Introduction to Python: Python is an interpreted, object oriented, high-level programming language with dynamic semantics.

Python syntax are easy compared to other programming languages.

Python was introduced by Guido Van Rossum in 1989, 1st version of python was released in 1991.

In python there is no type declaration of variables, parameters, functions or methods in source code. This makes the code short and flexible. Programmers can express logical concepts in fewer lines of code in comparison to other programming languages such as C, C++, C# or Java.

Features of python programming language:

- Smart indent and auto compilation.
- Easy and open source.
- Object oriented language.
- Interpreted language.
- Integrated and Embeddable language.
- Dynamic memory allocation.
- Platform-independent language.
- Expressive language.
- Multi-paradigm support.
- Large standard library.
- Dynamically typed language.
- Frontend and backend development.
- GUI programming support.

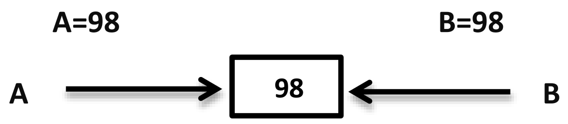
Applications of python programming:

Networking programming, Data analysis, Robotics, Desktop application, Games development, Scientific calculation, Machine learning & artificial intelligence etc.

Command prompt programs run ➡ python programName.py ↵ Saves programs with .py extension.

Variables: A variable is a name of the memory location. It is the container which holds the value while the python program is executed.

In python variables are a symbolic name that is a reference or pointer to an object. Once an object is assigned to a variable, we can refer to the object by that name.



The variables A and B refers to the same object because python does not create another object of same value variables.

Operators: An operator is a special symbol which is used to perform operations on operands.

Python has seven types of operators:

Arithmetic operators	+ , - , * , / , % , **(Exponentiation) , //(floor division)
Assignment operators	= , += , -= , *= , /= , //= , ** , %= , & , |=
Comparison operators	== , != , > , < , >= , <=
Logical operators	and , or , not
Bitwise operators	& , | , ^(Bitwise XOR) , ~(Bitwise not) , >>(Right shift), <<(Left shift)
Membership operators	in , not in
Identity operators	is , is not

➢ Arithmetic operators

- **Exponentiation operator (**)** perform exponential calculation like (2**5) translates to (2*2*2*2*2). Ex: print("Value=" ,5**3) Output: **value= 125**

- **Floor division operator** (//) returns the integer part of the division operation.

➢ **Assignment operators** are used to assign values to variables. (x *= y) same as (x = x+y)

➢ **Comparison operators** are to compare two values and returns a Boolean value, either True or False.

- **Logical operators are used to combine conditional statements. It returns a Boolean value either True or False.**

Ex:

```
x=77
y=89
z=78
print(x<88 and y>78 and z<=78)
print(not x==y)
```

```
Output:
True
True
```

- **Bitwise operators: Bitwise operators are used to perform bitwise calculations on integers. The integers are first converted into binary and then operations are performed on bit by bit and result is retuned in decimal format.**

Note: Python Bitwise operators work only on integers.

- **Bitwise AND operator: It returns 1 if both bits are 1 else it returns 0.**

Ex:

A	**0**	**0**	**1**	**1**	
B	**0**	**1**	**0**	**1**	
A & B	**0**	**0**	**0**	**1**	**value= 1**

- **Bitwise OR operator: It returns 1 if either of the bit is 1 else it returns 0.**

Ex:

A	**0**	**0**	**1**	**1**	
B	**0**	**1**	**0**	**1**	
A | B	**0**	**1**	**1**	**1**	**value= 7**

- **Bitwise XOR operator: It returns 1, if one of the bit is 1 and the another bit is 0, else it returns 0.**

Ex:

A	**0**	**0**	**1**	**1**	
B	**0**	**1**	**0**	**1**	
A ^ B	**0**	**1**	**1**	**0**	**value= 6**

- **Bitwise NOT operator: It is unary operator that returns one's complement of the number.**

- **Membership operators** are used to test a sequence is present in an object. It returns Boolean value either True or False.

- **in operator:** It returns True if a sequence with the specified value is present in the object.

Ex:
```
val="Welcome"
L=["Apple","Banana","Mango"]
print('come' in val)
print('Banana' in L)
```

```
Output:

True

True
```

- **not in operator:** It returns true if a sequence with the specified value is not pre object.

Ex:
```
val="Welcome"
L=["Apple","Banana","Mango"]
print('Man' not in val)
print('Orange' not in L)
```

```
Output:

True

True
```

- **Identity operators** compare the memory locations of two objects. It returns Boolean value either True or False.

- **is operator:** It returns true if both variables are the same object.

Ex:
```
x=98
y=98
print(x is y)
```

Output: True

- **is not operator:** It returns true if bot variables are not the same object.

