Chapter 4.

Python Functions, Modules and Packages

14 Marks

Introduction:

- Functions, modules and packages are all constructs in python programming that promote code modularization. The modularization refers to the process of breaking a large programming task into a separate, smaller, more manageable subtasks or modules.
- ***A function is a block of organized, reusable code which can be called whenever required. A function is a piece of code that performs a particular task.***
- ***A module in programming allows us to logically organize the python code***. A module is a single source code file. The module in python have the .py extension. The name of the module will be the name of the file.
- ***A python module can be defined as a python program file which contains a python code including python function, class or variables***.
- Python packages allow us to create a hierarchical file directory structure of modules. ***A python package is a collection of modules which have a common purpose***. i.e modules are grouped together to form package.

4.1 Use of Python Built-in Functions:

- Functions are ***self contained block of statements*** that act like a program that ***perform specific task.***
- The python interpreter has a number of functions that are always available for use. These functions are called ***built-in functions***. E.g. print() functions prints the given object to the standard output device.

Type Data Conversion Functions-

- It is necessary to perform conversion between the built-in types. To convert between types we simply use the **type** name as a function.
- Python define type conversion functions to directly convert one type of data type to another.

Python Implicit Data Type Conversion:

- It takes place either during compilation or during run time and is handled directly by python.

Example **: Converting integer to float**

```
num_int = 123

num_flo = 1.23

num_new = num_int + num_flo

print("datatype of num_int:",type(num_int))

print("datatype of num_flo:",type(num_flo))
print("Value of num_new:",num_new)

print("datatype of num_new:",type(num_new))
```

output

```
datatype of num_int: <class 'int'>

datatype of num_flo: <class 'float'>

Value of num_new: 124.23

datatype of num_new: <class 'float'>
```

In the above program,

- We add two variables num_int and num_flo, storing the value in num_new.
- look at the data type of all three objects respectively.
- In the output we can see the datatype of num_int is an integer, datatype of num_flo is a float.
- Also, we can see the num_new has float data type because Python always converts smaller data type to larger data type to avoid the loss of data.

Python Explicit Data Type Coversion -

- In Explicit Type Conversion, users ***convert the data type of an object to required data type***. We use the predefined functions like **int(), float(), str()**, etc to perform explicit type conversion.

- This type conversion is also called ***typecasting*** because the user casts (change) the data type of the objects.

Syntax :

(required_datatype)(expression)

Typecasting can be done by assigning the required data type function to the expression.

1. int(a,base) : This function converts **any data type to integer**. 'Base' specifies the **base in which string is** if data type is string.

2. float() : This function is used to convert **any data type to a floating point number.**

3. ord() : This function is used to convert a **character to integer.**

4. hex() : This function is to convert **integer to hexadecimal string**.

5. oct() : This function is to convert **integer to octal string**.

6. tuple() : This function is used to **convert to a tuple**.

7. set() : This function returns the **type after converting to set**.

8. list() : This function is used to convert **any data type to a list type**.

9. dict() : This function is used to **convert a tuple of order (key,value) into a dictionary**.

10. str() : Used to **convert integer into a string.**

11. complex(real,imag) : : This function **converts real numbers to complex(real,imag) number.**

***Example 1*:-**

```
# Python code to demonstrate Type conversion
# using int(), float()

# initializing string
s = "10010"

# printing string converting to int base 2
c = int(s,2)
print ("After converting to integer base 2 : ", end="")
print (c)

# printing string converting to float
e = float(s)
print ("After converting to float : ", end="")
print (e)
```

Output:

```
After converting to integer base 2 : 18
After converting to float : 10010.0
```

Example 2:-

```
# Python code to demonstrate Type conversion
# using  ord(), hex(), oct()
# initializing integer
s = ‘A'
# printing character converting to integer
c = ord(s)
print ("After converting character to integer : ",end="")
print (c)
# printing integer converting to hexadecimal string
c = hex(56)
print ("After converting 56 to hexadecimal string : ",end="")
print (c)
# printing integer converting to octal string
c = oct(56)
print ("After converting 56 to octal string : ",end="")
print (c)
```

Output:

```
After converting character to integer : 65
After converting 56 to hexadecimal string : 0x38
After converting 56 to octal string : 0o70
```

Example 3:-

```
# Python code to demonstrate Type conversion using  tuple(), set(), list()

# initializing string

s = ‘hello'

# printing string converting to tuple

c = tuple(s)

print ("After converting string to tuple : ",end="")

print (c)

# printing string converting to set

c = set(s)

print ("After converting string to set : ",end="")

print (c)

# printing string converting to list

c = list(s)

print ("After converting string to list : ",end="")

print (c)
```

OUTPUT:

```
After converting string to tuple : ('h', 'e', 'l', 'l', 'o')

After converting string to set : {'l', 'e', 'o', 'h'}
After converting string to list : ['h', 'e', 'l', 'l', 'o']
```

Example 4:-

```
# Python code to demonstrate Type conversion using dict(), complex(), str()

# initializing integers

a = 1

b = 2

# initializing tuple

tup = (('a', 1) ,('f', 2), ('g', 3))

# printing integer converting to complex number

c = complex(1,2)

print ("After converting integer to complex number : ",end="")

print (c)

# printing integer converting to string

c = str(a)

print ("After converting integer to string : ",end="")

print (c)

# printing tuple converting to expression dictionary

c = dict(tup)

print ("After converting tuple to dictionary : ",end="")

print (c)
```

OUTPUT:

```
After converting integer to complex number : (1+2j)

After converting integer to string : 1

After converting tuple to dictionary : {'a': 1, 'f': 2, 'g': 3}
```

Formatting Numbers and Strings –

- The built-in **format()** method returns **a *formatted representation of the given value controlled by the format specifier*.**

  Syntax:

  format(value,format)

e.g.

```
x=12.345

print(format(x,".2f"))
```

Output

```
12.35
```

Parameter values-

< - *Left aligns the result within the available space*

e.g.

x=10.23456

print(format(x,"<10.2f"))

Output:

‘10.23   ‘

> - *Right aligns the result within the available space*

e.g.

x=10.23456

print(format(x,“>10.2f"))

Output:

‘   10.23 ‘

^ - *Center aligns the result within the available space*

e.g.

x=10.23456

print(format(x,“^10.2f"))

Output:

‘   10.23   ‘

+, - - *Use a sign to indicate if the result is positive or negative*

e.g.

x=10.23456

y=-10.23456

print(format(x,"+"))

```
print(format(y,"-"))
```

Output:

```
+10.23456
-10.23456
```

,   - *Use a comma as athousand sepeartor*

e.g.

```
x=10000000
print(format(x,","))
```

Output:

```
10,000,000
```

_   -   *Use a underscore as a thousand seperator.*

e.g.

```
x=10000000
print(format(x,"_"))
```

Output:

```
10_000_000
```

b - *binary format*

e.g.

```
x=10
```

```
print(format(x,"b"))
```

Output:

```
1010
```

c – ***convert the value to corresponding unicode character.***

```
x=10
print(format(x,"c"))
```

Output:

```
\n
```

e - ***Scientific character with a lower case e***

e.g.

```
x=10
print(format(x,"e"))
```

Output:

```
1.000000e+01
```

E – ***Scientific character with a upper case E.***

```
x=10
print(format(x,“E"))
```

Output:

```
1.000000E+01
```

f - ***fix point number format***

e.g.

```
x=10
print(format(x,“f"))
```

Output:

```
10.000000
```

g – ***General Format.***

```
x=10
print(format(x,“g"))
```

Output:

```
10
```

o - ***octal format***

e.g.

```
x=10
print(format(x,“o"))
```

Output:

```
12
```

x – ***Hex Format, lower case***

x=10

print(format(x,“x"))

Output:

a

X **- *Hex format, Upper case***

e.g.

x=10

print(format(x,“X"))

Output:

A

% – Percentage format

x=100

print(format(x,“%"))

Output:

10000.000000%

>10s **- *String with width 10 in left justification***

e.g.

x=“hello”

print(format(x,“>10s"))

Output:

```
‘     hello’
```

<10s - *String with width 10 in right justification*

```
x=“hello”
print(format(x,“<10s"))
```

Output:

```
‘hello     ‘
```

Built-in Mathematical Functions -

- In Explicit Type Conversion, users ***convert the data type of an object to required data type***. We use the predefined functions like **int(), float(), str()**, etc to perform explicit type conversion.
- This type conversion is also called ***typecasting*** because the user casts (change) the data type of the objects.

Syntax :

```
(required_datatype)(expression)
```

Typecasting can be done by assigning the required data type function to the expression.

Built-In functions(Mathematical):

- These are the functions which doesn’t require any external code file/modules/ Library files.

- These are a part of the python core and are just built within the python compiler hence there is no need of importing these modules/libraries in our code.

min()- Returns smallest value among supplied arguments.

e.g. print(min(20,10,30))

o/p   10

max()- Returns largest value among supplied arguments.

e.g. print(max(20,10,30))

o/p   30

pow()- Returns the value of x to the power of y.

e.g. print(pow(2,3))

o/p   8

round()- Returns a floating point number that is rounded version of the specified number, with the specified number of decimals.

