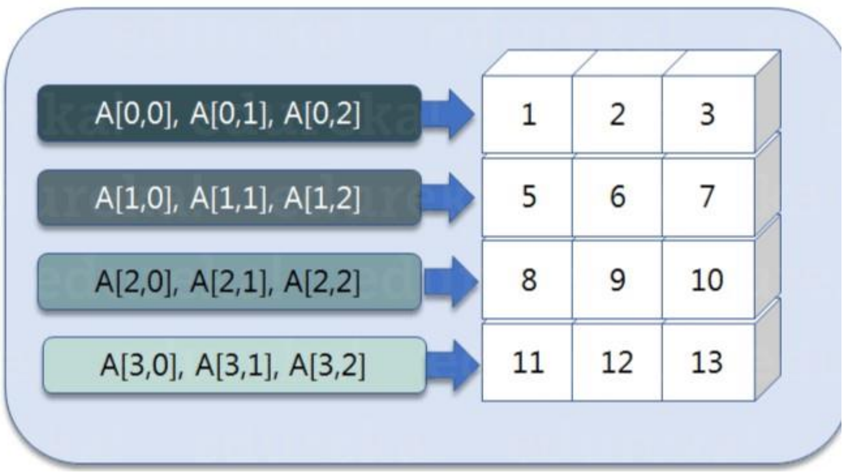
NumPy is a Python package which stands for 'Numerical Python'. It is the core library for scientific computing, which contains a powerful n-dimensional array objects, provide tools for integrating C, C++ etc. It is also useful in linear algebra, random number capability etc. NumPy array can also be used as an efficient multi-dimensional container for generic data.

**NumPy Array:** Numpy array is a powerful N-dimensional array object which is in the form of rows and columns. We can initialize numpy arrays from nested Python lists and access it elements. Numpy arrays essentially come in two flavors: vectors and matrices. Vectors are strictly 1-d arrays and matrices are 2-d (but you should note a matrix can still have only one row or one column).

**How do I install NumPy?**

To install Python NumPy, go to your command prompt and type "pip install numpy". Once the installation is completed, go to your IDE and simply import it by typing: "**import numpy as np**"





Here, I have different elements that are stored in their respective memory locations. It is said to be two dimensional because it has rows as well as columns. In the above image, we have 3 columns and 4 rows available.

**How to create NumPy arrays :**
**Syntax:** numpy.array(object, dtype = None, copy = True, order = None, subok = False, ndmin = 0)

| S.no | |
|---|---|
| 1 | **object**<br>Any object exposing the array interface method returns an array, or any (nested) sequence. |
| 2 | **dtype**<br>Desired data type of array, optional |
| 3 | **copy**<br>Optional. By default (true), the object is copied |
| 4 | **order**<br>C (row major) or F (column major) or A (any) (default) |
| 5 | **subok**<br>By default, returned array forced to be a base class array. If true, sub-classes passed through |
| 6 | **ndmin**<br>Specifies minimum dimensions of resultant array |

Let us see how it is implemented:

**Single-dimensional Numpy Array:**

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

**output:** [1 2 3]

**Multi-dimensional Array:**

```
import numpy as np
a=np.array([(1,2,3),(4,5,6)])
print(a)
```

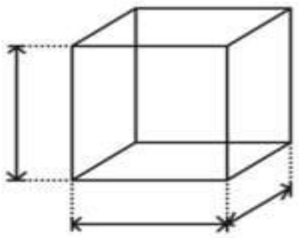
**output**:

```
[[1 2 3]
  [4 5 6]]
```

**Python NumPy Operations**

**ndim:**

- To **find the dimension of the array**, whether it is a two-dimensional array or a single dimensional array use ndim function..



Ex:

```
import numpy as np
a = np.array([(1,2,3),(4,5,6)])
print(a.ndim)
```

output: 2

**itemsize:**

- To **calculate the byte size of each element** in array use itemsize(). In the below code, I have defined a single dimensional array and with the help of 'itemsize' function, we can find the size of each element.

Ex:

```
import numpy as np
a = np.array([(1,2,3)])
print(a.itemsize)
```

Output: 4

So every element occupies 4 byte in the above numpy array.

**dtype:**

- if you want **to know the data type of a particular element**, you can use 'dtype' function which will **print the datatype along with the size**.

Ex:

```
import numpy as np
a = np.array([(1,2,3)])
print(a.dtype)
```

output: int32

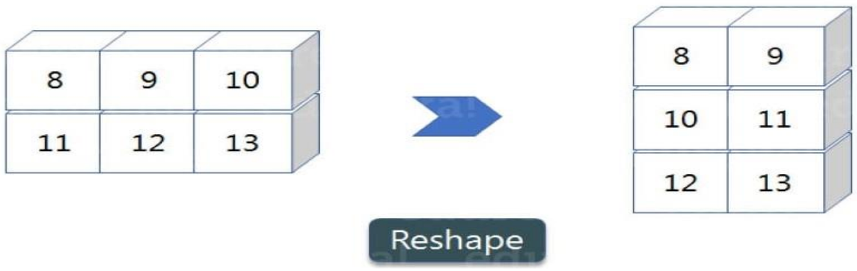
As you can see, the data type of the array is integer 32 bits. Similarly, you can find the size and shape of the array using 'size' and 'shape' function respectively.

**size:** returns the no. of elements present in an array
**shape:** it represents the shape of array i.e **no. of rows and no. of columns**

**Ex1**:
```
import numpy as np
a = np.array([(1,2,3,4,5,6)])
print(a.size)
print(a.shape)
```
**output**:
```
6
(1, 6)
```
**Ex2**:
```
import numpy as np
a = np.array([(1,2,3),(4,5,6)])
print(a.size)
print(a.shape)
```
**output:**
```
6
(2, 3)
```

**reshape:**

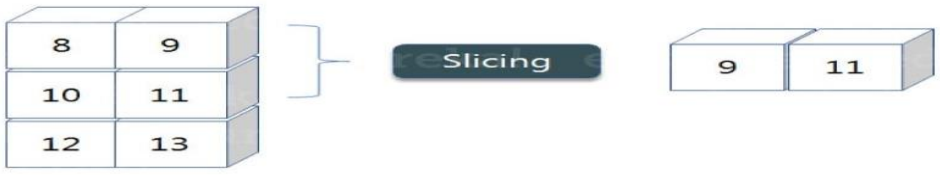
- Reshape is when you **change the number of rows and columns** which gives a new view to an object. Now, let us take an example to reshape the below array:



- in the above image, we have 3 columns and 2 rows which has converted into 2 columns and 3 rows.