Ex:
```
x=98
y=97
print(x is not y)
```

Output: True

Data types

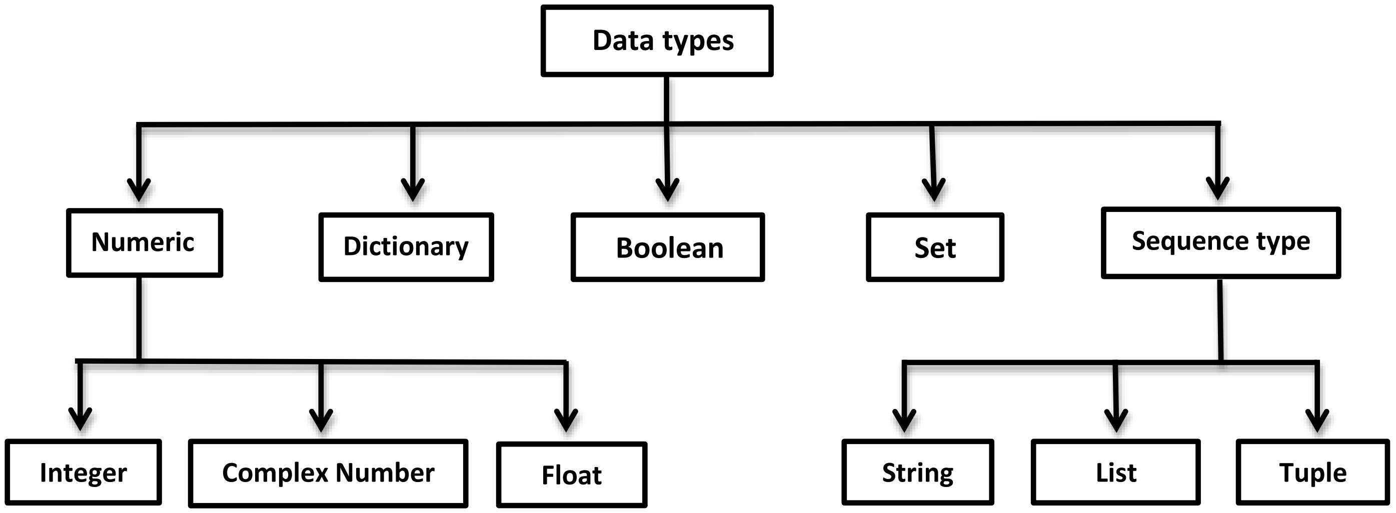
Data type determines the type and size of data associated with variables. Python is a dynamically typed language, hence we do not need to define the type of the variable while declaring it. Python interpreter automatically interpret variables type.

Python provides type() function that returns the type of the object passed.

Ex:
```
x=98

y=98.5
```

```
v=[97,98,101]

A=(97,98,101)

print(type(x),"\n",type(y),"\n",type(v),"\n",type(A)
```

```
Output:

<class 'int'>

<class 'float'>

<class 'list'>

<class 'tuple'>
```



Python data types can be distinguished into following major two types.

Mutable	**Immutable**
Mutable object can change its state or contents after creating it.	**Immutable object cannot change its state or contents after creating it.**
Classes that are mutable cannot considered final.	**Classes that are immutable can considered final.**
Thread unsafe.	**Thread safe.**
Mutable data types: List, Dictionary etc.	**Immutable data types: String, Tuple etc.**

* **Numeric data types:** The integer, float and complex values belong to a python numeric data type. Ex: integer → (x=6, y= -3) Float → (a=34.3, g=5.66) Complex → (x= 2+7j)

* **Dictionary: Dictionary is used to store data in key-value pair format. It is an unordered sequence of items. In dictionary we can store different types of data items such as string, tuple, list, integer or float. The items are stored in dictionary with key-value pair format, separated with comma enclosed within curly brackets.**
  Note: ⟶ Dictionary keys are case sensitive, it must be unique and single for each items.

Dictionary is mutable in python so we can change its contents after creating it.

Syntax:
```
Dictionary_Name={ ‘Key1’:”Value”,
                  ‘Key2’:”Value”,
                  ’Key3’:”Value” }
```

Ex:
```
D={1:"ONE",
    'Fruit':"Mango",
    'List':[95,97,98]
   }
 print(type(D),D)
```

```
Output:

<class 'dict'> {1: 'ONE', 'Fruit': 'Mango' , 'List': [95, 97, 98]}
```

* **Boolean: It stores either True or false.**

* **Set: Set is unordered sequence of items. In set we can store different types of data items such as string, float, integer etc. The items are stored in set separated with comma and enclosed within curly brackets. In set each item must be unique and single, sets removes the duplicate elements.**

  A set itself is mutable we can add or remove items from it.
  Syntax: Set_Name={ item1, item2, … itemN }
  Ex:
```
Set={98,97.88,"Python"}
print(type(Set),Set)
Set.add(101)
print(Set)
Set.remove(97.88)
print(Set)
```

```
Output:

<class 'set'> {97.88, 98, 'Python'}

{97.88, 98, 'Python', 101}

{98, 'Python', 101}
```

* **List: List is an ordered sequence of items. In list we can store different types of data items such as string, float, integer etc. The items are stored in list separated with comma and enclosed within square brackets.**
  List are mutable in python so we can change its contents after creating it.
  Syntax: List_Name=[item1, item2,…itemN]
  Ex:
```
List=[97,98,'Mango','Apple']
```

```
Output:

<class 'list'> [97, 98, 'Mango', 'Apple']

[98, 'Mango']

[97, 98, 'Mango', 'Apple', 97, 98, 'Mango', 'Apple']
```

```
print(type(List),List)
print(List[1:3])
print(List*2)
```

∗ **Tuple:** **Tuple is an ordered sequence of items. In tuple we can store different types of data items such as string, float, integer etc. The items are stored in tuple separated with comma and enclosed within parentheses.**
Tuple are immutable in python so we cannot change its contents after creating it.
Syntax: :   Tuple_Name=(item1, item2,…itemN)
Ex:

```
Tuple=(97,98,'Mango','Apple')
print(type(Tuple),Tuple)
print(Tuple[1:3])
print(Tuple*2)
```

```
Output:

<class 'tuple'> (97, 98, 'Mango', 'Apple')

(98, 'Mango')

(97, 98, 'Mango', 'Apple', 97, 98, 'Mango', 'Apple')
```

Difference between → **List** **and** **Tuple**

Lists are mutable.	Tuple are immutable.
List iteration is slower and is time consuming.	Tuple iteration is faster.
List consumes more memory.	Tuple consumes less memory.
Lists have several built-in methods.	Tuples have less built-in methods.
List operations are more error prone.	Tuple operations are less error prone.