The default number of decimals is 0, i.e it will return nearest integer.

e.g. print(round(10.2345))

o/p   10

print(round(5.76543))

o/p   6

print(round(5.76543,2))

o/p   5.77

abs()- Returns the non negative value of the argument value.

e.g. print(abs(-5))

o/p 5

Built-In functions(Math Module):

- The second type of function requires some external files(modules) in order to be used. The process of using these external files in our code is called importing.(we need to import math.)

ceil()- Returns smallest integral value grater than the number.

```
e.g. import math
     print(math.ceil(2.3))
o/p  3
```

floor()- Returns the greatest integral value smaller than the number.

```
e.g. import math
     print(math.floor(2.3))
o/p  2
```

cos()- Returns the cosine of value passed as argument. The value passed in this function should be in radians.

```
e.g. import math
     print(math.cos(3))
o/p  -0.9899924966004454
```

cosh()- Returns the hyberbolic cosine of x.

e.g. import math

```
      print(math.cosh(3))
```

o/p   10.067661995777765

copysign()- Returns x with sign of y.

e.g. import math

```
      print(math.copysign(10,-12))
```

o/p   -10.0

exp()- Returns the exponential of x.

e.g. import math

```
      print(math.exp(1))
```

o/p 2.718281828459045

fabs()- Returns the absolute or positive value.

e.g. import math

```
      print(math.fabs(-20))
```

o/p   20.0

factorial()- Returns factorial of x.

e.g. import math

```
      print(math.factorial(5))
```

o/p   120

fmod()- Returns the x % y.

e.g. import math

    print(math.fmod(50,20))

o/p 10.0

log(a,(base))- Used to compute the natural logarithm(base e) of a.

e.g. import math

    print(math.log(14))

o/p   2.6390573296152584

log2(a)- Used to compute the logarithm(base 2) of a.

e.g. import math

    print(math.log2(14))

o/p 3.807354922057604

log10(a)- Used to compute the logarithm(base 10) of a.

e.g. import math

    print(math.log10(14))

o/p 1.146128035678238

sqrt()- Returns the square root of x for x>0.

e.g. import math

    print(math.sqrt(100))

o/p 10.0

trunc()- Returns the truncated integer of x.

e.g. import math

```
    print(math.trunc(3.354))
```

o/p 3

4.2 User Defined Functions–

- A user defined function is a block of related code statements that are organized to achieve a single related action or task.
- A key objective of the concept of the user defined function is to encourage modularity and enable reusability of code.

Function Definition:

- Function definition is a block where the statements inside the function body are written.

Syntax:

```
def function_name(parameters):
“function_docstring”
function_suite
return [expression]
```

Function Definition:

- Function blocks starts with the keyword def taken after by the function name and parentheses().
- Any input parameters or arguments should be put inside these brackets.

- The first statement of a function can be optional statement- the documentation string of the function or doc string.
- The code block inside each capacity begins with a colon(:) and is indented.
- The statement return[expression] exits a function, alternatively going back an expression to the caller. A return statemnt with no arguments is the same as return None.

Function Calling:

The def statement only creates a function but does not call it. After the def has run, we call the function by adding parentheses after the function's name.

Example:

```
def square(x):  #function definition
        return x*x
square(4)   # function call
```

output:

```
16
```

Advantages of User defined functions:

1. It help to decompose a large program into small segments which makes program easy to understand, maintain and debug.
2. We can call python function any number of times in a program and from any place in a program.

3. Reusability is the main achievement of python functions.
4. Functions in python reduces the overall size of the program.
5. We can track a large python program easily when it is divided into multiple functions.
6. By using functions, we can avoid rewriting same logic/code again and again in a program

Concept of Actual and Formal Parameters: **(Parameter Passing)**

Actual Parameters:

- ***The parameters used in the function call are called actual parameters***. These are the actual values that are passed to the function.
- The data types of actual parameters must match with the corresponding data types of formal parameters(variables) in the function definition.

1. They are used in the function call.
2. They are actual values that are passed to the function definition through the function call.
3. They can be constant values or variable names(such a local or global)

Formal Parameters:

- ***The parameters used in the header of function definition are called formal parameters of the function***. These parameters are used to receive values from the calling function.

1. They are used in function header.
2. They are used to receive the values that are passed to the function through function call.
3. They are treated as local variables of a function in which they are used in the function header.

Example:

```
def cube(x):            # formal parameters
    return x*x*x
result=cube(7)          # actual parameters
print(result)
```

Output:

```
343
```

Call by object reference:

Python utilizes a system, which is known as “Call by Object Reference” or “Call by assignment”. In the event that you pass arguments like whole numbers, strings or tuples to a function, the passing is like call-by-value because you can not change the value of the immutable objects being passed to the function. Whereas passing mutable objects can be considered as call by reference because when their values are changed inside the function, then it will also be reflected outside the function.

***Example*: Python code to demonstrate call by value**

```
string = "Hello"

def test(string):

    string = "Hello World"

    print("Inside Function:", string)

test(string)

print("Outside Function:", string)
```

Output:

```
Inside Function: Hello World

Outside Function: Hello
```

***Example :* Python code to demonstrate call by reference**

```
def add_more(list):

    list.append(50)

    print("Inside Function", list)

mylist = [10,20,30,40]

add_more(mylist)

print("Outside Function:", mylist)
```

Output:

```
Inside Function [10, 20, 30, 40, 50]

Outside Function: [10, 20, 30, 40, 50]
```

Function Arguments:

Many built-in functions need arguments to be passed with them. Many functions require two or more arguments. The value of argument is always assigned to a variable known as **parameter**.

There are four types of arguments using which can be called are ***required arguments, keyword arguments, default arguments and variable length arguments***.

Required Arguments:

These are the arguments passed to a function in correct positional order. Here the number of arguments in the function call should match exactly with the function definition.

Example: Required Arguments

```
def printme(str):
   "This prints a passed string into this function"
   print(str)
   return
# Call printme function
```

```
mylist=[10,20,30]

printme()       # required arguments
```

Output:

```
Traceback (most recent call last):

  File "D:\python programs\sample.py", line 8, in <module>

    printme()

TypeError: printme() missing 1 required positional argument: 'str'
```

<u>Keyword Arguments</u>:

These are the arguments related to the function calls. When we use it in the function call, the caller identifies the arguments by the parameter name

This allows us to skip arguments or place them out of order because the python interpreter is able to use the keywords provided to match the values with parameters.

Example 1: Keyword Arguments

```
# Keyword Arguments

def printme(str):
    "This prints a passed string into this function"
    print(str)
    return
# Call changename function
printme(str="My string")
```

Output:

```
My string
```

Example 2: Keyword Arguments

```
def printinfo(name,age):
    "This prints a passed info into this function"
    print("Name: ",name)
    print("Age: ",age)
    return
# Call changename function
printinfo(age=35,name="Amol")
```

Output:

```
Name:  Amol

Age:  35
```

<u>Default Arguments</u>:

A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument.

Example: **Default Arguments**

```
def printinfo(name,age=35):
   "This prints a passed info into this function"
   print("Name: ",name)
   print("Age: ",age)
   return
# Call changename function
printinfo(age=40,name="Amol")
printinfo(name="Kedar")
```

Output:

```
Name:  Amol

Age:  40

Name:  Kedar
```

Age: 35

<u>Variable length Arguments</u>:

In many cases where we are required to process a function with more number of arguments than we specified in the function definition. These types of arguments are known as variable length arguments.

Example **: Variable length Arguments**

```
def printadd(farg,*args):
   "To add given numbers"
   print("Output is: ",farg)
   sum=0
   for i in args:
      sum+=i
   print("sum of all numbers",(farg+sum))
   return
```

```
# Call printadd function
printadd(5,10)
printadd(5,10,20,30)
```

Output:

```
Output is:  5
sum of all numbers 15
Output is:  5
sum of all numbers 65
```

Return Statement:

- The return statement return[expression] exits a function,

alternatively going back an expression to the caller.

- A return statement with no arguments is the same as return none.

Example:

```
# A function to calculate factorial value
def fact(n):
    prod=1
    while n>=1:
        prod*=n

        n-=1

     return prod

# Call fact function

for i in range(1,11):

    print('Factorial of {} is {}'.format(i,fact(i)))
```

Output:

```
Factorial of 1 is 1

Factorial of 2 is 2

Factorial of 3 is 6

Factorial of 4 is 24

Factorial of 5 is 120

Factorial of 6 is 720

Factorial of 7 is 5040

Factorial of 8 is 40320

Factorial of 9 is 362880
Factorial of 10 is 3628800
```

Scope of Variables:

- All variables in a program may not be available at all areas in that program. This relies on upon where you have announced a variable.
- There are two basic scopes of variables in python

Local variables:

- When we declare a variable inside a function, it becomes a local variable.
- Its scope is limited only to that function where it is created. That means the local variable is available only in that function but not outside that function.

Global variables:

- When we declare a above a function it becomes a global variable.

- Such variables are available to all functions which are written after it.

Example: **Local variable in a function**

```
def myfunction():
   a=1  # local variable
   a+=1
   print(a)
# Call myfunction
myfunction()
print(a)
```

Output:

```
2
Traceback (most recent call last):
  File "F:\2019-2020\PYTH prog\for_ex.py", line 9, in <module>
   print(a)
NameError: name 'a' is not defined
```

Example: **Global variable in a function**

```
a=1  # global variable

def myfunction():

    b=2  # local variable

    print('a= ',a)

    print('b= ',b)

# Call myfunction

myfunction()

print(a)

print(b)
```

Output:

```
a=  1

b=  2

1

Traceback (most recent call last):

  File "F:\2019-2020\PYTH prog\for_ex.py", line 11, in <module>

    print(b)

NameError: name 'b' is not defined
```

4.3 Modules:

- A module are primarily **.py files** that represents ***group of classes, methods and functions***.
- We have to ***group them depending on their relationship*** into various modules and later use these modules in other programs. It means, when a module is developed it can be ***reused in any program that needs that module***.
- In python there are several built-in modules. Just like these modules we can create our own modules and use whenever we need them.
- Once a module is created, any programmer in the project team can use that module. Hence modules will make software development easy and faster.

Advantages of modules –

- **Reusability**: Working with modules makes the code reusability a reality.

- **Simplicity:** Module focuses on a small proportion of the problem, rather than focusing on the entire problem.
- **Scoping:** A separate namespace is defined by a module that helps to avoid collisions between identifiers.

Writing Module:

Writing a module means simply creating a file which contains python definition and statements. The file name is the module name with the extension .py. To include module in a file, use the import statement.

Follow the following steps to create modules:

1. Create a first file as a python program with extension as .py. This is your module file where we can write a function which performs some task.
2. Create a second file in the same directory called main file where we can import the module to the top of the file and call the function.

Example:

*Create **p1.py** file and add the below code in the file*

```
def add(a,b):
    result=a+b

    return result

def sub(a,b):

    result=a-b

    return result

def mul(a,b):

    result=a*b

    return result

def div(a,b):

    result=a/b

    return result
```

Create ***p2.py*** *file I same directory where p1.py is created and add the below code in the file*

```
import p1

print("Addition= ",p1.add(10,20))

print("Subtraction= ",p1.sub(10,20))
print("Multiplication= ",p1.add(10,20))

print("Division= ",p1.add(10,20))
```

Output:

```
Addition=  30

Subtraction=  -10

Multiplication=  30

Division=  30
```

Importing Modules:

Import statement is used to import a specific module by using its name. Import statement creates a reference to that module in the current namespace. After using import statement we can refer the things defined in that module.