**Ex:**
```
import numpy as np
a = np.array([(8,9,10),(11,12,13)])
print("original array is\n",a)
a=a.reshape(3,2)
print("after converting\n",a)
```
**output:**
```
original array  is
 [[ 8  9 10]
 [11 12 13]]
after converting
 [[ 8  9]
 [10 11]
 [12 13]]
```
**Slicing:**

- Slicing is basically **extracting particular set of elements from an array**. This slicing operation is pretty much similar to the one which is there in the list as well. Consider the following example:



**Syntax for 1d-array:**

Array_name[start:end:step]

**Synatax for 2d-array:**

Array_name[start:end:step, start:end: step]

(Row) (col)

Ex:
```
import numpy as np
a=np.array([(1,2,3,4),(3,4,5,6)])
print(a)
print(a[0:,0::2])
```
**output:**
```
[[1 2 3 4]
 [3 4 5 6]]

[[1 3]
 [3 5]]
```

**min(): returns the minimum element from array**

**max():returns the maximum element from array**

**sum():returns the sum off all elements present in an array**

**Ex:**
```
import numpy as np
a= np.array([1,2,3])
print(a.min())……………1
print(a.max())……………3
print(a.sum())……………6
```
**above min,max,sum functions works only for one dimensional array.**

**For two dimensional array use axis object**
**i.e for columns sum use arrayname.sum(axis=0)**
**for rows sum use arrayname.sum(axis=1)**

**axis 0**

|  |  |  |  |
|---|---|---|---|
| **axis 1** | **1** | **4** | **3** |
|  | **0** | **2** | **5** |

**ex:**
```
import numpy as np
a= np.array([(1,4,3),(0,2,5)])
print(a)
print(a.sum(axis=0))………… [1 6 8]
print(a.sum(axis=1))…………. [8 7]
print(a.min(axis=0))………….. [0 2 3]
print(a.min(axis=1))…………. [1 0]
print(a.max(axis=0))………….. [1 4 5]
print(a.max(axis=1))………….. [4 5]
```

**<u>Square Root & Standard Deviation:</u>**

**sqrt():**it returns array with Square Root of all element present in an array
**syntax**:numpy.sqrt(array_name)

**Standard Deviation** :is a measure of the amount of variation or dispersion of a set of values. In python using std() function we can find the Standard Deviation of the given array values
**Syntax:** numpy.std(array_name)

**Ex:**
```
import numpy as np
a=np.array([(1,2,9),(16,4,1,)])
print(np.sqrt(a))
print(“Standard Deviation is:”,np.std(a))
```
output:
```
[[1.   1.41421356      3.  ]
  [4.        2.               1. ]]
```

```
Standard Deviation is :5.439056290693573
```

**Note:** we can also perform addition ,subtraction, multiplication and division operations on arrays
**Ex: for addition/substraction/division/multiplication**

```
import numpy as np
x= np.array([(1,2,3),(3,4,5)])
y= np.array([(1,2,3),(3,4,5)])
print("substraction of two arrays is:\n",x-y)
print("addtion of two arrays is:\n",x*y)
print("division of two arrays  is :\n",x/y)
print("multiplication of two arrays is:\n",x*y)
```

**output:**

```
substraction of two arrays is:
 [[0 0 0]
 [0 0 0]]
addtion of two arrays is:
 [[ 1  4  9]
 [ 9 16 25]]
division of two arrays is :
 [[1. 1. 1.]
 [1. 1. 1.]]
multiplication of two arrays is:
 [[ 1  4  9]
 [ 9 16 25]]
```

**Note**: ***** is used for array multiplication (multiplication of corresponding elements of two arrays) not matrix multiplication**.** To multiply two matrices, we **use dot()** method
**Ex:**

```
import numpy as np
x= np.array([(1,2,3),(3,4,5)])
y= np.array([(1,2),(3,4),(5,6)])
print(x.dot(y))
```

**output:**

```
[[22   28]
[40   52]]
```

**Vertical & Horizontal Stacking**

if you want to **concatenate two arrays and not just add them**, you can perform it using two ways – vertical stacking and horizontal stacking.
**Ex:**

```
import numpy as np
x= np.array([(1,2,3),(3,4,5)])
y= np.array([(7,8,9),(4,5,6)])
print("vertical stacking of two matrices is:\n",np.vstack((x,y)))
print("horizontal stacking of two matrices is:\n",np.hstack((x,y)))
```

**output:**

```
vertical stacking of two matrices is:
 [[1 2 3]
 [3 4 5]
 [7 8 9]
 [4 5 6]]
horizontal stacking of two matrices is:
 [[1 2 3 7 8 9]
 [3 4 5 4 5 6]]
```

**ravel():**to convert one numpy **array into a single column values**

**ex:**
```
import numpy as np
x= np.array([(1,2,3),(3,4,5)])
print(x.ravel()
```

**output:** [1 2 3 3 4 5]

**Built-in Methods**

**arange():**Return evenly spaced values within a given interval**.**

**Syntax:** arrange(start,stop,step, dtype=None)

**Ex1:**
```
import numpy as np
np.arange(3)……………. array([0, 1, 2])
```
**Ex2:**
```
import numpy as np
np.arange(4.0)……………. array([0., 1., 2., 3.])
```
Ex3:
```
import numpy as np
np.arange(3,7)……..…….. array([3, 4, 5, 6])
```
Ex4:
```
import numpy as np
np.arange(3,10,2)……….. array([3, 5, 7, 9])
```
Ex5:
```
import numpy as np
np.arange(3,10,2,dtype="complex")……………array([3.+0.j, 5.+0.j, 7.+0.j, 9.+0.j])
```

**zeros():**Return a new array of given shape and type, filled with zeros

**Syntax:** zeros(shape, dtype=float, order='C')

Ex1:
```
import numpy as np
np.zeros(5)………………. array([0., 0., 0., 0., 0.])
```
Ex2:
```
import numpy as np
np.zeros((5,), dtype=int)………… rray([0, 0, 0, 0, 0])
```
Ex3:
```
import numpy as np
np.zeros((2, 1))
```
output:
```
array([[0.],
       [0.]]
```
Ex4:
```
import numpy as np
s = (2,2)
np.zeros(s)
```
output:
```
array([[0.],
        [0.]])
```
Ex5:
```
import numpy as np
Print(np.zeros((2, 1)))
```
Output:
```
[[0.],
  [0.]]
```

**Ones():**Return a new array of given shape and type, filled with ones, and default datatype is float

Syntax:ones(shape, dtype=None, order='C')

Examples
```
  >>> np.ones(5)
  array([ 1.,  1.,  1.,  1.,  1.])
  >>> np.ones((5,), dtype=int)
 array([1, 1, 1, 1, 1])
  >>> np.ones((2, 1))
  array([[ 1.],
        [ 1.]])
.  >>> s = (2,2)
   >>> np.ones(s)
 array([[ 1.,  1.],
                [ 1.,  1.]])
```
**Linspace():**Return evenly spaced numbers over a specified interval.

**Syntax:**linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None, axis=0)

**Examples**
```
>>> np.linspace(2.0, 3.0, num=5)
array([ 2.  ,  2.25,  2.5 ,  2.75,  3.  ])
>>> np.linspace(2.0, 3.0, num=5, endpoint=False)
array([ 2. ,  2.2,  2.4,  2.6,  2.8])
>>> np.linspace(2.0, 3.0, num=5, retstep=True)
(array([ 2.  ,  2.25,  2.5 ,  2.75,  3.  ]), 0.25)
>>> np.linspace(1,100,num=4,dtype=int)
array([  1,  34,  67, 100])
```

**random():it returns Random Number Generation array**

**Utility functions of random function**

**rand():**creates array of specifid shape and fills it with random values as per uniform distribution
```
Ex:      print(np.random.rand(2,2))
Output: [[0.08516095 0.38017884]
          [0.64591064 0.10340191]]
```
**randn():**creates array of specifid shape and fills it with random values as per standard normal distribution
```
Ex:       print(np.random.randn(2,2))
Output: [[ 1.13837121 -0.9422925 ]
          [ 0.85272927  0.18278011]]
```
**randint():**crates array of specified shape and fills it with random integers from low(inclusive) to high(exclusive). If high is not mentioned then the interval will be [0,low]

Syntax: randint(low, high=None, size=None, dtype='l')
```
ex:      print(np.random.randint(1, size=5))…………..  [0 0 0 0 0]
         print(np.random.randint(3, size=5))…………… [2 0 2 1 2]
         print(np.random.randint(1,5, size=(2, 4)))
                        [[4 4 3 2]
                         [1 2 1 3]]
         print(np.random.randint(4,5, size=(2, 3)))
                        [[4 4 4]
                        [4 4 4]]
```
**eye():**Creates an identity matrix.
```
Ex:      import numpy as np
         print(np.eye(2,2))
                 [[1. 0.]
                  [0. 1.]]
         print(np.eye(3,3))
                 [[1. 0. 0.]
                  [0. 1. 0.]
                  [0. 0. 1.]]
         print(np.eye(2,4))
                 [[1. 0. 0. 0.]
                  [0. 1. 0. 0.]]
```
**concatenate():**Join a sequence of arrays along an existing axis.