∗ **String:** **String can be defined as sequence of characters represented in the quotation marks.**
In python we can use single quotes, double quotes and triple quotes to define a string.
The triple quotes can be used to represent multiline strings.
Strings are immutable in python so we cannot change its contents after creating it.

String indexing

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	
Left→	**W**	**E**	**L**	**C**	**O**	**M**	**E**		**T**	**O**		**P**	**Y**	**T**	**H**	**O**	**N**	**←Right**
	-17	-16	-15	-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1	

String slicing→ [**Start : Stop : StepSize**]

Ex:

```
Output:

<class 'str'> Welcome to python

Welcome , to python

Wloet yhn , nhy teolW
```

```
Str="Welcome to python"

print(type(Str),Str)

print(Str[0:7],",",Str[-9:-7],Str[11::])

print(Str[0::2],",",Str[-1::-2])
```

- **String implementation:**

1. **Print all string in three ways.**
2. **Extract last 5 characters.**
3. **Exclude last 5 characters.**
4. **Print all string with leave 2 characters.**
5. **Print reverse of string in 3 ways.**
6. **Print all string characters in reverse order at odd indexes.**
7. **Alternate the string from both sides.**

```
Str="Hello World"

print("1.",Str,",",Str[0::],",",Str[-11::])

print("2.",Str[-5::])

print("3.",Str[0:5])

print("4.",Str[0::3])

print("5.",Str[-1::-1],Str[10::-1],Str[::-1])

print("6.",Str[-1::-2])

print("7.",Str[::2],Str[-1::-2])
```

```
Output:

1. Hello World , Hello World , Hello World

2. World

3. Hello

4. HlWl

5. dlroW olleH dlroW olleH dlroW olleH

6. drWolH

7. HloWrd drWolH
```

- **String iteration:**

```
str="Hello World"

for i in range(len(str)):

   print(str[i],end="")
print()#For new line

for V in str:

   print(V,end="")

#Iteration in reverse order

print()#For new line

for V in str[-1::-1]:

   print(V,end="")

print()#For new line

#Alternate the string

for i in range(0,len(str),2):

   print(str[i],end=" ")
```

```
Output:

Hello World

Hello World

dlroW olleH

H l o W r d
```

- #Print all string characters without vowels.

```
str=input("Enter a string value= ")
print("All string characters without vowels: ",end="")
for V in str:
   i=V.lower()
   if i=='a'or i=='i'or i=='o' or i=='u' or i=='e':
      continue
   print(V,end="")
```

```
Output:

Enter a string value= AppleOrange

All string characters without vowels: pplrng
```

- #Find greatest from N numbers.

```
N=int(input("How many numbers do you want to enter="))

L=[]

for i in range(N):

   num=int(input("Enter {}th number= ".format(i+1)))

   L.append(num)

G=L[0]
```

```
Output:

How many numbers do you want to enter=5

Enter 1th number= 44

Enter 2th number= 56

Enter 3th number= 76

Enter 4th number= 78

Enter 5th number= 98

Greatest number=  98
```

```
for V in L:
   if V>G:
      G=V
print("Greatest number= ",G)
```

* **Comments:** In python comments are the lines in the code that are ignored by the interpreter during the execution of the program. It enhance the readability of the code and help the programmers to understand the code very carefully.

  There are three types of comments in python:

  1. Single line comments (#)
  2. Multiline comments (#,#)
  3. Docstring comments (“”” …….. ”””)

* **Continue lines with backslash (\):** In python backslash is a line continuation character. If a backslash is placed at the end of a line, it is considered that the line is continued on the next line.

  Ex:

  ```
  mul=2*\
      2*\
      2
  print("Multiplication: ",mul)
  ```

  Output: Multiplication 8

* **Decision control-flow:** In python decision control flow decide the direction of the flow of program execution.

  1. **if statement:** Syntax:

     ```
     if (condition):
         statement1
     ```

  2. **if else statement:** Syntax:

     ```
     if(condition):
         statement1
     else:
         statement2
     ```

  3. **nested if statement:** When we place an if statement inside another if statement is called nested if statement.

Syntax:

```
If(condition1):
    If(condition2):
        Statement
```

4. **if elif else statement:** **Syntax:**

```
if(condition1):
    statement1
elif(condition2):
    statement2
else:
    statement
```

5. **Sort hand if statement: The statement can be put on the same line as the if statement.**

   Syntax: **if(**condition**): statement**

6. **Sort hand if else statement: This can be used to write if_else statement in a single line.**

   Syntax: statement_when_true **if(**condition**) else** false_statement

∗ **Loop statement: A loop statement allows us to execute a statement or group of statements multiple times**.

1. **While loop:** **Syntax:**

```
initialization
while(condition):
    block of code
    increment/decrement
```

2. **For loop: A for loop is used for iterating over a sequence or other iterable object. Such as list, tuple, string, dictionary etc.**

   Syntax:

```
for val in sequence:
    Loop body
```

➢ **The range() function: The range() function returns a sequence of numbers, starting from 0 by default and increment by 1 by default and stop before a specified number.**

range (start, stop, step_size): # By default step_size is 1 if not provided.
# By default starting point is 1.