- Use import statement to make use of module

e.g.

```
Import employee
```

- In this case we have to access the functions in employee module using employee.hra(), employee.da() etc. To avoid this we can use

```
from employee import *

from employee import hra, da
```

***Example*: using from x import ***

```
from p1 import *

print("Addition= ",add(10,20))

print("Multiplication= ",mul(10,20))
```

Output:

```
Addition=  30

Multiplication=  200
```

***Example*: using from x import a,b**

```
from p1 import add,sub

print("Addition= ",add(10,20))

print("Subtraction= ",sub(10,20))
```

Output:

```
Addition=  30
```

```
Subtraction=  -10
```

Import with renaming (Aliasing Modules):

- It is possible to modify the names of modules and their functions within python by using the ‘as’ keyword.
- We can make alias because we have already used the same name for something else in the program or we may want to shorten a longer name.

Syntax:

```
import module as another_name
```

e.g. if a module created with name p1 which has function named ‘add’

```
import p1 as m
print(‘addition is :’,m.add)
```

Python Built-in Modules:

- A module is a ***collection of python objects such as functions, classes*** and so on.

- ***Python interpreter*** is bundled with a ***standard library*** consisting of large number of built-in modules.
- Built-in modules are generally ***written in C and bundled with python interpreter in precompiled form***. A built-in module may be a python script(.py) containing useful utilities.
- A module may contain one or more functions, classes, constants or any other python resources.

Numeric and Mathematical Modules:

- This module provides numeric and math related functions and data types. Following are the modules which are classified as numeric and mathematical modules.

1. numbers(Numeric abstract base classes)
2. math(Mathematical functions)
3. cmath(Mathematical functions for complex numbers)
4. decimal(Decimal fixed point and floating point arithmetic)
5. fractions(Rational numbers)
6. random(Generate pseudo random numbers)
7. Statistics(Mathematical statistics functions)

math module

The math module is a standard module in Python and is always available. To use mathematical functions under this module, you have to import the module using **import math**.

It gives access to hyperbolic, trigonometric and logarithmic functions for real numbers.

```
# importing built-in module math

import math

# using square root(sqrt) function contained

# in math module

print math.sqrt(25)

# using pi function contained in math module

print math.pi

# 2 radians = 114.59 degreees

print math.degrees(2)

# 60 degrees = 1.04 radians

print math.radians(60)
```

```
# Sine of 2 radians

print math.sin(2)

# Cosine of 0.5 radians

print math.cos(0.5)

# Tangent of 0.23 radians

print math.tan(0.23)


# 1 * 2 * 3 * 4 = 24

print math.factorial(4)
```

Output:

```
5.0

3.14159265359

114.591559026

1.0471975512

0.909297426826

0.87758256189
```

```
0.234143362351

24
```

cmath module

The cmath module allows us to work with mathematical functions for complex numbers.

Example

```
from cmath import *

c=2+2j

print(exp(c))

print(log(c,2))

print(sqrt(c))
```

Output:-

```
(-3.074932320639359+6.71884969742825j)

(1.5000000000000002+1.1330900354567985j)

(1.5537739740300374+0.6435942529055826j)
```

Decimal module:

- These are just the floating point numbers with fixed decimal

points.

- We can create decimals from integers, strings, floats or tuples.
- A decimal instance can represent any number exactly, round up or down and supply a limit to the number of significant digits.

Example:

```
from decimal import Decimal

print(Decimal(121))

print(Decimal(0.05))

print(Decimal('0.15'))

print(Decimal('0.012')+Decimal('0.2'))

print(Decimal(72)/Decimal(7))

print(Decimal(2).sqrt())
```

Output:

```
121
```

```
0.05000000000000000277555756156289135105907917022705078125

0.15

0.212

10.28571428571428571428571429

1.414213562373095048801688724
```

Fractions module:

- A fraction is a number which represents a whole number being divided into multiple parts. Python fraction module allows us to manage fractions in our python programs

Example:

```
import fractions

for num,decimal in [(3,2),(2,5),(30,4)]:

    fract=fractions.Fraction(num,decimal)

    print(fract)
```

Output:

```
3/2

2/5

15/2
```

Random module:

- Sometimes we want the computer to pick a random number in a given range or a list.
- To perform this random module is used. This function is not accessible directly, so we need to import random module and then we need to call this function using random static object.

Example:

```
import random

print(random.random())

print(random.randint(10,20))
```

***Output**:*

```
0.8709966760966288

16
```

Statistics module:

- It provides access to different statistics functions such as mean, median, mode, standard deviation.

Example:

```
import statistics

print(statistics.mean([2,5,6,9]))

print(statistics.median([1,2,3,8,9]))

print(statistics.mode([2,5,3,2,8,3,9,4,2,5,6]))

print(statistics.stdev([1,1.5,2,2.5,3,3.5,4,4.5,5]))
```

Output:

```
5.5

3

2

1.3693063937629153
```

Functional Programming Modules:

- This modules provide ***functions and classes that support a functional programming style and general operations on callable***.
- Following modules are used here

1. Itertools
2. Functools
3. operator

<u>Itertools Module</u>:

- It provide us various ways to manipulate the sequence while we are traversing it.
- Python itertools chain() function just accepts multiple iterable and return a single sequence as if all items belongs to that sequence.

Example:

```
from itertools import *
for value in chain([1.3,2.51,3.3],['c++','python','java']):
    print(value)
```

output:

```
1.3
```

```
2.51
3.3
c++
python
java
```

functools Module:

- It provide us various tools which allows and encourage us to write reusable code.
- Python functools partial() functions are used to replace existing functions with some arguments already passed in. It also creates new version of the function in a well documented manner.

  ***Example**:*

  Suppose we have a function called multiplier which just multiplies two numbers. Its definition looks like:

  ```
  def multiplier(x,y):
          return x*y
  ```

  Now if we want to make some dedicated functions to double or triple a number we will have to define new functions as:

```
def multiplier(x,y):

        return x*y

def doubleIt(x)

        return multiplier(x,2)

def tripleIt(x):

        return multiplier(x,3)
```

But what happens when we need 1000 such functions? Here we can use partial functions:

```
from functools import partial

def multiplier(x.y):

        return x*y

double=partial(multiplier,y=2)

triple=partial(multiplier,y=3)

print(“Double of 5 is {}’.format(double(5)))

print(“Triple of 5 is {}’.format(triple(5)))
```

Output:

```
Double of 5 is 10

Triple of 5 is 15
```

<u>Operator Module:</u>

The operator module supplies functions that are equivalent to python's operators. These functions are handy in cases where callables must be stored, passed as arguments or returned as function results.

Function supplied by operator module are

abs(a)-absolute	add(a,b)-addition	and_(a,b)- logical and
div(a,b)- division	eq(a,b)- equal to	gt(a,b)- greater than
le(a,b)- less than	mod(a,b)- modulus	mul(a,b)-multiplication
or_(a,b)- logical or	not_(a)- logical not	repeat(a,b)- a*b
xor(a,b)- logical xor		

Namespace and Scoping:

- A ***namespace*** is a system to have a ***unique name*** for each and every object in python.
- An object might be a ***variable or a method***.
- Python itself maintains a namespace in the form of a python dictionary.

- Python interpreter understands what exact method or variable one is trying to point to in the code, depending upon the namespace.
- ***Name means an unique identifier*** while ***space talks something related to scope***.
- ***Name*** might be any ***python method or variable*** and ***space*** depends upon the ***location from where is trying to access*** a variable or a method.

Types of Namespace:

- ***Local Namespace***: This namespace covers the ***local names inside a function***. Python creates this namespace for every function called in a program.
- ***Global Namespace***: This namespace covers the ***names from various imported modules used in a project***. Python creates this namespace for every module included in the program.

- ***Built-in Namespace***: This namespace covers the ***built-in functions and built-in exception names***. Python creates it as the interpreter starts and keeps it until we exit.

Python Variable Scoping:

- ***Scope is the portion of the program from where a namespace can be accessed directly without any prefix***.
- At any given moment there are atleast following three nested scope.

1. ***Scope of the current function which has local names.***
2. ***Scope of the module which has global names.***
3. ***Outermost scope which has built-in names***.

- When a reference is made inside a function the name is searched in the local namespace, then in the global namespace and finally in the built-in namespace.
- If there is a ***function inside another function***, ***a new scope is nested inside the local scope***. Python has ***two scopes***
- ***Local scope variable-*** All variables which are assigned inside a function.
- ***Global scope variable-***All those variables which are outside the function termed

```
# Example pf Global and Local Scope

global_var=30     # global scope

def scope():

    local_var=40  #local scope

    print(global_var)

    print(local_var)

scope()

print(global_var)
```

Output:

```
30

40

30
```

4.4 Python Packages:

Introduction-

- Suppose we have developed a very ***large application that includes many modules***. As the number of modules grows***, it becomes difficult to keep track*** of them as they have similar names or functionality.
- It is necessary to ***group and organize them*** by some mean which can be ***achieved by packages***.
- ***A package is a hierarchical file directory structure that defines a single python application environment that comprises of modules and subpackages and sub-subpackages and so on.***
- Packages allow for a hierarchical structuring of the module namespace ***using dot notation***.
- Packages are a ***way of structuring many packages and modules*** which help in a ***well organized hierarchy of data set, making the directories and modules easy to access***.
- ***A package is a collection of python modules*** i.e. a package is a directory of python modules containing an additional _ _init_ _.py file. E.g Phone/_ _init_ _.py

Writing Python Packages-

- ***To create a package in Python, we need to follow these three simple steps*:**

1. First, we create a directory and give it a package name, preferably related to its operation.
2. Then we put the classes and the required functions in it.
3. Finally we create an __init__.py file inside the directory, to let Python know that the directory is a package.