**Synatx:** concatenate((a1, a2, ...), axis=0, out=None)

a1, a2, ... : sequence of arrays: The arrays must have the same shape, except in the dimension corresponding to `axis`**.**

**Ex:1**
```
a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6]])
np.concatenate((a, b), axis=0)
```
```
output:  array([[1, 2],
        [3, 4],
        [5, 6]])
```

**Ex2:**
```
a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6]])
np.concatenate((a, b), axis=1)
```
output: ValueError: all the input array dimensions except for the concatenation axis must match exactly

**Ex3:**
```
a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6]])
np.concatenate((a, b.T), axis=1)
```
output:
```
array([[1, 2, 5],
       [3, 4, 6]])
```

**<u>Matrix creation using matrix class</u>**

A matrix is a two-dimensional data structure where numbers are arranged into rows and columns.

**Syntax:** numpy.matrix(data, dtype=None, copy=True)

- Returns a matrix from an array-like object, or from a string of data

**Ex1:**
```
import numpy as np
a = np.matrix('1 2; 3 4')
print(a)
```
**output:**
```
[[1 2]
 [3 4]]
```
**Ex2:**
```
import numpy as np
a = np.matrix([[1, 2], [3, 4]])
print(a)
```
**output:**
```
[[1 2]
 [3 4]]
```

**Matrix addition(+),substraction(-), multiplication(*)**

**Ex:**
```
import numpy as np
x= np.matrix([(1,2,3),(3,4,5),(4,5,6)])
y= np.matrix([(1,2,3),(3,4,6),(5,6,7)])
print("addition of two matrices is:\n",x+y)
print("substraction of two matrices is:\n",x-y)
print("multiplication  of two matrices is:\n",x*y)
```
**output:**
```
addition of two matrices is:
 [[ 2  4  6]
 [ 6  8 11]
 [ 9 11 13]]
substraction of two matrices is:
 [[ 0  0  0]
 [ 0  0 -1]
 [-1 -1 -1]]
multiplication  of two matrices is:
 [[22 28 36]
 [40 52 68]
 [49 64 84]]
```

**Matrix transpose() –**

The transpose of a matrix is obtained by moving the rows data to the column and columns data to the rows. If we have an array of shape (X, Y) then the transpose of the array will have the shape (Y, X)

**Using numpy array:**

```
import numpy as np
a = np.array([[1, 2, 3], [4, 5, 6]])
print("Original Array:\n",a)
a_t = a.transpose()
print("Transposed Array:\n",a_t)
```

**output:**

```
Original Array:
 [[1 2 3]
 [4 5 6]]
Transposed Array:
 [[1 4]
 [2 5]
 [3 6]]
```

**Using numpy matrix:**

```
import numpy as np
a = np.matrix([[1, 2, 3], [4, 5, 6]])
print("Original matrix:\n",a)
print("Transposed matrix:\n",a.T)
```

**output**:

```
Original matrix:
 [[1 2 3]
 [4 5 6]]
Transposed matrix:
 [[1 4]
 [2 5]
 [3 6]]
```

**Inverse of a matrix:**

We use numpy. linalg. inv() function to calculate the inverse of a matrix. The inverse of a matrix is such that if it is multiplied by the original matrix, it results in identity matrix

```
import numpy as np
m = np.array([[1,2],[3,4]])
print("Original matrix:")
print(m)
result =  np.linalg.inv(m)
print("Inverse of a given matrix:")
print(result)
```

**output:**

```
Original matrix:
[[1 2]
 [3 4]]
Inverse of the said matrix:
[[-2.   1. ]
 [ 1.5 -0.5]]
```

# SCIPY

## What is SciPy?

SciPy is an Open Source Python-based library, which is used in mathematics, scientific computing, Engineering, and technical computing.

SciPy also pronounced as "Sigh Pi."

The SciPy library depends on NumPy, which provides convenient and fast N-dimensional array manipulation. The SciPy library is built to work with NumPy arrays and provides many user-friendly and efficient numerical practices such as routines for numerical integration and optimization. Together, they run on all popular operating systems, are quick to install and are free of charge. NumPy and SciPy are easy to use, but powerful enough to depend on by some of the world's leading scientists and engineers

## Why use SciPy

SciPy contains varieties of sub packages which help to solve the most common issue related to Scientific Computation.

SciPy is the most used Scientific library only second to GNU Scientific Library for C/C++ or Matlab's.

Easy to use and understand as well as fast computational power.

It can operate on an array of NumPy library.

## Numpy VS SciPy

**Numpy:**

- Numpy is written in C and use for mathematical or numeric calculation.
- It is faster than other Python Libraries
- Numpy is the most useful library for Data Science to perform basic calculations.
- Numpy contains nothing but array data type which performs the most basic operation like sorting, shaping, indexing, etc.

**SciPy:**

- SciPy is built in top of the NumPy
- SciPy is a fully-featured version of Linear Algebra while Numpy contains only a few features.
- Most new Data Science features are available in Scipy rather than Numpy.

Before start to learning SciPy, you need to know basic functionality as well as different types of an array of NumPy

**The standard way of import SciPy modules and Numpy:**

```
#same for other modules
        from scipy import special
        import numpy as np
```

## Sub-packages of SciPy:

## Linear Algebra:

Linear algebra deals with linear equations and their representations using vector spaces and matrices. SciPy is built on ATLAS LAPACK and BLAS libraries and is extremely fast in solving problems related to linear algebra. In addition to all the functions from numpy.linalg, scipy.linalg also provides a number of other advanced functions. Also, if numpy.linalg is not used along with ATLAS LAPACK and BLAS support, scipy.linalg is faster than numpy.linalg.

## Finding the Inverse of a Matrix:

Mathematically, the inverse of a matrix A is the matrix B such that AB=I where I is the identity matrix consisting of ones down the main diagonal denoted as B=A-1. In SciPy, this inverse can be obtained using the **linalg.inv** method

**Example:**

```
import numpy as np
from scipy import linalg
A = np.array([[1,2], [4,3]])
B = linalg.inv(A)
print(B)
```

**output:**

```
[[-0.6  0.4]
 [ 0.8 -0.2]]
```

**Finding the Determinants:**

The value derived arithmetically from the coefficients of the matrix is known as the determinant of a square matrix. In SciPy, this can be done using a function **det** which has the following syntax:

**SYNTAX:** det(a)

Where a : (M, M) Is a square matrix

**Example:**

```
import numpy as np
from scipy import linalg
A = np.array([[1,2], [4,3]])
B = linalg.det(A)
print(B)
```

**output: -5.0**

**Eigenvalues**

Eigenvalues are a specific set of scalars linked with linear equations. The most common problem in linear algebra is eigenvalues and eigenvector which can be easily solved **using eig() function**.