Ex:

```
for i in range(6):

   print(i,end=" ")

print() #For new line

for i in range(2,9,2):

   print(i,end=" ")

print()

for i in range(11,0,-2):

   print(i,end=" ")

print()

List=[1,2,3,4,5,6]

for V in List:

   print(V,end=" ")
```

```
Output:

0 1 2 3 4 5

2 4 6 8

11 9 7 5 3 1

1 2 3 4 5 6
```

- **Nested loop: If a loop exists inside the body of another loop known nested loop. The inner loop will be executed one time for each iteration of the outer loop.**

Ex:

```
A=["Red","Sweet"]

B=["apple",'Mango']

for V in A:

    for M in B:

         print(V,M)
```

```
Output:

Red apple

Red Mango

Sweet apple

Sweet Mango
```

∗ **Loop control statements are used to control loops. It change the flow of execution. These can be used if we want to skip an iteration or stop the execution.**

The three types of loop control statements:

break **continue** **pass**

- **pass statement is a null statement. But difference between pass and comment is that, comment is ignored by the interpreter whereas pass is not ignored by interpreter.**

Ex:

```
L=[23,45,21,34,21,98]
  for i in L:
     if i==21:
       pass
     else:
       print(i,end=" ")
```

Output:

```
23 45 34 98
```

* **List iteration:** **A list can have another list as an item. This is called nested list.**

```
L=[1,2,3,[1,2,3,4,],5,6,7,8]
print(L)
print(L[3][3])

for i in range(len(L)-1,-1,-1):  #Reverse order iteration
  print(L[i],end="")

for i in range(0,len(L)):
  print(L[i],end="")
```

```
Output:

[1, 2, 3, [1, 2, 3, 4], 5, 6, 7, 8]

4

8765[1, 2, 3, 4]321

123[1, 2, 3, 4]5678
```

List functions:

del	**For deletion element.**	**Ex: del list_name[index_no]**
pop()	**Ex:**	**list_name.pop(index_no)**
remove()	**Ex:**	**list_name.remove(element)**
clear()	**Ex:**	**list_name.clear(),** max(), min(), count(), sum(), reverse(), copy()
insert()	**Ex:**	**list_name.insert(index_no,element)**
append()	**Ex:**	**list_name.append(value)**
extend()	**Ex:**	**list_name.extend(another_list_name)**
sort()	**Ex:**	**list_name.sort()**

List comprehension provides a shorter syntax when we want to create a new list based on the values of an existing list.

Advantage of list comprehension:

- **More time efficient and space efficient then loops.**
- **Require fewer lines of code.**

Syntax:

NewListName=[Expression(element) fpr element in OldListName if condition]

Ex:

```
L=[1,2,3,4,5,6,7,8,9,10,11]

newL=[i for i in L if i%2==1]

print(newL)


List=[v for v in range(13) if v%2==0]

print(List)


str="World"

A=[v for v in str]

print(A)


S=["apple","orange","mango","pomegranate"]

Arr=[V for V in S if 'o' in V]

print(Arr)
```

```
Output:

[1, 3, 5, 7, 9, 11]

[0, 2, 4, 6, 8, 10, 12]

['W', 'o', 'r', 'l', 'd']

['orange', 'mango', 'pomegranate']
```

- **zip() function: The zip() function returns a zip object, which is an iterator of tuples where 1st item in each passed iterator is paired together and then the second item in each passed iterator are paired together etc.**

Ex:

```
L1=[1,2,3,4]
L2=[5,6,7,8]
for x,y in zip(L1,L2):
    print(x,y)
x=zip(L1,L2)
print(x)
print(list(x))
```

Output:

```
1 5

2 6

3 7

4 8

<zip object at 0x0000027D2D78CB40>

[(1, 5), (2, 6), (3, 7), (4, 8)]
```

- **The split() method splits a string into a list. We can specify the separator, the default separator is any whitespace.**
  - **Separator is optional, it specifies the separator to use when splitting the string.**
  - **Max split is optional, it specifies how many splits to do. By default value -1 which is all occurrances.**

Syntax:

```
VariableName=StringName.split(separator,max_split)
```

Ex:

```
S="Python is a pure object oriented language"

x=S.split()

print(x)

Str="Apple,Mango,Banana"

y=Str.split(',',3)

print(y)
```

Output:

```
['Python', 'is', 'a', 'pure', 'object', 'oriented', 'programming', 'language']

['Apple', 'Mango', 'Banana']
```

* **Dictionary functions and implementation:**

get()	**Ex: variable=dictionaryName.get('keyName')**
del()	**Ex: del dictionaryName['key']**
update()	**Ex: dictionayName({'key':values})**
pop()	**Ex: variable= dictionaryName.pop('key')**
clear()	**Ex: dictionaryName.clear()**
dict()	**For creates a dictionary, Ex: variableName=dict(key='value'….)**

Ex1:

```
D={}
D['Fruit']="Mango" #insert
D['Flower']="Marigold"
print(D)

D['veg']="Patato","Tamato"
print(D)

Dic={1:'One',2:'Two'}
for a in Dic:
   print(a,Dic[a])
```

```
Output:

{'Fruit': 'Mango', 'Flower': 'Marigold'}

{'Fruit': 'Mango', 'Flower': 'Marigold', 'veg': ('Patato', 'Tamato')}

1 One

2 Two
```

EX2:

```
D={ 'Fruit':"Mango",
  'Flower':"Marigold"
  }
print(D,type(D))

for V in D.keys():
   print(V,end="")
print()

for V in D.values():
   print(V,end="")
print()

for V,X in D.items():
   print(V,X,end="")
print()

D.update({'Fruit':"Apple"})
print(D)
```

```
Output:

{'Fruit': 'Mango', 'Flower': 'Marigold'} <class 'dict'>

FruitFlower

MangoMarigold

Fruit MangoFlower Marigold

{'Fruit': 'Apple', 'Flower': 'Marigold'}
```

Nested dictionary:

```
D={ 'Fruit':{'Big':"Mango",'Small':"Mango"},

  'Flower':"Marigold"
  }
print(D)
```

```
Output:

{'Fruit': {'Big': 'Mango', 'Small': 'Mango'}, 'Flower': 'Marigold'}
```