Example to write Python Packages-

- Create a directory named mypkg and create a __init__.py file and save in the mypkg directory so that mypkg will be considered as package.
- Create file p1.py in the mypkg directory and add this code

```
def m1():
   print("First Module")
```

- Create file p2.py in the mypkg directory and add this code

```
def m2():
   print("second module")
```

- Create file pkgsample.py and add this code

```
from mypkg import p1,p2
```

```
p1.m1()

p2.m2()
```

Output-

```
First Module

second module
```

Using Standard Packages-

Math:

- Some of the most popular mathematical functions are defined in the math module. These includes ***trigonometric functions, representation functions, logarithmic functions and angle conversion function***s.
- Two mathematical constants are also defined in math module.

- **Pie**(∏) is a well known mathematical constant which is defined as the ratio of the circumference to the diameter of a circle and its value is 3.141592653589793

```
import math

print(math.pi)
```

o/p

```
3.141592653589793
```

Another well known mathematical constant defined in the math module is **e**.

It is called Euler’s number and it is a base of the natural Logarithm. It value is 2.718281828459045

e.g

```
import math

print(math.e)
```

o/p

```
2.718281828459045
```

NumPy:

- NumPy is the fundamental package for ***scientific computing*** with python.
- ***NumPy stands for "Numeric Python***".
- It provides a ***high performance multidimensional array object*** and ***tools for working with these arrays***.
- An array is ***a table of elements*** (usually numbers) ***all of the same type, indexed by a tuple of positive integers and represented by a single variable***. NumPy's array class is called **ndarray**. It is also known by the **alias array**.
- In NumPy arrays the individual data items are called **elements**. Dimensions are called **axes**.
- The size of NumPy arrays are fixed , once created it cannot be changed again.
- NumPy arrays are great alternatives to python lists.

a two-dimensional array.

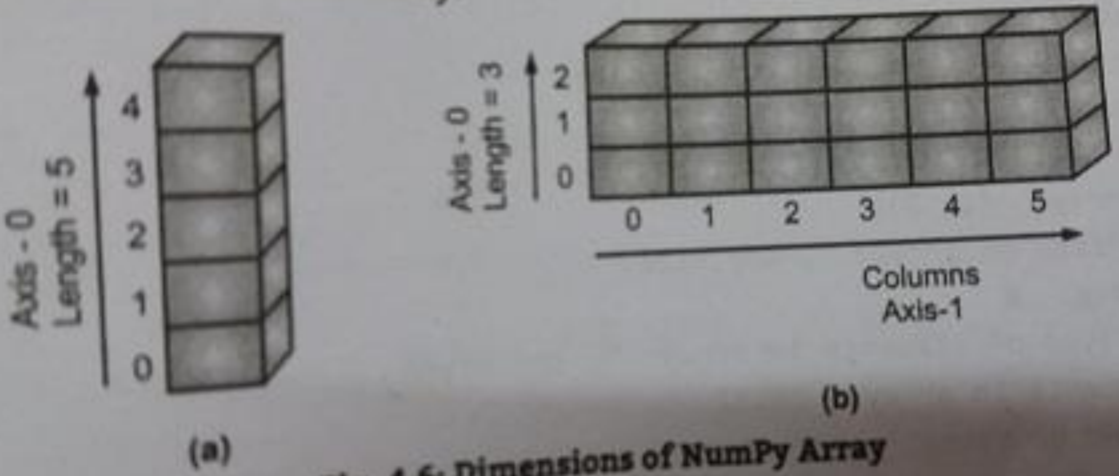


Fig. 4.6: Dimensions of NumPy Array

- A one dimensional array has ***one axis indicated by Axis-0***. That axis has **five elements** in it, so we say it has ***length*** of **five**.
- A ***two dimensional array*** is made up of ***rows and columns***. All ***rows are indicated by Axis-0*** and all ***columns are indicated by Axis-1***. If Axis-0 in two dimensional array has three elements, so its length is three and Axis-1 has six elements so its length is 6.
- ***Using NumPy , a developer can perform the following operations***:

1. Mathematical and logical operations on arrays.
2. Fourier transforms and routines for shape manipulation.
3. Operations related to linear algebra.

4. NumPy has in-built functions for linear algebra and random number generation.

Array Object:

- NumPy’s main object is the homogeneous multidimensional array. It is a table of elements(usually numbers) all of same type, indexed by a tuple of positive integers.
- In NumPy dimensons are called as Axes. NumPy’s array class is called **ndarray**. It is also known by the **alias array**.

Basic attributes of the ndarray class are as follow:

- **shape-** A tuple that specifies the number of elements for each dimension of array.
- **size-** The total number of elements in the array.
- **ndim-** Determines the dimension of an array.
- **nbytes-** Number of bytes used to store the data.
- **dtype-** Determines the datatype of elements stored in array.

Example:

```
import numpy as np

a=np.array([10,20,30])

print(a)

arr=np.array([[1,2,3],[4,5,6]])

print(arr)

type(arr)   # <class ‘numpy.ndarray’>

print("No. of dimension: ",arr.ndim)   #2

print("Shape of array: ",arr.shape)    #(2,3)

print("Size of array: ",arr.size)       # 6

print("Type of elements in array: ",arr.dtype)  # int32

print("No. of bytes: ",arr.nbytes)   # 24
```

Basic Array Operations:

- In NumPy array allows a ***wide range of operations*** which can be ***performed on a particular array*** or a combination of Arrays.
- These ***operations include some basic mathematical operation*** as well as ***Unary and Binary operations***. In case of **+=,-=, *= operators**, the existing array is modified.

1. **Unary operators**: Many unary operations are provided as a method of ndarray class. This includes **sum, min, max** etc. These ***functions can also be applied row-wise or column-wise*** by ***setting an axis parameter***.
2. **Binary Operators**: These operations apply on array elementwise and a new array is created. You can use all basic arithmetic operators like +,-,/, * etc. In case of +=, -=, *= operators existing array is modified.

Example:

```
import numpy as np

arr1=np.array([1,2,3,4,5])

arr2=np.array([2,3,4,5,6])

print(arr1)

print("Add 1 in each element ",arr1+1)  # [2 3 4 5 6]

print("Subtract 1 from each element: ",arr-1)  #[0 1 2 3 4]

print("Multiply 10 with each element in array: ",arr1*10) #[10 20
30 40 50]

print("Sum of all array elements: ",arr1.sum())  # 15
print("Array sum= ",arr1+arr2)  # [3 5 7 9 11]

print("Largest element in array: ",arr1.max())  # 5
```

Reshaping of Array:

- We can also perform reshape operation using python numpy operation.
- Reshape is when you change the number of rows and columns which gives a new view to an object.

Example:

```
import numpy as np

arr=np.array([[1,2,3],[4,5,6]])

a=arr.reshape(3,2)

print(a)
```

Output:

```
([[1,2],

  [3,4],

  [5,6]])
```

Slicing of Array:

- Slicing is basically extracting particular set of elements from an array. Consider an array([(1,2,3,4),(5,6,7,8)])
- Here the array(1,2,3,4) is at index 0 and (5,6,7,8) is at index 1 of the python numpy array. We need a particular element (say3) out of a given array.

e.g.

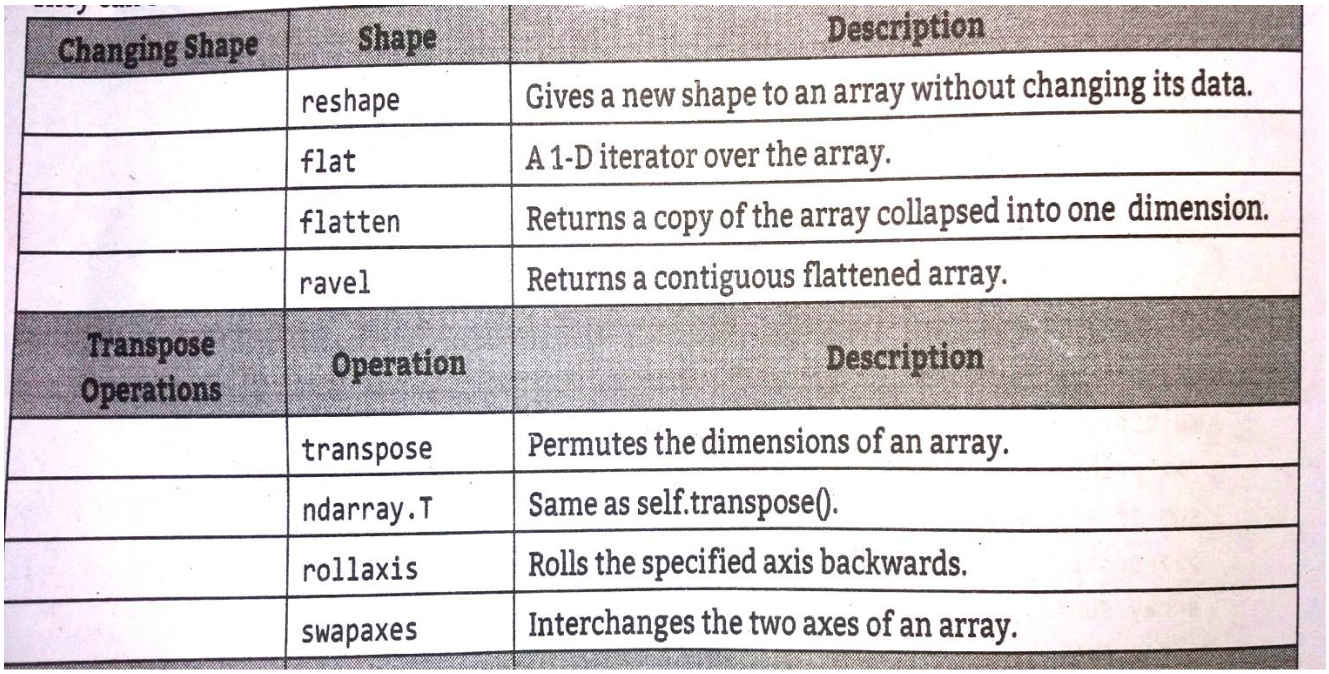
```
import numpy as np

a=np.array([(1,2,3,4),(5,6,7,8)])

print(a[0,2])   # 3
```

e.g. When we need 2nd element from 0 and 1 index of the array

```
import numpy as np

a=np.array([(1,2,3,4),(5,6,7,8)])

print(a[0:,2])     # [3 7]
```