**Example:**

```
from scipy import linalg
import numpy as np
#define two dimensional array
arr = np.array([[5,4],[6,3]])
#pass value into function
eg_val, eg_vect = linalg.eig(arr)
#get eigenvalues
print("eigenvalues are\n",eg_val)
#get eigenvectors
print("eigenvectors are\n",eg_vect)
```

**output:**

```
eigenvalues are
 [ 9.+0.j -1.+0.j]
eigenvectors are
 [[ 0.70710678   -0.5547002 ]
 [ 0.70710678    0.83205029]]
```

**Sub-packages of SciPy:**

**Special Function package**

- **scipy.special** package contains numerous functions of mathematical physics.
- SciPy special function includes Cubic Root, Exponential, Log sum Exponential, Lambert, Permutation and Combinations, Gamma, Bessel, hypergeometric, Kelvin, beta, parabolic cylinder, Relative Error Exponential, etc**.**

  **Cubic Root Function:** Cubic Root function finds the cube root of values.

  **Syntax:** scipy.special.cbrt(x)

  **Example1:**

  ```
  from scipy.special import cbrt
  #Find cubic root of 125 & 64 using cbrt() function
  cb = cbrt([125, 64])
  #print value of cb
  print(cb)
  ```

  **output:  [5. 4.]**

**Exponential and Trigonometric Functions:**

SciPy's Special Function package provides a number of functions through which you can find exponents and solve trigonometric problems.

**Syntax for exponential function:** special.expn(m)….(n power m== n**m)

**Syntax for Trigonometric function: special.sindg(degree/value)**

**Example:**

```
from scipy import special
a = special.exp10(2)
print(a)
b = special.exp2(4)
print(b)
c = special.sindg(30)
print(c)
d = special.cosdg(60)
print(d)
```

**output:**

```
100.0
16.0
0.49999999999999994
0.49999999999999994
```

**Log Sum Exponential Function:**

Log Sum Exponential computes the log of sum exponential input element.

**Syntax :**scipy.special.logsumexp(x)

**Example:**

```
from scipy.special import logsumexp
a = np.arange(10)
scipy.special.logsumexp(a)
```

**output:** 9.45862974442671

**using numy**

```
import numpy as np
a = np.arange(10)
np.log(np.sum(np.exp(a)))
```

**output:** 9.45862974442671

**Permutations & Combinations**:

SciPy also gives functionality to calculate Permutations and Combinations.

**Synatax for Combinations -** scipy.special.comb(N,k)

**Example:**

```
from scipy.special import comb
#find combinations of 5, 2 values using comb(N, k)
com = comb(5, 2, exact = False)
print(com)
```

**output: 10.0**

**Note:** if exact=True it returns int value, else (exact=False) returns float value

**Syntax for Permutations :** scipy.special.perm(N,k)

**Example:**

```
from scipy.special import perm
#find permutation of 5, 2 using perm (N, k) function
per = perm(5, 2, exact = True)
print(per)
```

**output:** 20

# MATPLOTLIB

## What is Matplotlib

**Matplotlib** is a Python library which is defined as a multi-platform data visualization library built on Numpy array. It can be used in python scripts, shell, web application, and other graphical user interface toolkit.

The **John D. Hunter** originally conceived the matplotlib in **2002**. It has an active development community and is distributed under a **BSD-style license**. Its first version was released in 2003, and the latest **version 3.1.1** is released on **1 July 2019**.

There are various toolkits available that are used to enhance the functionality of the matplotlib. Some of these tools are downloaded separately, others can be shifted with the matplotlib source code but have external dependencies.

- **Bashmap**: It is a map plotting toolkit with several map projections, coastlines, and political boundaries.
- **Cartopy:** It is a mapping library consisting of object-oriented map projection definitions, and arbitrary point, line, polygon, and image transformation abilities.
- **Excel tools**: Matplotlib provides the facility to utilities for exchanging data with Microsoft Excel.
- **Mplot3d:** It is used for 3D plots.
- **Natgrid:** It is an interface to the Natgrid library for irregular gridding of the spaced data

**General Concepts:** A Matplotlib figure can be categorized into several parts as below:

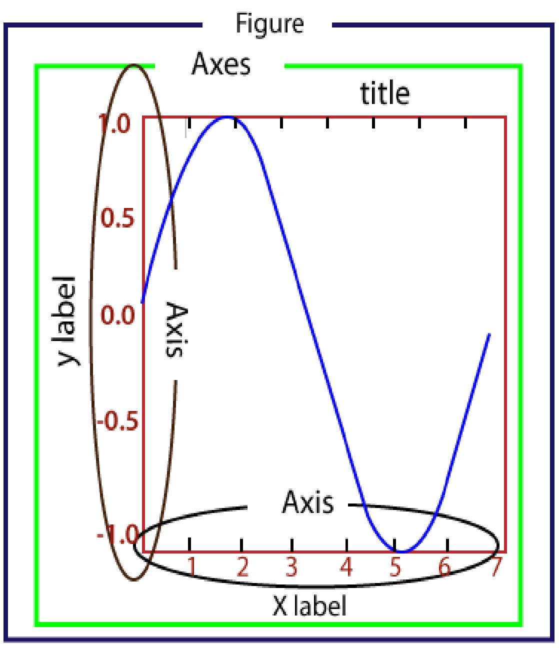


**Figure:** It is a whole figure which may contain one or more than one axes (plots). You can think of a Figure as a canvas which contains plots.

**Axes:** It is what we generally think of as a plot. A Figure can contain many Axes. It contains two or three (in the case of 3D) Axis objects. Each Axes has a title, an x-label and a y-label.

**Axis:** They are the number line like objects and take care of generating the graph limits.

**Artist:** Everything which one can see on the figure is an artist like Text objects, Line2D objects, collection objects. Most Artists are tied to Axes.

## Types of Plots

There are various plots which can be created using python matplotlib. Some of them are listed below:

## Getting Started with pyplot

Pyplot is a module of Matplotlib which provides simple functions to add plot elements like lines, images, text, etc. to the current axes in the current figure.

**# Generating a simple graph**

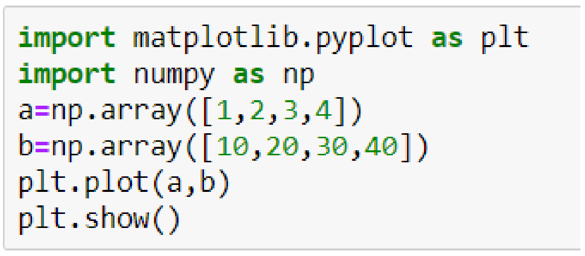
import matplotlib.pyplot as plt  (or )

from matplotlib import pyplot as plt

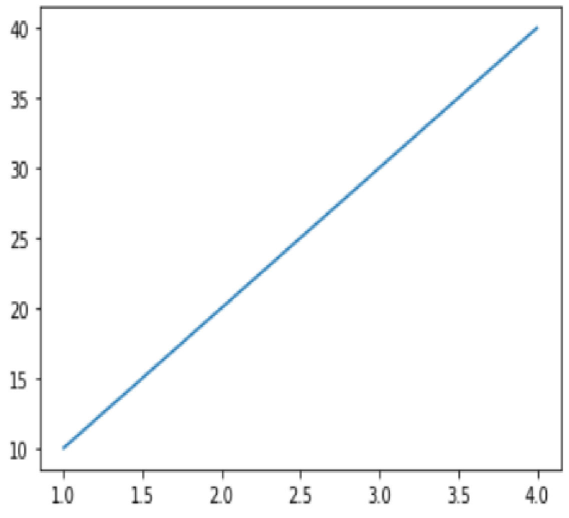
import numpy as np

In this snippet, we imported the pyplot module of the matplotlib library under the alias plt. An alias is a kind of short form of something. As we have to use pyplot in our coding a lot, we will use our plt alias instead.