* **Set functions and implementation.**

copy()	**Returns copy of the set.**
removes()	**Removes the specified element**
update()	**Update the set with another set or any other iterable.**
set()	**Convert a list into set.**
clear()	**Clear the set.**
add()	**Add an element to the set**

Ex:

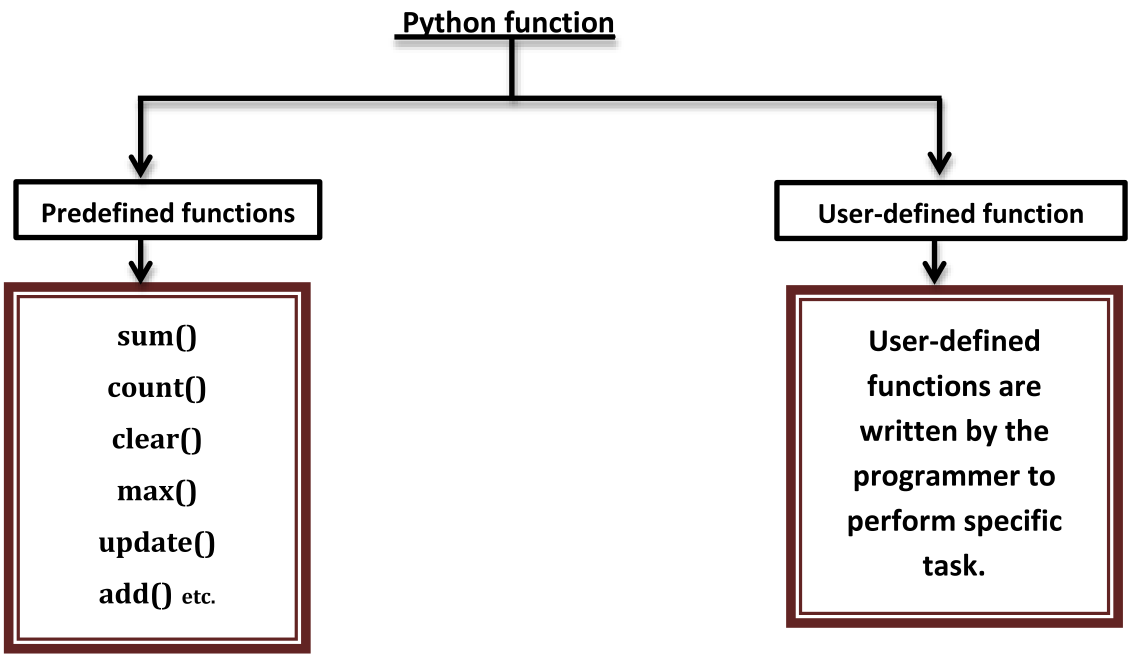
```
S={1,1,2,3,4,5}

print(S,type(S))


L=[96,97,98]

S.update(L)

print(S)


List=["mango","Orange","Apple"]

Set=set(List)

print(Set,type(Set))


for V in S: #Set iteration

   print(V,end=" ")
```

```
Output:

{1, 2, 3, 4, 5} <class 'set'>

{96, 1, 2, 3, 4, 5, 97, 98}

{'Apple', 'Orange', 'mango'} <class 'set'>

96 1 2 3 4 5 97 98
```

* **Function:** **A function is a block of codes that perform specific task. When we divides a large program into the basic building blocks known as functions.**

Advantages of functions:

- Functions improves reusability and readability of the code.
- Function reduce code redundancy and make code modular.
- Function reduce complexity of big program and make optimize the code.



➢ **User-defined function:** In python user defined functions declaration begins with the def keyword and followed by the function name.

The function may take arguments as input within the opening and closing parenthesis, just after the function name followed by colon.

After defining the function name and arguments a block of program statements start at the next line and these statements must be indented.

Syntax:

```
def functionName(argument1,argument2….):
    Statement1
    Statement2
```

Ex1:

These are the positional arguments that need to be included in the proper position or order.

```
def sum(a,b,c):
    A=a+b+c
```

```
            return A
  Val=sum(4,5,6)
 print("Sum=",Val)
```

```
Output:   Sum= 15
```

Ex2:

Default argument

```
def sum(a,b=7):
   A=a+b
  return A
V1=sum(8,11)
V2=sum(8)
print("Sum1=",V1)
print("Sum2=",V2)
```

```
Output:

Sum1= 19

Sum2= 15
```

- **Arbitrary arguments, * args:** If we do not know how many arguments that will be passed into our function, then we have to add a *(star) before the parameter name in the function definition. This way function will receive a tuple of arguments and can access the items accordingly.

  Ex:

  ```
  def function1(*A):
      print(A,type(A))
      function1(1,2,3,4,5,6)
  ```

  ```
  Output:

  (1, 2, 3, 4, 5, 6) <class 'tuple'>
  ```

- **Arbitrary keyword arguments **args:** If we do not know how many keywords arguments that will be passed into our function, then we have to add two asterisk (**) before the parameter name in the function definition. This way function will receive a dictionary of arguments and can access the items accordingly.

  Ex:

  ```
  def function2(**A):
      print(A,type(A))
  function2(key1="Python",key2="Programming")
  ```

  ```
  Output:

  {'key1': 'Python', 'key2': 'Programming'} <class 'dict'>
  ```

- **<u>Keyword argument:</u> Keyword arguments are values that, when passed into a function, are identifiable by specific parameter names.**
    - **The way the order of the arguments does not matter.**

Ex:

```
def function3(key1,key2,key3):

    print("key1=",key1,"\nkey2=",key2,"\nkey3=",key3)

function3(key2='Python',key1="Programming",key3="Function")
```

```
Output:

key1= Programming

key2= Python

key3= Function
```

- **<u>Return multiple values from a function:</u> In python we can return multiple values from functions.**
  Python provides a few handy methods to return multiple values from the function.
    1. **Comma separated values (Tuples)**
    2. **Using python list**
    3. **Using python dictionary**

- **Comma separated values (Tuples): For return multiple values in python basically uses a tuple to achieve this.**

Ex:

```
def function1():

    return(1,2,3,"Apple","Orange")

V=function1()

print(type(V),V)
```

```
Output:

<class 'tuple'> (1, 2, 3, 'Apple', 'Orange')
```

- **Using python list:**

Ex:

```
def function2():
    return[1,2,3,"Apple","Mango"]
V=function2()
print(type(V),V)
```

```
Output:

<class 'list'> [1, 2, 3, 'Apple', 'Mango']
```

In this method we can also use the index of the list to get the value individually.