Array Manipulation Functions:

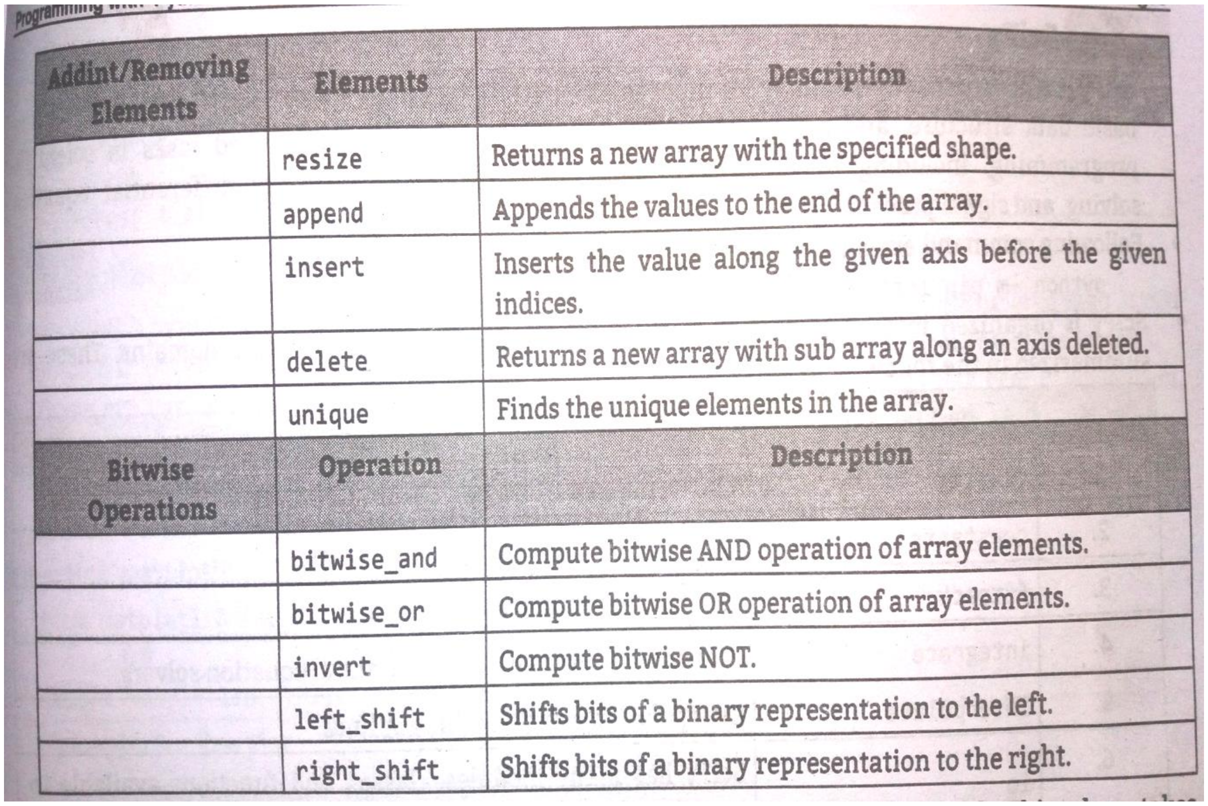
- Several routines are available in NUmPy package for manipulation of elements in ndarray object. They can be classified into the following types.



Changing Shape	Shape	Description
	`reshape`	Gives a new shape to an array without changing its data.
	`flat`	A 1-D iterator over the array.
	`flatten`	Returns a copy of the array collapsed into one dimension.
	`ravel`	Returns a contiguous flattened array.
Transpose Operations	**Operation**	**Description**
	`transpose`	Permutes the dimensions of an array.
	`ndarray.T`	Same as self.transpose().
	`rollaxis`	Rolls the specified axis backwards.
	`swapaxes`	Interchanges the two axes of an array.

Changing Dimensions	Dimension	Description
	`broadcast`	Produces an object that mimics broadcasting.
	`broadcast_to`	Broadcasts an array to a new shape.
	`expand_dims`	Expands the shape of an array.
	`squeeze`	Removes single dimensional entries from the shape of an array.
Joining Arrays	**Array**	**Description**
	`concatenate`	Joins a sequence of arrays along an existing axis.
	`stack`	Joins a sequence of arrays along a new axis.
	`hstack`	Stacks arrays in sequence horizontally (column wise).
	`vstack`	Stacks arrays in sequence vertically (row wise).
Splitting Arrays	**Array**	**Description**
	`split`	Splits an array into multiple sub-arrays.
	`hsplit`	Splits an array into multiple sub-arrays horizontally (column wise).
	`vsplit`	Splits an array into multiple sub-arrays vertically (row wise).

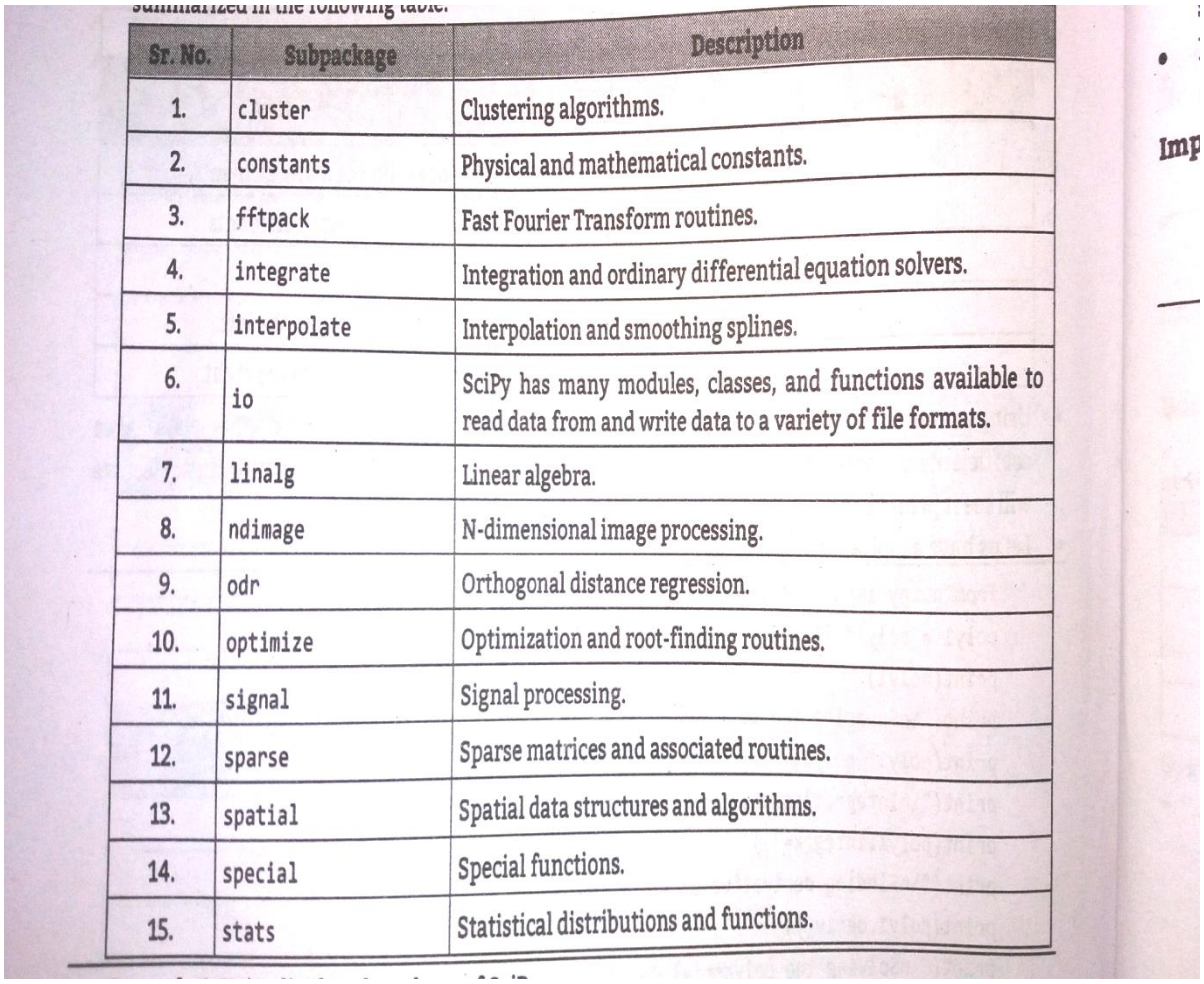
contd...



Addint/Removing Elements	Elements	Description
	resize	Returns a new array with the specified shape.
	append	Appends the values to the end of the array.
	insert	Inserts the value along the given axis before the given indices.
	delete	Returns a new array with sub array along an axis deleted.
	unique	Finds the unique elements in the array.
Bitwise Operations	**Operation**	**Description**
	bitwise_and	Compute bitwise AND operation of array elements.
	bitwise_or	Compute bitwise OR operation of array elements.
	invert	Compute bitwise NOT.
	left_shift	Shifts bits of a binary representation to the left.
	right_shift	Shifts bits of a binary representation to the right.