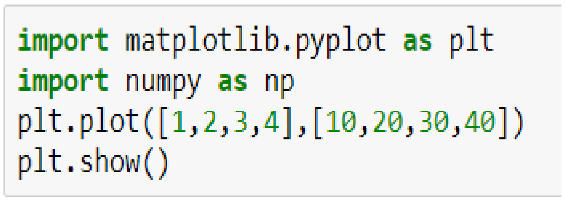
Here we import Matplotlib’s Pyplot module and Numpy library as most of the data that we will be working with will be in the form of arrays only.



```
import matplotlib.pyplot as plt
import numpy as np
a=np.array([1,2,3,4])
b=np.array([10,20,30,40])
plt.plot(a,b)
plt.show()
```



**OR**

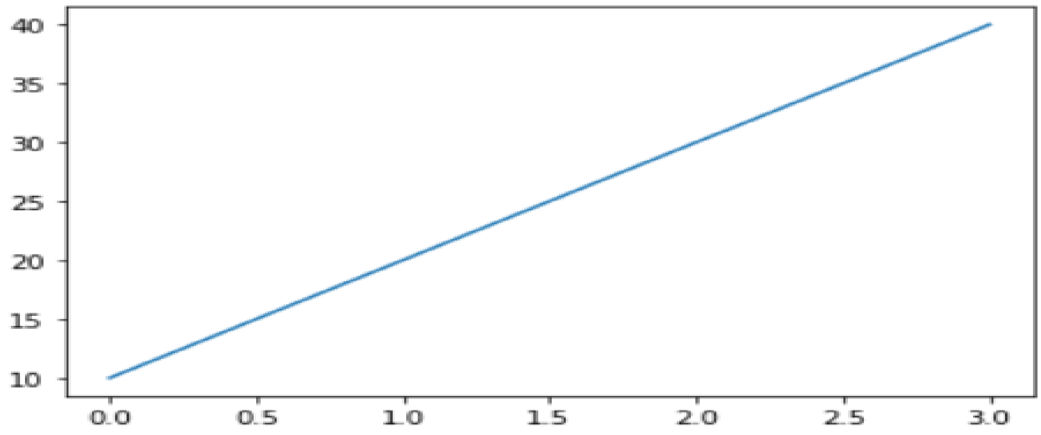


```
import matplotlib.pyplot as plt
import numpy as np
plt.plot([1,2,3,4],[10,20,30,40])
plt.show()
```

We pass two arrays as our input arguments to pyplot's plot() method and use show() method to invoke the required plot. Here note that the first array appears on the x-axis and second array appears on the y-axis of the plot.

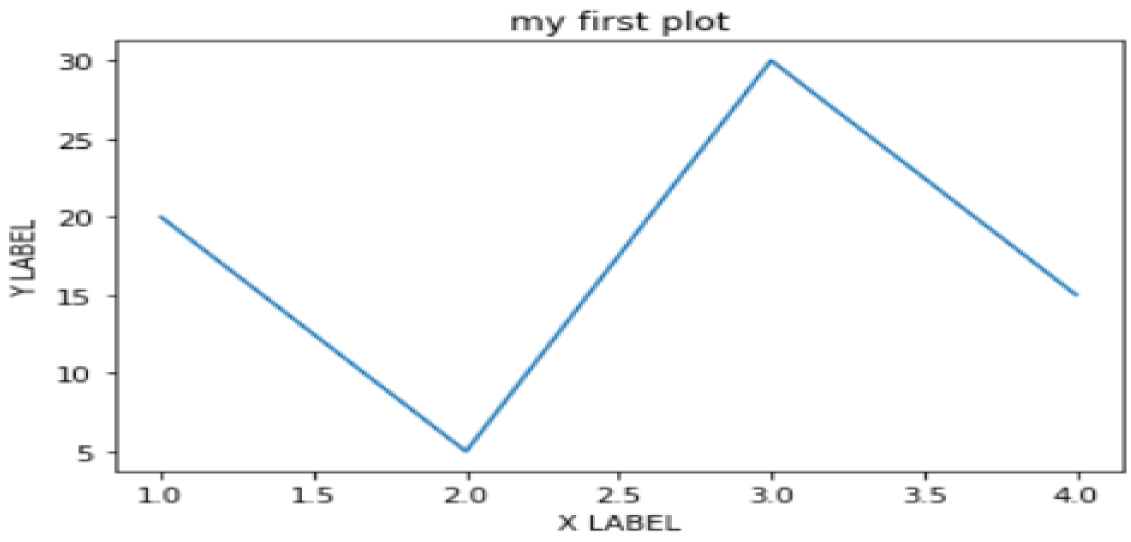
If we provide a single list to the plot(), matplotlib assumes it is a sequence of y values, and automatically generates the x values.. i.e show in below example

```
import matplotlib.pyplot as plt
import numpy as np
plt.plot([10,20,30,40])
plt.show()
```



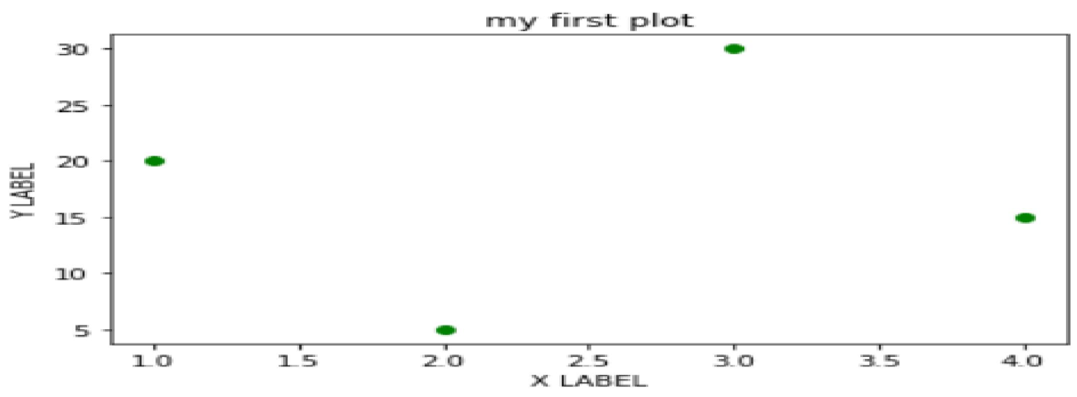
Now that our first plot is ready, let us add the title, and name x-axis and y-axis using methods title(), xlabel() and ylabel() respectively.

```
import matplotlib.pyplot as plt
import numpy as np
plt.plot([1,2,3,4],[20,5,30,15])
plt.title("my first plot")
plt.xlabel("X LABEL")
plt.ylabel("Y LABEL")
plt.show()
```



The default format is b- which means a solid blue line. In the figure below we use go which means green circles. Likewise, we can make many such combinations to format our plot.