❖ Using python dictionary:

Ex:

```
def function3():
    D=dict()
    D['key1']="Apple"
    D['key2']="Mango"
    D['key3']="Orange"
    return D
V=function3()
print(type(V),V)
```

Output:

```
<class 'dict'> {'key1': 'Apple', 'key2': 'Mango', 'key3': 'Orange'}
```

* **Module:** **Module is a code library or a file that contains a set of functions, or classes than we can include in our application.**
  Advantages of modules
  - **Reusability: Working with modules makes the code reusable.**
  - **Simplicity: The Module targets a small proportion of the problem rather than aiming at the complete problem.**
  - **Complexity: Module reduce complexity of big program and make optimize the code.**

To create a module, we have to save the code that we wish in a file with the .py file extension. Then the name of the file becomes the name of the module.

To incorporate the created module into our program we have to use import keyword. And if we want to use only specific method from a module we have to use the from keyword.

Ex:

Creating a python user defined module:

Saving a python file with .py extension as $\longrightarrow$ Mod.py

⇨ Importing created Mod.py module in our program.

```
import Mod as M

obj=M.Super()


Sum=obj.addition(1,4,5,6,7,66,)

Mul=M.multiplication(4,3,8)


print("Sum=",Sum)

print("Multiplication=",Mul)
```

```
Output:

Sum= 89

Multiplication= 96
```

Mod.py (Module creation)

```
class Super:
    def addition(self,*arg):
        A=0
        for i in arg:
            A=A+i
        return A


def multiplication(*A):
    M=1
    for V in A:
        M*=V
    return M
```

⇨ Importing specific function from created Mod.py module in our program.

```
from Mod import multiplication
M=multiplication(2,2,3,8)
print("Multiplication=",M)
```

```
Output:

Multiplication= 96
```

⇨ Importing all attributes from created Mod.py module in our program.

```
from Mod import *
obj=Super() #Making class object
add=obj.addition(42,34,5,7,8)
print("Addition=",add)
```

```
Output:

Addition= 96
```

Class & object

Class: A Class is a group of objects which have common properties. It is a logical entity it cannot be physical.

Object: An object is a real word entity. it is the member of class. Each object has an identity a behavior and a state.

```
class MyClass:   #Class creation
         def __init__(self,X,Y):   #Constructor
                self.A=X
                self.B=Y
         def summation(self):     #Function
                return self.A+self.B
        def product(self):   #Function
               return self.A*self.B

Obj1=MyClass(47,48)  #Object creation
Obj2=MyClass(12,8)  #Object creation

S=Obj1.summation()   #Function calling
M=Obj2.product()    #Function calling

print("Addition=",S)
print("Multiplication=",M)
```

```
Output:

Addition= 95

Multiplication= 96
```

* **Inheritance:** Inheritance is a mechanism in which a new class acquires all properties and behaviors of an existing parent class.
  Advantages of Inheritance
  - Inheritence improves reusability and readability of the code.
  - Minimizing duplicate code.
  - Method Overriding can be achieved.
  - Inheritence reduce complexity of big program and make optimize the code.

➢ Single Inheritance: When a class inherits another class, it is known as a single inheritance.

```
class Base:

        def input(self):

               print("Enter Name,SRN and Phone number of a student= ")

               self.Name=input()
```

```
            self.SRN=input()

            self.Pho=input()

class Derived(Base):

        def output(self):

              print("Name= {}\nSRN= {}\nPhone No= {}".format(self.Name,self.SRN,self.Pho))

Obj=Derived()

Obj.input()

Obj.output()
```

```
Output:

Enter Name,SRN and
Phone number of a
student=

Rohit

R22DE123

9736253768

Name= Rohit

SRN= R22DE123

Phone No= 9736253768
```

- **Multilevel inheritance: When there is a chain of inheritance, it is known as multilevel inheritance.**

```
class A:
      def input(self):
            self.N=int(input("Enter a number="))
class B(A):
       def factorial(self):
             f=1
       num=self.N
       while(num>0):
            f=f*num
            num-=1
       self.fact=f
class C(B):
       def output(self):
            print("Factorial of {} is= ".format(self.N),self.fact)
obj=C()
obj.input()
obj.factorial()
obj.output()
```

```
Output:

Enter a number=6

Factorial of 6 is=  720
```

- **Multiple inheritance: When a class is derived from more then one base class it is known as multiple inheritance.**

  Ex:

```
class A:
   def inputSide(self):
      self.S=int(input("Enter the side of the square="))
class B:
   def inputRadius(self):
      self.R=int(input("Enter the radius of the circle="))

class C(A,B):
   def area(self):
      square=self.S*self.S
      circle=3.14*self.R*self.R
      print("Square area=",square)
      print("Circle area=",circle)
obj=C()
obj.inputSide()
obj.inputRadius()
obj.area()
```

```
Output:

Enter the side of the square=7

Enter the radius of the circle=8

Square area= 49

Circle area= 200.96
```