SciPy:

- ***SciPy is a library that uses NumPy for more mathematical functions***.
- SciPy uses NumPy arrays as the basic data structure and come with modules for various ***commonly used tasks in scientific programming, including linear algebra, integration (calculus), ordinary differential equation solving and signal processing***.
- SciPy is organized into subpackage covering different scientific computing domains. These are as given …



Sr. No.	Subpackage	Description
1.	cluster	Clustering algorithms.
2.	constants	Physical and mathematical constants.
3.	fftpack	Fast Fourier Transform routines.
4.	integrate	Integration and ordinary differential equation solvers.
5.	interpolate	Interpolation and smoothing splines.
6.	io	SciPy has many modules, classes, and functions available to read data from and write data to a variety of file formats.
7.	linalg	Linear algebra.
8.	ndimage	N-dimensional image processing.
9.	odr	Orthogonal distance regression.
10.	optimize	Optimization and root-finding routines.
11.	signal	Signal processing.
12.	sparse	Sparse matrices and associated routines.
13.	spatial	Spatial data structures and algorithms.
14.	special	Special functions.
15.	stats	Statistical distributions and functions.

e.g. using linalg sub package of SciPy

```
import numpy as np

from scipy import linalg

a=np.array([[1.,2.],[3.,4.]])

linalg.inv(a)  # find inverse of array
```

o/p

```
array([[-2.,1.],[1.5,-0.5]])
```

e.g. using linalg sub package of SciPy

```
import numpy as np

from scipy import linalg

a=np.array([[1,2,3],[4,5,6],[7,8,9]])

linalg.det(a)  # find determinant of array
```

o/p

```
0.0
```

e.g. Using special sub package of SciPy

```
From scipy.special import cbrt

cb=cbrt([81,64])

print(cb)        # find cube root
```

o/p

```
array([4.32674871, 4.    ])
```

Matplotlib:

- ***It is a plotting library used for 2D graphics in python programming language.***
- It can be used in ***python scripts, shell, web application servers*** and other ***graphical user interface***.
- There are various plots which can be created using python matplotlib like bar graph, histogram, scatter plot, area plot, pie plot.

e.g.

```
import matplotlib.pyplot as plt

x=[1,2,3,4,5,6,7]

y=[1,2,3,4,5,6,7]

plt.plot(x,y)

plt.show()
```

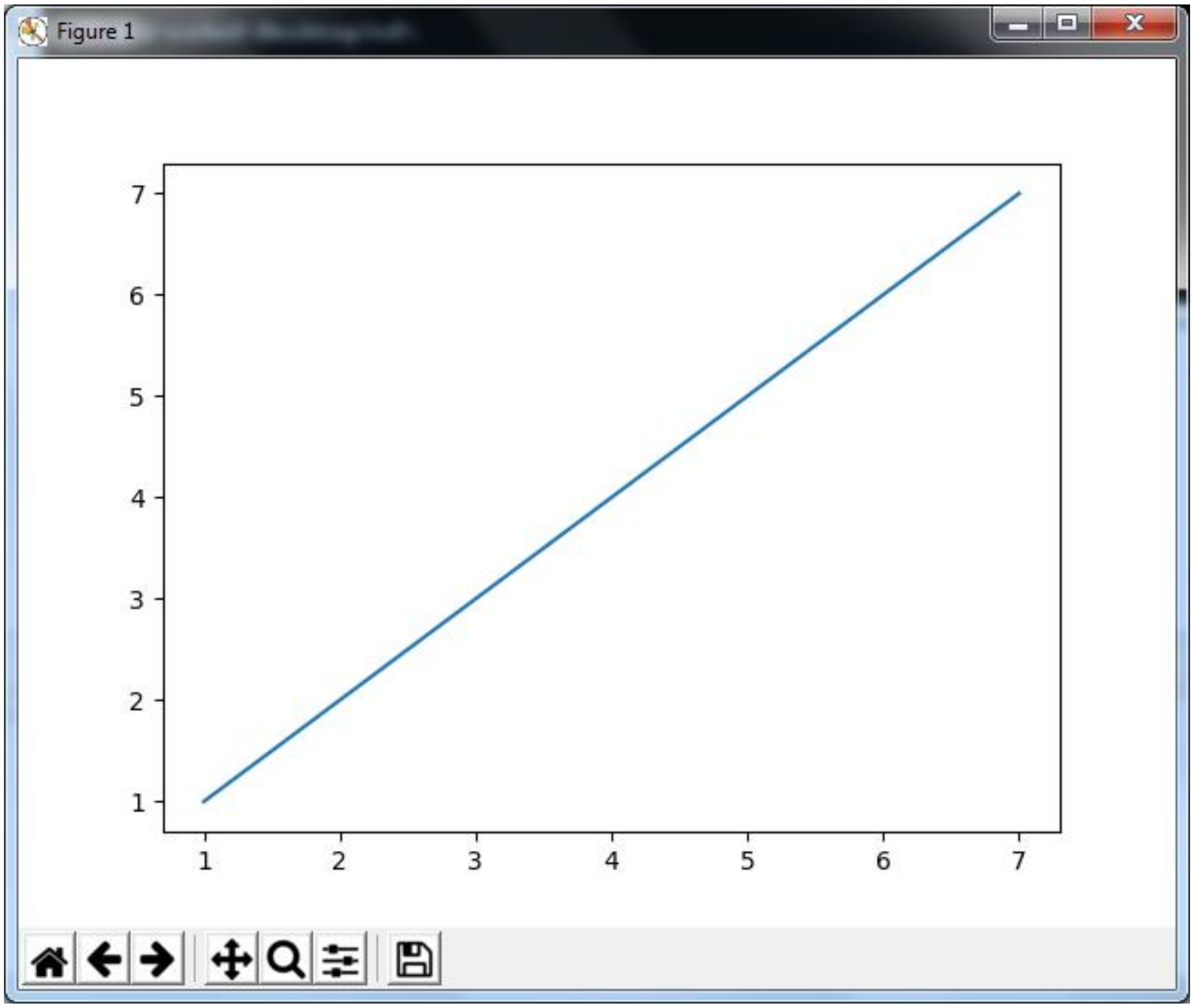
e.g.

```
import matplotlib.pyplot as plt

x=[1,2,3,4,5,6,7]

y=[1,2,3,4,5,6,7]
```

plt.plot(x,y)

plt.show()

output:



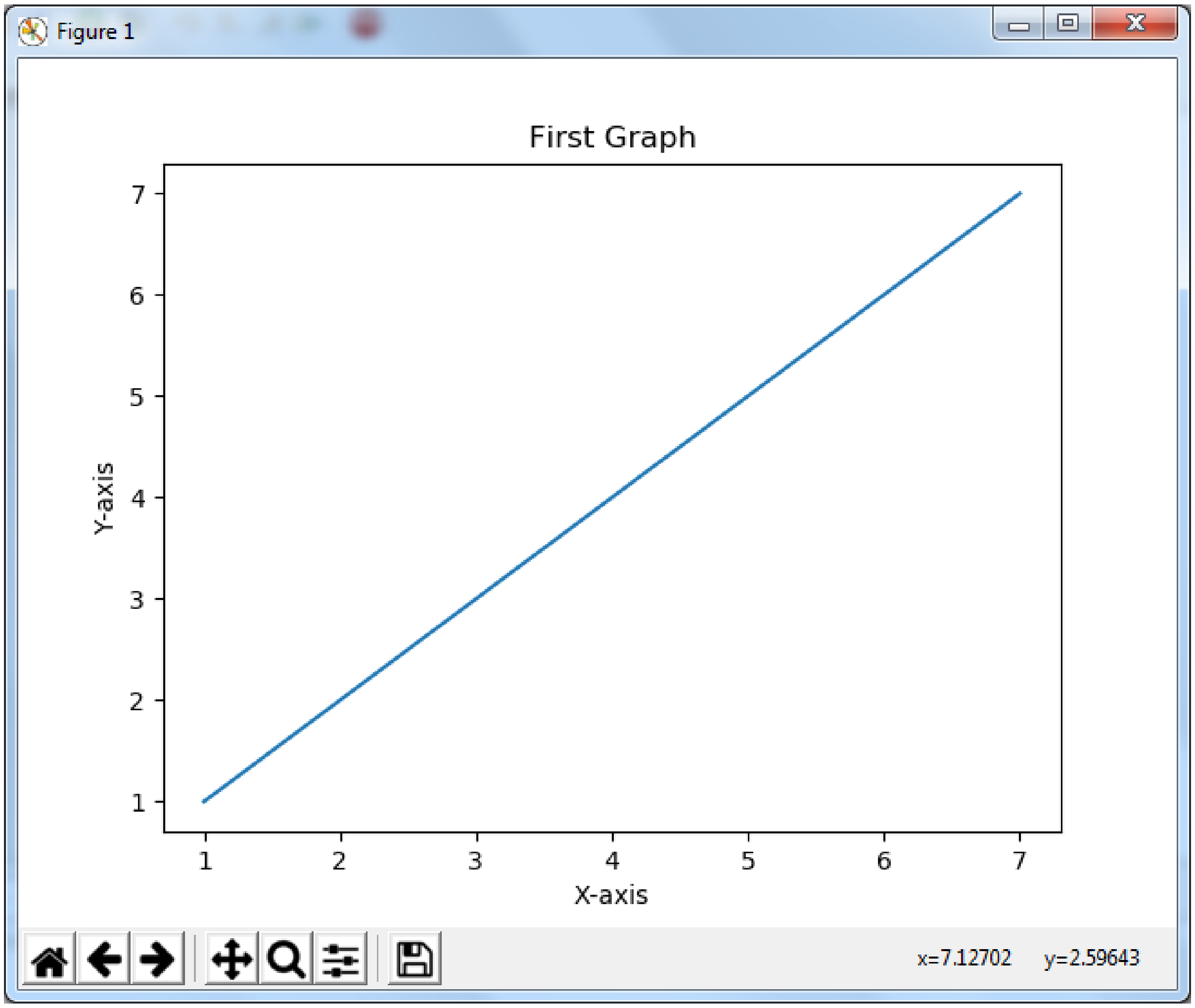
- In the above example we have taken two datasets namely x and y.
- The first line of code is import statement.

- After import statement we can easily use the functions provided by the matplotlib.
- The plot function takes the value from x and y datasets and plot the graph for the same. While providing the dataset you need to remember that the number of elements in x and y dataset should be equal.
- Last statement is using the show() function in order to display graph on the monitor.