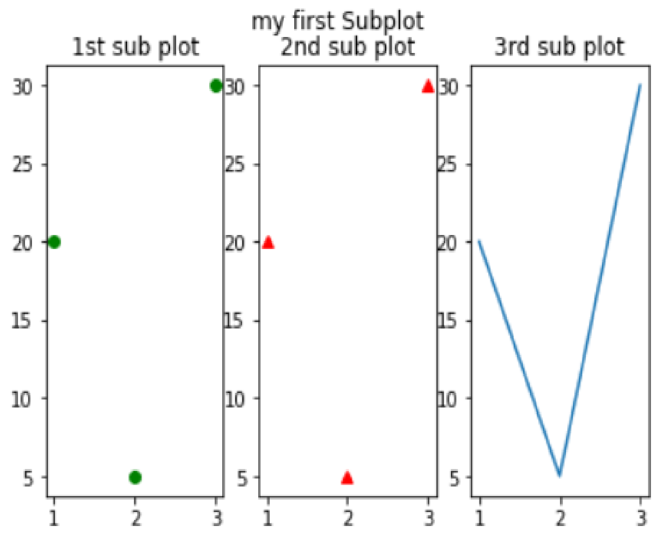
```
import matplotlib.pyplot as plt
import numpy as np
plt.plot([1,2,3,4],[20,5,30,15],"go")
plt.title("my first plot")
plt.xlabel("X LABEL")
plt.ylabel("Y LABEL")
plt.show()
```



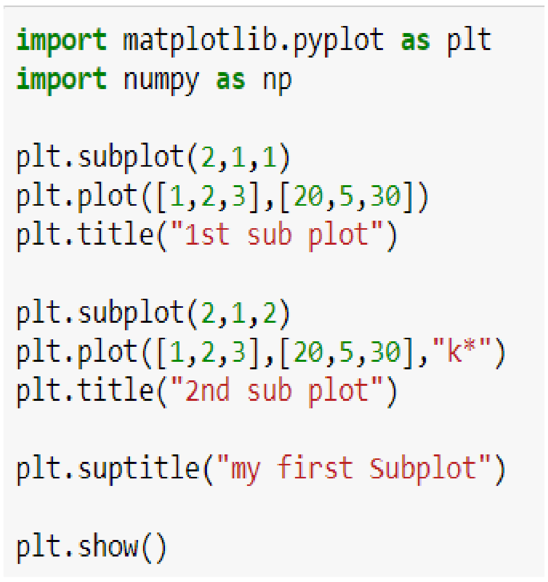
## Multiple plots in one figure:

We can use subplot() method to add more than one plots in one figure. In the image below, we used this method to separate two graphs which we plotted on the same axes in the previous example. The subplot() method takes three arguments: they are nrows, ncols and index. They indicate the number of rows, number of columns and the index number of the sub-plot. For instance, in our example, we want to create three sub-plots in one figure such that it comes in one row and in three columns and hence we pass arguments (1,3,1) , (1,3,2) and (1,3,3) in the subplot() method. Note that we have separately used title() method for both the subplots. We use suptitle() method to make a centralized title for the figure.

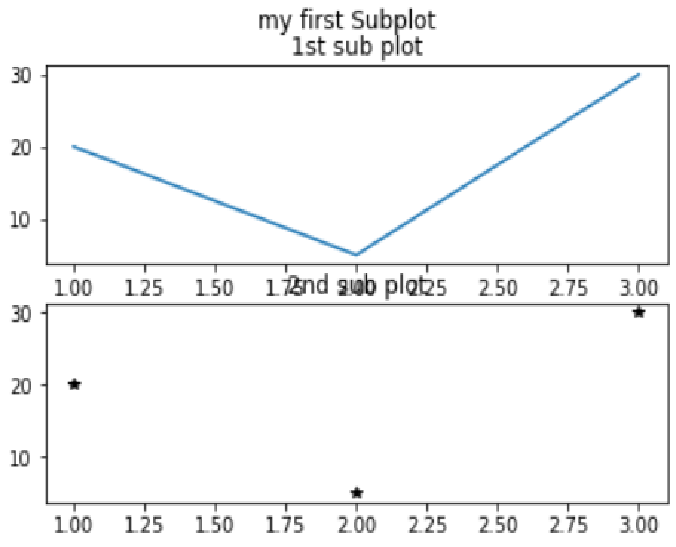
```
import matplotlib.pyplot as plt
import numpy as np

plt.subplot(1,3,1)
plt.plot([1,2,3],[20,5,30],"go")
plt.title("1st sub plot")

plt.subplot(1,3,2)
plt.plot([1,2,3],[20,5,30],"r^")
plt.title("2nd sub plot")

plt.subplot(1,3,3)
plt.plot([1,2,3],[20,5,30])
plt.title("3rd sub plot")

plt.suptitle("my first Subplot")

plt.show()
```



If we want our sub-plots in two rows and single column, we can pass arguments (2,1,1) and (2,1,2)



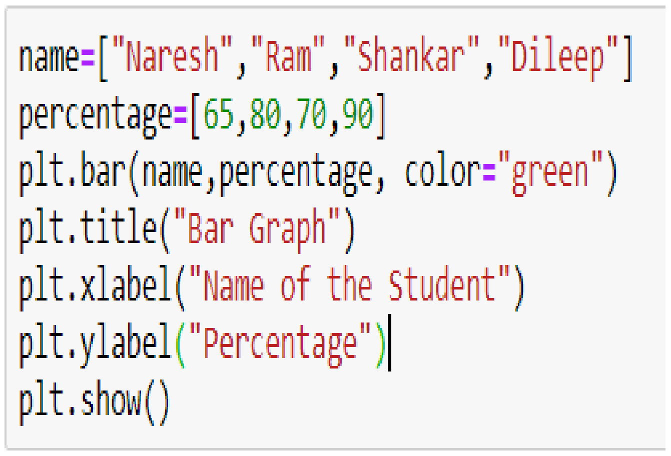
```python
import matplotlib.pyplot as plt
import numpy as np

plt.subplot(2,1,1)
plt.plot([1,2,3],[20,5,30])
plt.title("1st sub plot")

plt.subplot(2,1,2)
plt.plot([1,2,3],[20,5,30],"k*")
plt.title("2nd sub plot")

plt.suptitle("my first Subplot")

plt.show()
```



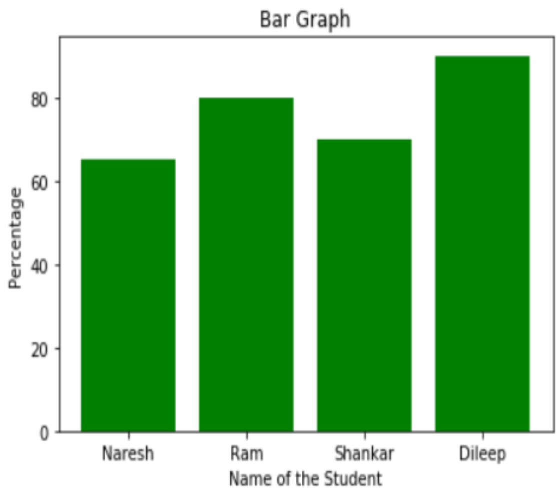
## Bar Graphs

A bar graph uses bars to compare data among different categories. It is well suited when you want to measure the changes over a period of time. It can be represented horizontally or vertically. Also, the important thing to keep in mind is that longer the bar, greater is the value. Now, let us practically implement it using python matplotlib.

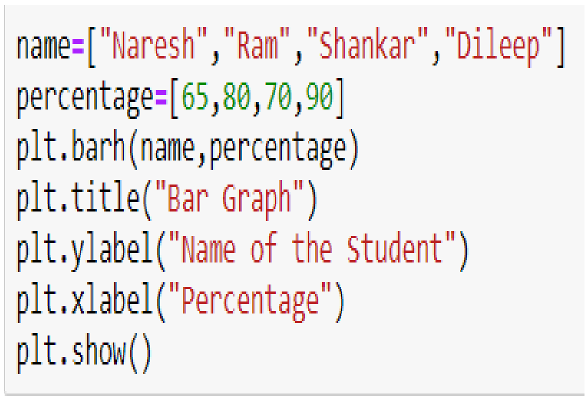
Pyplot provides a method bar() to make bar graphs which take arguments: categorical variables, their values and color (if you want to specify any).