- **Hierarchical inheritance: When a single class is inherited by two or more classes, it is known as hierarchical inheritance.**

```
class A:
   def input(self):
      print("Enter length and breadth of a rectangle=")
      self.l=int(input())
      self.b=int(input())
class B(A):
   def area(self):
      Ar=self.l*self.b
      print("Rectangle area=",Ar)
class C(A):
   def perimeter(self):
      Pe=2*(self.l+self.b)
      print("Rectangle perimeter=",Pe)
class Main:
   Obj1=B()
   Obj2=C()
   Obj1.input()
```

```
Output:

Enter length and breadth of a rectangle=

67

87

Rectangle area= 5829

Enter length and breadth of a rectangle=

78

87

Rectangle perimeter= 330
```

```
    Obj1.area()
    Obj2.input()
    Obj2.perimeter()
```

* **Method overriding:** Whenever we writes a method in Base class and derived class in such a way that method name and parameter must be same known as method overriding.

  Ex:

```
class A:
    def arithmeticOperation(self,a,b,c):
        sum=a+b+c
        print("Summation= ",sum)
class B(A):
    def arithmeticOperation(self,a,b,c):
        super().arithmeticOperation(a,b,c)
        mul=a*b*c
        print("Multiplication= ",mul)

class Main:
    obj=B()
    print("Enter three numbers=")
    x=int(input())
    y=int(input())
    z=int(input())
    obj.arithmeticOperation(x,y,z)
```

```
Output:

Enter three numbers=

6

7

8

Summation=  21

Multiplication=  336
```

* **Access modifiers** determine the accessibility of the members of a class.
  - <u>public:</u> All member variables of the class are by default public.
  - <u>private:</u> To define a private variable add two underscores as a prefix at the start of a variables name or methods name.
  - <u>protected:</u> To define a protected variable add one(single) underscore as a prefix at the start of a variable name.

* **Get and Set attributes:** To implement proper encapsulation in python we need to use get and set attributes.

  In oops languages get and set methods are used to retrieve and update data.

The main purpose of get and set methods is:

- **To avoid direct access of private variables.**
- **To add validation logic for setting a value.**

➢ **Get(): The get() method is used for retrieves an object current attribute value and access a class private attribute.**

➢ **Set(): The set() method is used for changes an object's attribute value and set the value of private attributes in a class.**

Ex:

```
class Student:
   def __init__(self,N,S):
      self.__Name=N
      self.__SRN=S
   def getDetails(self):
      return self.__Name,self.__SRN
   def setDetails(self,name,srn):
      self.__Name=name
      self.__SRN=srn

Obj=Student('Ranvir','R22DE078')
V=Obj.getDetails()

print("Entered student records are following: ")
print("Name={}\tS R N={}".format(V[0],V[1]))

print("Enter the name and SRN of a student=")
Name=input()
SRN=input()

Obj.setDetails(Name,SRN)
V=Obj.getDetails()

print("Entered student records are following: ")
print("Name={}\tS R N={}".format(V[0],V[1]))
```

```
Output:

Entered student records are following:

Name=Ranvir        S R N=R22DE078

Enter the name and SRN of a student=

Ranjeet

R22DE088

Entered student records are following:

Name=Ranjeet       S R N=R22DE088
```

* **Inner function: A function which is defined inside another function is known as inner function or nested function.**

Ex:

```
def function1():
   S="Python programming"
   def function2():
      print(S)
   function2()
#Driver code
function1()
```

```
Output: Python programming
```

* **Anonymous function:** **In python an anonymous function is a function that is defined without a name while normal functions are defined using the def keyword, but anonymous functions are defined using lambda keyword, hence anonymous functions are also called lambda functions.**

Ex:

```
Str="Python programming"
upperRev=lambda string:string.upper()[::-1]
print(upperRev(Str))
```

```
Output:

GNIMMARGORP NOHTYP
```

* **Generator function:** **A generator function is defined like a normal function but whenever it needs to generate a value, it does so with the yield keyword rather then return.**

If the body of a def contains yield, the function automatically becomes a generator function.

Ex:

```
def function1():
   yield 1
   yield 2
   yield 3
for Val in function1():
   print(Val)
```

```
Output:

1

2

3
```

- **Generator function returns a generator object.**

Ex:

```
def function2():

   yield 1
   yield 2
   yield 3
V=function2()
print(next(V))
print(next(V))
print(next(V))
```

```
Output:
1
2
3
```

* **Exception handling:** **Exception handling is a mechanism to handle run time errors such as valueError, NameError, IndexError, ZeroDivisionError, TypeError, KeyError etc.**

  The main advantage of exception handling is to maintain the normal flow of the application. An exception normally disrupts the normal flow of the application, that's why we need to handle exceptions.

  - **Syntax error:** **Syntax errors are mistakes in the source code, such as spelling and punctuation errors, incorrect labels and so on, which cause an error message to be generated by the compiler or interpreter.**

    In other words syntax error caused by not following the proper structure (syntax) of the programming language is called syntax error.

    Ex:
    ```
    B=2
    If B>3    #syntax error
       Print("B is greater")
    ```

  - **Logical error:** **Errors that occurs at runtime after passing the syntax test. In logical error the program can run but produces an incorrect result.**

    Logical error occurs when there is a fault in the logic of the problem in programming language.

In python try, except, else and finally keywords are used to handle exceptions.