Example-

```
import matplotlib.pyplot as plt

x=[1,2,3,4,5,6,7]

y=[1,2,3,4,5,6,7]

plt.plot(x,y)

plt.xlabel(‘X-axis’)

plt.ylabel(‘Y-axis’)

plt.title(‘First Graph’)

plt.show()
```

output:



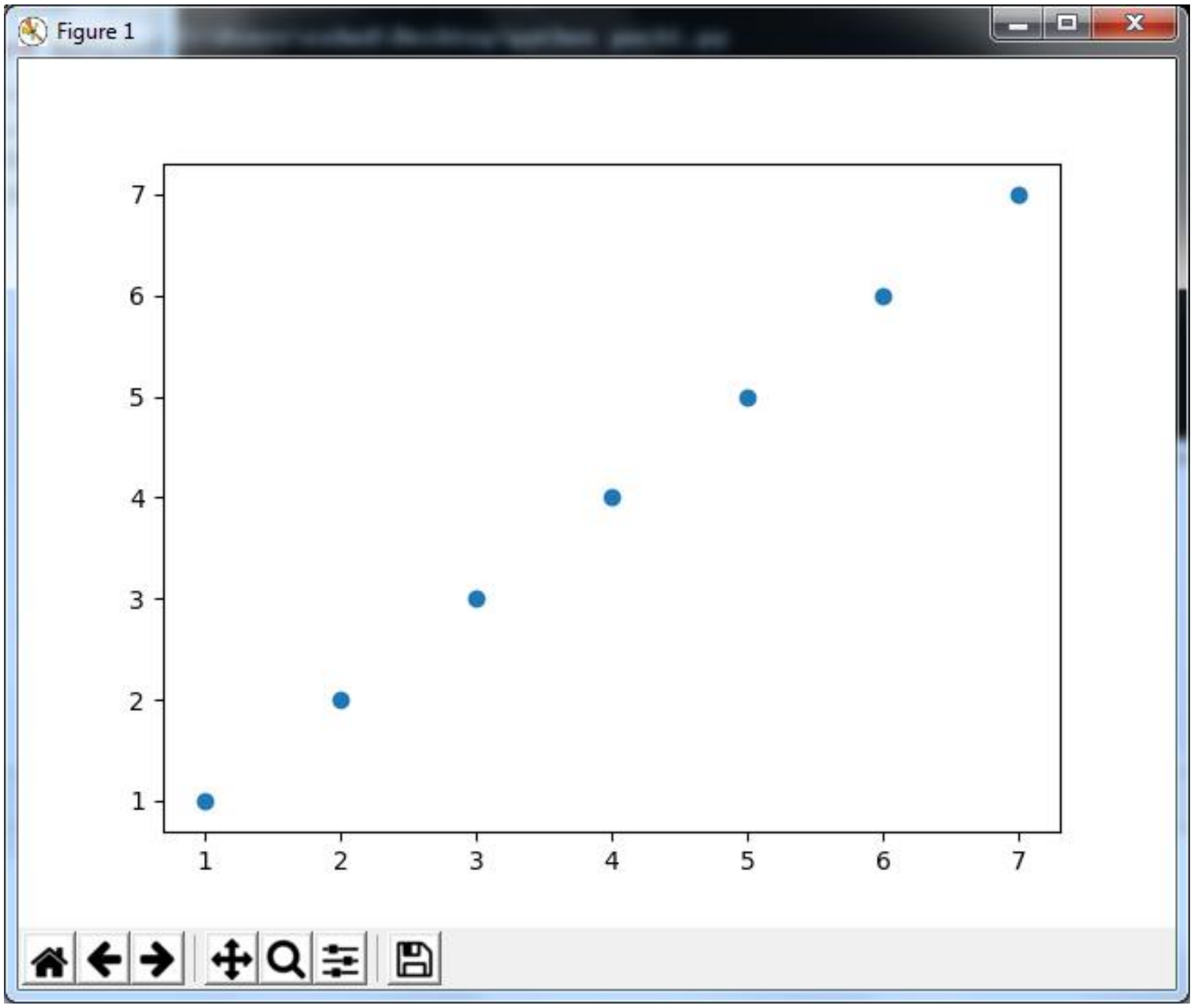
Scatter Plot:

- This graph will only mark the point that is being plotted in the graph.
- The graph can be plotted using scatter() function.

e.g.

```
import matplotlib.pyplot as plt

x=[1,2,3,4,5,6,7]

y=[1,2,3,4,5,6,7]

plt.scatter(x,y)

plt.show()
```

output:



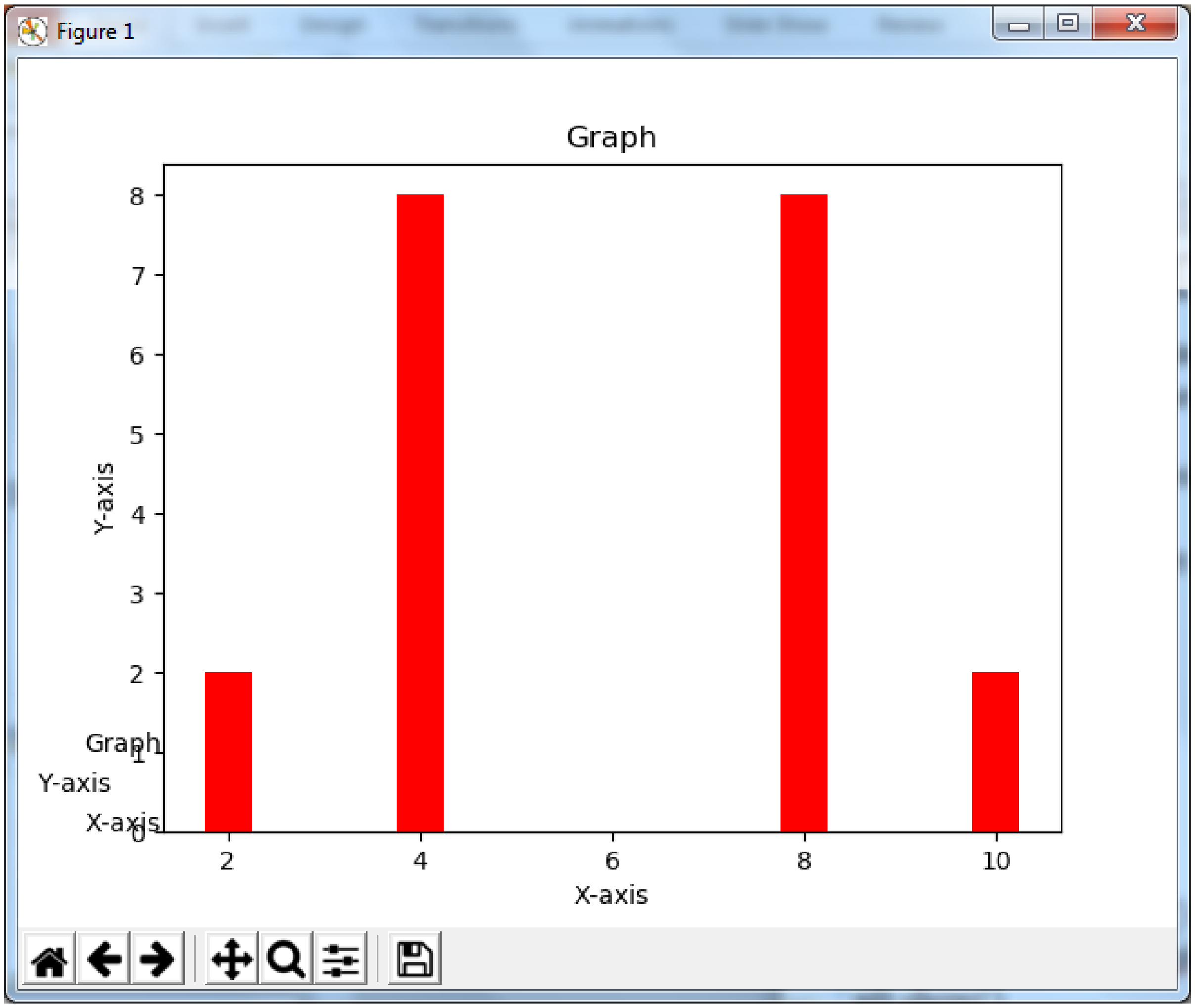
Bar Graph:

- 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.

e.g.

```
from matplotlib import pyplot as plt

x=[2,4,8,10]

y=[2,8,8,2]

plt.xlabel('X-axis')

plt.text(0.5,0,'X-axis')

plt.ylabel('Y-axis')

plt.text(0,0.5,'Y-axis')

plt.title('Graph')

plt.text(0.5,1.0,'Graph')

plt.bar(x,y,label="Graph",color='r',width=0.5)

plt.show()
```

output:



# Program to display Horizontal bar

import numpy as np

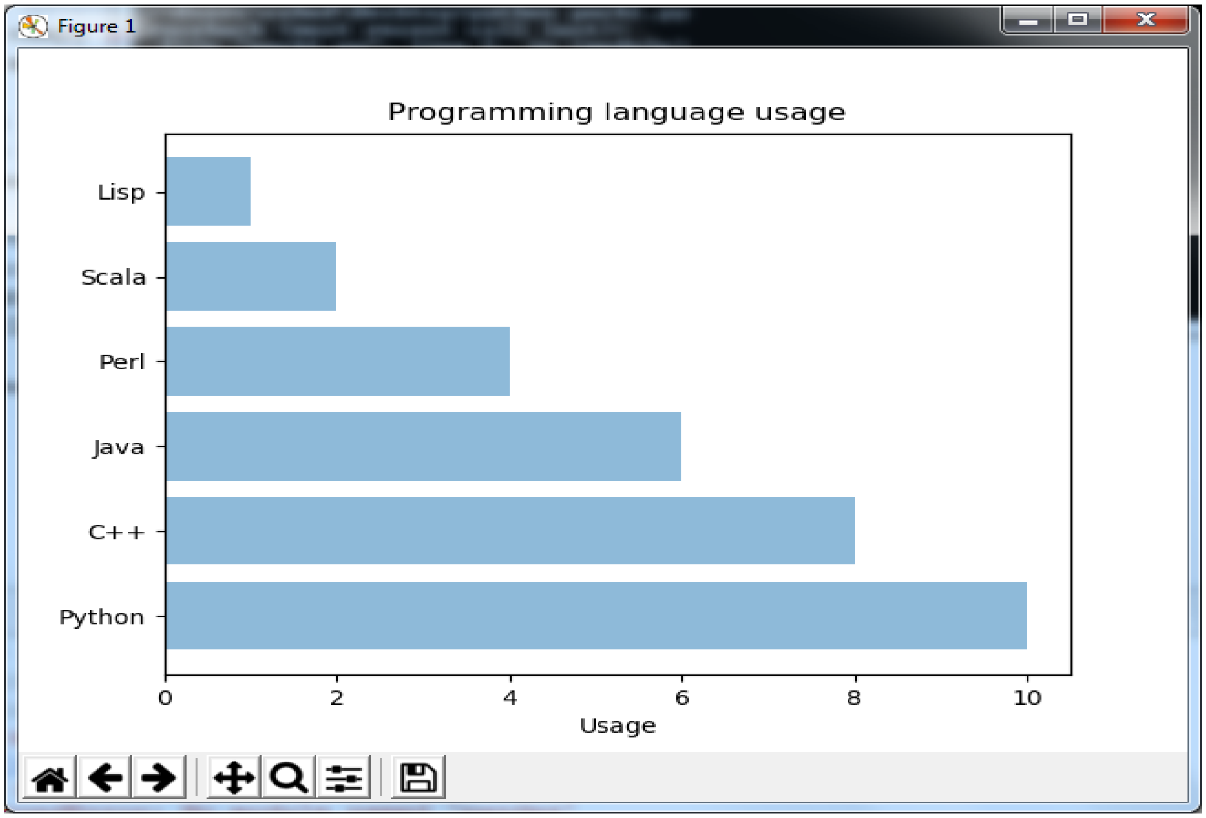
import matplotlib.pyplot as plt

objects = ('Python', 'C++', 'Java', 'Perl', 'Scala', 'Lisp')

y_pos = np.arange(len(objects))

```
performance = [10,8,6,4,2,1]

plt.barh(y_pos, performance, align='center', alpha=0.5)

plt.yticks(y_pos, objects)

plt.xlabel('Usage')

plt.title('Programming language usage')

plt.show()
```

Output-



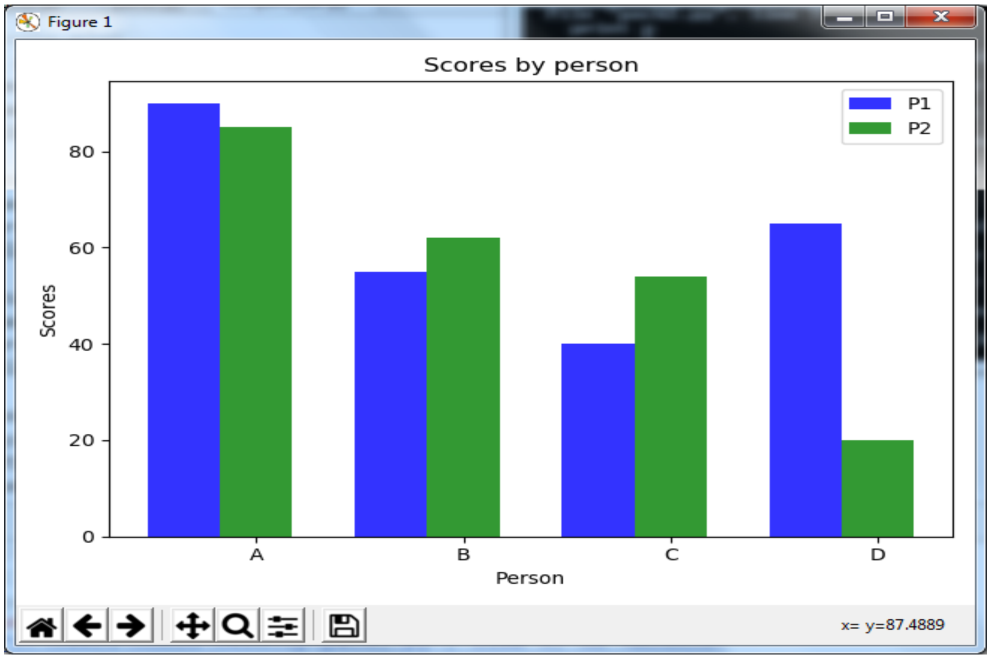
- NumPy **arange()** is an inbuilt numpy function that returns a ndarray object containing evenly spaced values within the given range. The arange() returns an array with evenly spaced elements as per the interval. The interval mentioned is half opened i.e. [Start, Stop)

Example:

```
import numpy as np

import matplotlib.pyplot as plt

# data to plot

n_groups = 4

means_P1 = (90, 55, 40, 65)

means_P2 = (85, 62, 54, 20)

# create plot

index = np.arange(n_groups)

bar_width = 0.35

opacity = 0.8

rects1 = plt.bar(index, means_P1,
bar_width,alpha=opacity,color='b',label='P1')

rects2 = plt.bar(index + bar_width, means_P2,
bar_width,alpha=opacity,color='g',label='P2')
```

```
plt.xlabel('Person')

plt.ylabel('Scores')

plt.title('Scores by person')

plt.xticks(index + bar_width, ('A', 'B', 'C', 'D'))

plt.legend()

plt.tight_layout()

plt.show()
```

Output-



Pandas:

- It is an ***open source python library providing high performance data manipulation*** and analysis tool using its powerful data structures.
- Pandas is python library which is ***used for data analysis purpose***.
- In pandas data can be represented in 3 ways:

1. Series
2. Data frames
3. Panel

Series-

It is ***one dimensional array like structure defined in pandas.***

It can be ***used to store the data of any type***.

e.g.

```
import pandas as pd

data=[1.1,2.1,3.1,4.1,5.1,6.1,7.1,8.1]

df=pd.Series(data)

print('The series is:\n',df)
print('The type is:',type(df))
```

Output-

```
The series is:
0   1.1
1   2.2
2   3.1
3   4.1
4   5.1
5   6.1
6   7.1
7   8.1
dtype: float64
The type is: <class ‘pandas.core.series.Series’>
```

DataFrame-

- Dataframes are ***two dimensional structures*** which ***consist of columns and rows***.
- It can be used to ***store the data of any type***.

e.g.

```
import pandas as pd

dict1={'a':1.1,'b':2.1,'c':3.1,'d':4.1}

dict2={'a':5.1,'b':6.1,'c':7.1,'d':8.1}

data={'Col 1':dict1,'Col 2':dict2}

df=pd.DataFrame(data)

print(df)
```

Output-

```
     Col1 Col2

a    1.1       5.1

b    2.1       6.1

c    3.1       7.1

d    4.1       8.1
```

Example:-

```
import pandas as pd

s1=pd.Series([1,3,4,5,6,2,9])
```

```
s2=pd.Series([1.1,3.5,4.7,5.8,2.9,9.3,8.9])

s3=pd.Series([‘a’,’b’,’c’,’d’,’e’,’f’,’g’])

data={‘first’:s1,’second’:s2,’third’:s3}

dfseries=pd.DataFrame(data)

print(dfseries)
```

Output-

```
      first    second     third

  0    1          1.1      a

  1    3          3.5      b

  2    4          4.7      c

   3    5         5.8      d

   4    6         2.9      e

   5    2         9.3      f

   6    9         8.9      g
```

Panel-

- Panel is a three dimensional data structure with homogeneous data.
- It is hard to represent the panel in graphical representation.
- But a panel can be illustrated as a container of DataFrame.

  It takes following arguments-

data: The data can be of any form like ndarray, list, dict, map, DataFrame

item: axis 0, each item corresponds to a dataframe contained inside.

major_axis: axis 1, it is the index (rows) of each of DataFrames.

minor_axis: axis 2, it is the column of DataFrames.

dtype: The data type of each column.

copy: It takes a Boolean value to specify whether or not to copy the data. The default value is false.

***Example 1*: # creating an empty panel**

```
import pandas as pd

import numpy as np
```

```
data = np.random.rand(2,4,5)

p = pd.Panel(data)

print p
```

Output:

```
<class 'pandas.core.panel.Panel'>

Dimensions: 2 (items) x 4 (major_axis) x 5 (minor_axis)

Items axis: 0 to 1

Major_axis axis: 0 to 3

Minor_axis axis: 0 to 4
```

***Example 2:* #creating an empty panel**

```
import pandas as pd

import numpy as np

data = {'Item1':pd.DataFrame(np.random.randn(4, 3)), 'Item2' :
pd.DataFrame(np.random.randn(4, 2))}

p = pd.Panel(data)

print(p)
```

Output:

```
Dimensions: 2 (items) x 4 (major_axis) x 3 (minor_axis)

Items axis: Item1 to Item2

Major_axis axis: 0 to 3

Minor_axis axis: 0 to 2
```

- **random**. **randn**() function: This function return a sample (or samples) from the “standard normal” distribution. A single float **randomly** sampled from the distribution is returned if no argument is provided. This is a convenience function. If you want an interface that takes a tuple as the first argument, use numpy.

User defined Packages:

- We organize a large number of files in different folders and subfolders based on some criteria, so that we can find and manage them easily.
- In the same way a package in python takes the concept of the modular approach to next logical level.

- A package can contain one or more relevant modules.
- Physically a package is actually a folder containing one or more module files.

Example to create and access the packages-

- Create a directory named mypkg1 and create a __init__.py file and save in the mypkg1 directory so that mypkg1 will be considered as package.
- Create modules message.py and mathematics.py with following code:

message.py

```
def sayhello(name):
   print("Hello " + name)
   return
```

mathematics.py

```
def sum(x,y):
   return x+y
def average(x,y):
   return (x+y)/2
```

```
def power(x,y):
    return x**y
```

Create p1.py with the code given

p1.py

```
from mypkg1 import mathematics
from mypkg1 import message
message.sayhello("Amit")
x=mathematics.power(3,2)
print("power(3,2) :",x)
```

Output

```
Hello Amit
power(3,2) : 9
```

- Normally __init__.py file is kept empty. However it can be used to choose specific functions from modules in the package folder and make them available for import.

__init__.py

```
from .mathematics import average,power

from .message import sayhello
```

- Create a test.py file and the specified functions in __init_.py can now be imported in the interpreter session or another executable script.

test.py

```
from mypkg1 import power,average,sayhello

sayhello("Amol")

x=power(3,2)

print("power(3,2): ",x)
```

Output-

```
Hello Amol

power(3,2):  9
```