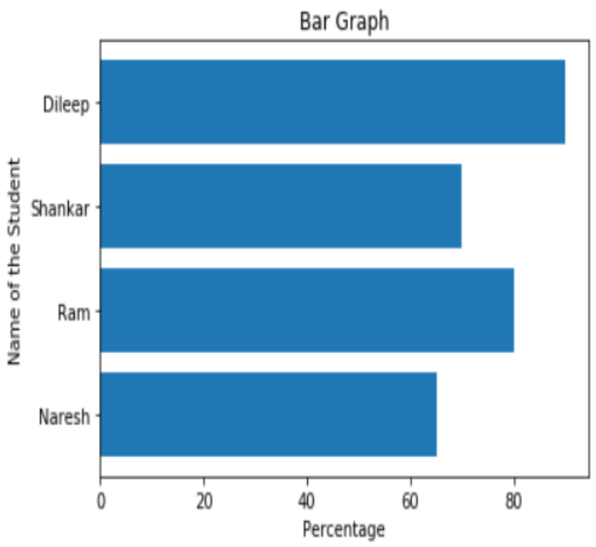
```python
name=["Naresh","Ram","Shankar","Dileep"]
percentage=[65,80,70,90]
plt.bar(name,percentage, color="green")
plt.title("Bar Graph")
plt.xlabel("Name of the Student")
plt.ylabel("Percentage")
plt.show()
```



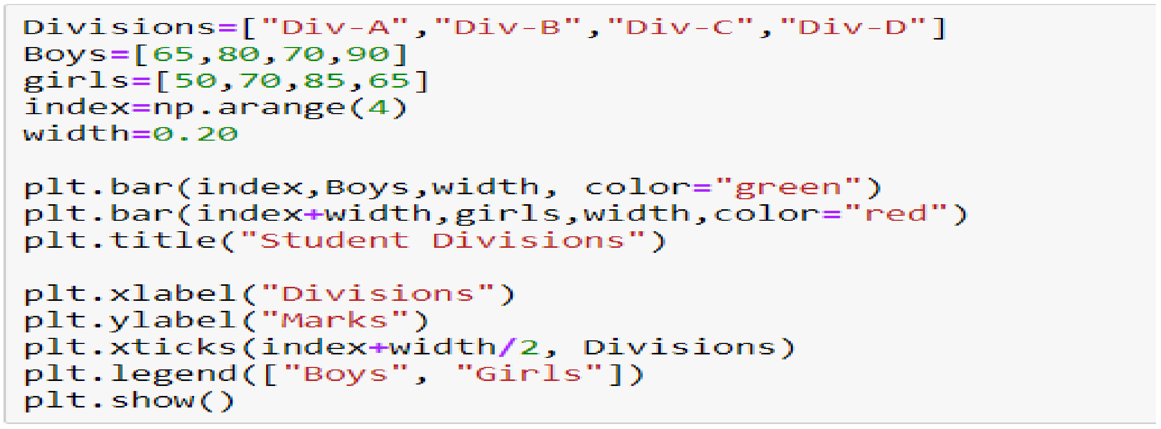
To make horizontal bar graphs use method **barh() i.e**



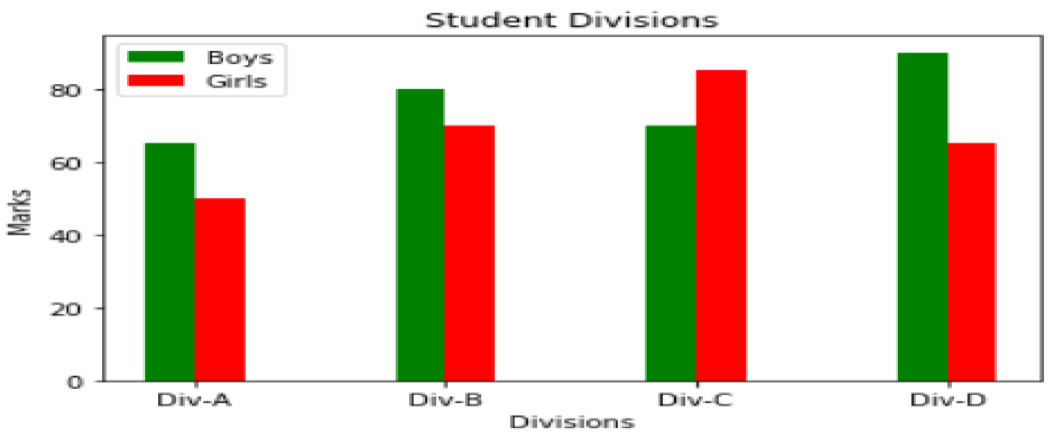
```
name=["Naresh","Ram","Shankar","Dileep"]
percentage=[65,80,70,90]
plt.barh(name,percentage)
plt.title("Bar Graph")
plt.ylabel("Name of the Student")
plt.xlabel("Percentage")
plt.show()
```



To create horizontally stacked bar graphs we use the bar() method twice and pass the arguments where we mention the index and width of our bar graphs in order to horizontally stack them together. Also, notice the use of two other methods legend() which is used to show the legend of the graph and xticks() to label our x-axis based on the position of our bars.



```
Divisions=["Div-A","Div-B","Div-C","Div-D"]
Boys=[65,80,70,90]
girls=[50,70,85,65]
index=np.arange(4)
width=0.20

plt.bar(index,Boys,width, color="green")
plt.bar(index+width,girls,width,color="red")
plt.title("Student Divisions")

plt.xlabel("Divisions")
plt.ylabel("Marks")
plt.xticks(index+width/2, Divisions)
plt.legend(["Boys", "Girls"])
plt.show()
```

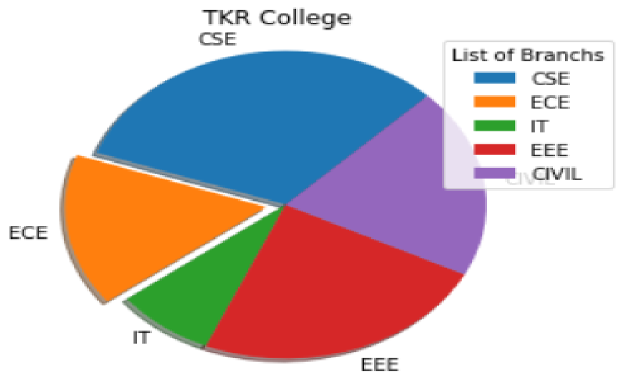


## Pie Graph:

A pie chart refers to a circular graph which is broken down into segments i.e. slices of pie. It is basically used to show the percentage or proportional data where each slice of pie represents a category.

Pyplot provides a method pie() to make pie graphs. We can also pass in arguments to customize our Pie chart to show shadow, explode a part of it, tilt it at an angle as follows:

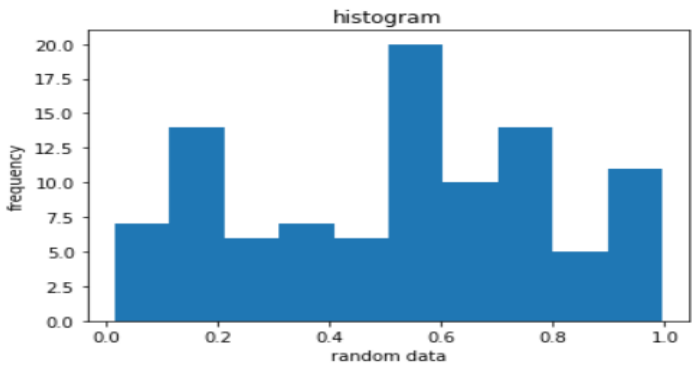
```
branch=["CSE","ECE","IT","EEE","CIVIL"]
seats=[240,120,60,180,150]
Explode=[0,0.1,0,0,0]
plt.pie(seats,explode=Explode, labels=branch,shadow=True,startangle=45)
plt.axis("equal")
plt.legend(title="List of Branchs")
plt.title("TKR College")
plt.show()
```



## Histograms

Histograms are a very common type of plots when we are looking at data like height and weight, stock prices, waiting time for a customer, etc which are continuous in nature. Histogram’s data is plotted within a range against its frequency. Histograms are very commonly occurring graphs in probability and statistics and form the basis for various distributions like the normal -distribution, t-distribution, etc. In the following example, we generate a random continuous data of 100 entries and plot it against its frequency with the data divided into 10 equal strats.