- **try: The "try" keyword is used to specify a block of code where exception may be occur. It means we can't use try block alone. The try block must be followed by either except or finally.**

- **except: The "except" block is used to handle the exception. It must be preceded by try block which means we can't use except block alone. It can be followed by finally block later.**
- **else: The else part executes if the try clause does not raise an exception.**
- **finally: The "finally" block is used to execute the necessary code of the program. It is always executed whether an exception is handled or not.**

Ex:

```
try:
   print("Enter two numbers=")
   a=int(input())
   b=int(input())
   d=a/b
   print("Division= ",d)
   List=[2,3,4,5,6,7,8,9]
   i=int(input("Enter index number="))
   print("Element at entered index= ",List[i])
except ZeroDivisionError:
   print(" A number can not be divided by zero")
except ValueError:
   print("Enter only integer number=")
except IndexError:
   print("Index no. is out of range")
else:
   print("Their is no any exception in try block")
finally:
   print("Program has been terminated")
```

Output:

```
Enter two numbers=
5
7
Division=  0.7142857142857143
Enter index number=6
Element at entered index=  8
Their is no any exception in try block
Program has been terminated
```

File handling

Files are the named locations on disk to store related information. They are used to store data permanently in a non-volatile memory.

❈ Create a new empty text file: The new file can be created by using one of the following access modes with the function open().

The open() function accepts two arguments file_name and access_mode in which the file is accessed. The function returns the file object which can be used to perform various operations like reading, writing etc.

Mode	Description
x	It creates a new file with the specified name. It causes an error if a file exits with the same name.
a	It creates a new file with specified name if no such file exits. It appends the content to the file if the file already exits with the specified name.
w	It creates a new file with specified name if no such file exits. It overwrites the exits file.
r	Open an existing file for a read operation.
r+	To read and write data into the file. The previous data in the file will be overridden.
w+	To write and read data. It will override existing data.
a+	To append and read data from the file. It won't override existing data.

Before performing any operation on the file first we have to open that file with open() function. But at the time of opening file, we have to specify the mode which represents the purpose of the opening file.
The following operations we can perform on the file.

- Creating a new file
- Open a file
- Write data into the file
- Read data from the file
- Close the file

➢ <u>Creating a new file:</u> Syntax⟶ file_object=open("Location\\fileName","mode")

➢ <u>Open a file:</u> The open() function is used to open internally stored file. It returns the contents of the file as python object. Ex: obj=open("E:myfile.txt",'w')

- **<u>Write data into the file:</u> The write() method is used to write data into a file using the file object. Write() method writes strings into the into the file. Always remember to close the file object after creating it.**

Ex:

```
obj=open("E:\\Myfile.txt",'w')

string=input("Enter the data=")

obj.write(string)

obj.close()
```

- **<u>Read data from the file:</u> The read(), readline() and readlines() methods are used to read data from the file.**
    - **<u>read():</u> The read() method returns the specified number of bytes from the file. Default is -1 which means the whole file.**
      <u>Ex:</u>
      ```
      obj=open("E:\\Myfile.txt",'r')
      print(obj.read(6))
      obj.close()
      ```
    - **<u>readline():</u> The readline() method returns one line from the file. We can also specified how many bytes from the line to return, by using the size parameter.**
      <u>Ex:</u>
      ```
      obj=open("E:\\Myfile.txt",'r')
      print(obj.readline())
      print(obj.readline())
      obj.close()
      ```
    - **<u>readlines():</u> The readlines() method returns a list containing each line in the file as a list. We can also specified how many bytes from the line to return, by using the size parameter.**
      <u>Ex:</u>
      ```
      obj=open("E:\\Myfile.txt",'r')
      print(obj.readlines())
      obj.close()
      ```

- <u>Close the file:</u> The close() method closes an open file.
  Syntax: fileObject.close()

- <u>Write a binary file with write():</u> To open a file in binary format and add wb modes as parameter to open the file in binary format for writing.
  Ex:
  ```
  obj=open("E:\\Myfile.bin",'wb')
  List=[2,3,4,5,6]
  Ar=bytearray(List)
  obj.write(Ar)
  obj.close()
  ```

- Read a binary file: To read a binary file, firstly we have to open that file in rb mode, then we can read that file opened file.
  Ex:
  ```
  obj=open("E:\\Myfile.bin",'rb')
  dataByte=obj.read()
  Values=list(dataByte)
  print(Values)
  obj.close()
  ```

* **Decorators:** A decorator is a design pattern in python that allows a user to add new functionality to an existing object without modifying its structure.
  Decorators are usually called before the definition we want to decorate.

Decorators are a very powerful tool in python, it allows programmers to modify the behavior of a function or class.

- A function is an instance of the object type.
- We can store the function in a variable.
- We can pass the function as a parameter to another function.
- We can return the function from a function.
- We can store the function in data structures such as lists, tuple etc.

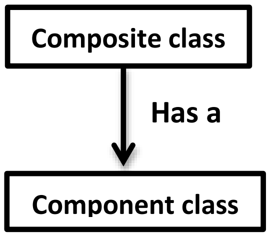
Ex:

```
def upperFunction(text):
```

```
    return text.upper()
def lowerFunction(text):
    return text.lower()
def function(method):
    str=input("Enter a string value=")
    val=method(str)
    print(val)
function(upperFunction)
function(lowerFunction)
```

```
Output:
Enter a string value=python
PYTHON
Enter a string value=DECORATORS
decorators
```

* **Composition:** **Composition is a concept that models a has a relationship. It enables creating complex types by combining objects of other types. This means that a class composite can contain an object of another class component. This relationship means that a composite has a component.**



Ex:

```
class Salary:
    val=989
    def empSalary(self):
        return 2322282
class Employee:
    def getSal(self):
        self.Emp=Salary()
        return self.Emp.empSalary()
    def value(self):
        print(self.Emp.val)
obj=Employee()
```

```
Output:
2322282
989
```

```
        print(obj.getSal())
        obj.value()
```

⇨ **# The string with every ‘p’ replace with an ‘u’.**

```
str="apple papaya"
for i in str:
   if i=='p':
      str=str.replace(i,'u')
print(str)

      # OR
print(str.replace('o','u'))
```