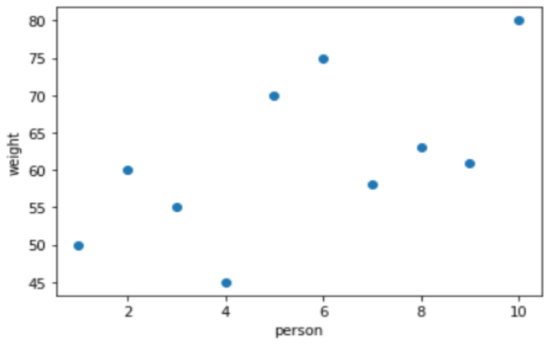
```
x=np.random.rand(100)
plt.title("histogram")
plt.xlabel("random data")
plt.ylabel("frequency")
plt.hist(x,10)
plt.show()
```



## Scatter Plots and 3-D plotting

Scatter plots are widely used graphs, especially they come in handy in visualizing a problem of regression. Usually we need scatter plots in order to compare variables, for example, how much one variable is affected by another variable to build a relation out of it. The data is displayed as a collection of points, each having the value of one variable which determines the position on the horizontal axis and the value of other variable determines the position on the vertical axis.

```
import matplotlib.pyplot as plt
import numpy as np
p=np.array([1,2,3,4,5,6,7,8,9,10])
w=np.array([50,60,55,45,70,75,58,63,61,80])
plt.scatter(p,w)
plt.xlabel("person")
plt.ylabel("weight")
plt.show()
```

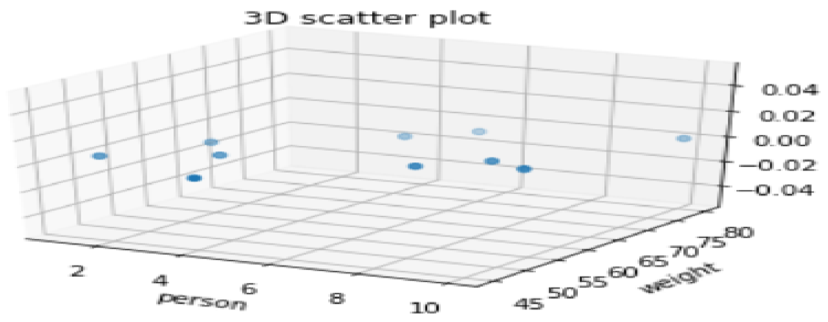


The above scatter can also be visualized in three dimensions. To use this functionality, we first import the module mplot3d as follows:

from mpl_toolkits import mplot3d

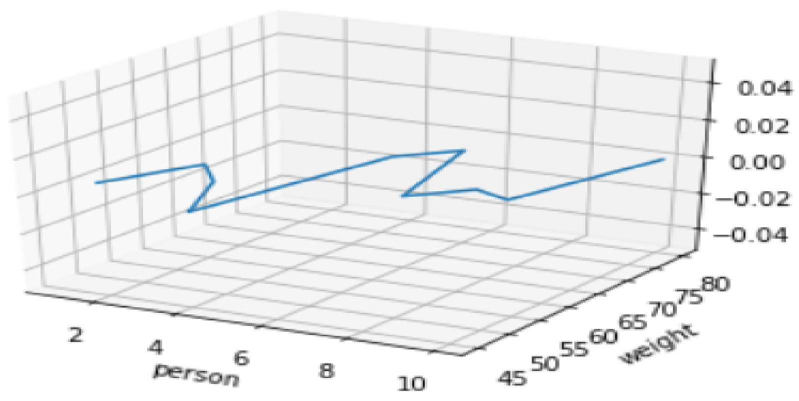
Once the module is imported, a three-dimensional axes is created by passing the keyword projection='3d' to the axes() method of Pyplot module. Once the object instance is created, we pass our arguments height and weight to scatter3D() method.

```
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d
import numpy as np
p=np.array([1,2,3,4,5,6,7,8,9,10])
w=np.array([50,60,55,45,70,75,58,63,61,80])
ax=plt.axes(projection="3d")
ax.scatter3D(p,w)
ax.set_xlabel("person")
ax.set_ylabel("weight")
ax.set_title("3D scatter plot")
plt.show()
```



We can also create 3-D graphs of other types like line graph, surface, wireframes, contours, etc. The above example in the form of a simple line graph is as follows: Here instead of scatter3D() we use method plot3D()

```
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d
import numpy as np
p=np.array([1,2,3,4,5,6,7,8,9,10])
w=np.array([50,60,55,45,70,75,58,63,61,80])
ax=plt.axes(projection="3d")
ax.plot3D(p,w)
ax.set_xlabel("person")
ax.set_ylabel("weight")
plt.show()
```



## List of methods used in matplotlib.pyplot:

- **plot(x-axis values, y-axis values)** — plots a simple line graph with x-axis values against y-axis values
- **show()** — displays the graph
- **title("string")** — set the title of the plot as specified by the string
- **xlabel("string")** — set the label for x-axis as specified by the string
- **ylabel("string")** — set the label for y-axis as specified by the string
- **figure()** — used to control a figure level attributes
- **subplot(nrows, ncols, index)** — Add a subplot to the current figure
- **suptitle("string")** — It adds a common title to the figure specified by the string
- **subplots(nrows, ncols, figsize)** — a convenient way to create subplots, in a single call. It returns a tuple of a figure and number of axes.
- **set_title("string")** — an axes level method used to set the title of subplots in a figure
- **bar(categorical variables, values, color)** — used to create vertical bar graphs
- **barh(categorical variables, values, color)** — used to create horizontal bar graphs
- **legend(loc)** — used to make legend of the graph

- **xticks(index, categorical variables)** — Get or set the current tick locations and labels of the x-axis
- **pie(value, categorical variables)** — used to create a pie chart
- **hist(values, number of bins)** — used to create a histogram
- **xlim(start value, end value)** — used to set the limit of values of the x-axis
- **ylim(start value, end value)** — used to set the limit of values of the y-axis
- **scatter(x-axis values, y-axis values)** — plots a scatter plot with x-axis values against y-axis values
- **axes()** — adds an axes to the current figure
- **set_xlabel("string")** — axes level method used to set the x-label of the plot specified as a string
- **set_ylabel("string")** — axes level method used to set the y-label of the plot specified as a string
- **scatter3D(x-axis values, y-axis values)** — plots a three-dimensional scatter plot with x-axis values against y-axis values
- **plot3D(x-axis values, y-axis values)** — plots a three-dimensional line graph with x-axis values against y-axis values .

## <u>The matplotlib supports the following color abbreviation:</u>

| Character | Color | Character | Color |
|---|---|---|---|
| 'b' | Blue | 'y' | Yellow |
| 'g' | Green | 'k' | Black |
| 'r' | Red | 'w' | White |
| 'c' | Cyan | 'm' | Magenta |

**Example format String**

| | |
|---|---|
| **'b'** | Using for the blue marker with default shape. |
| **'ro'** | Red circle |
| **'-g'** | Green solid line |
| **'--'** | A dashed line with the default color |
| **'^k:'** | Black triangle up markers connected by a dotted line |