1. INTRODUCTION TO PYTHON

Python History and Versions

- Python laid its foundation in the late 1980s.
- The implementation of Python was started in the December 1989 by **Guido Van Rossum** at CWI in Netherland.
- In February 1991, van Rossum published the code (labeled version 0.9.0) to alt.sources.
- In 1994, Python 1.0 was released with new features like: lambda, map, filter, and reduce.
- Python 2.0 added new features like: list comprehensions, garbage collection system.
- On December 3, 2008, Python 3.0 (also called "Py3K") was released. It was designed to rectify fundamental flaw of the language.
- *ABC programming language* is said to be the predecessor of Python language which was capable of Exception Handling and interfacing with Amoeba Operating System.
- Python is influenced by following programming languages:
    - ABC language.
    - Modula-3

Python Version List

Python Version	Released Date
Python 1.0	January 1994
Python 1.5	December 31, 1997
Python 1.6	September 5, 2000
Python 2.0	October 16, 2000
Python 2.1	April 17, 2001
Python 2.2	December 21, 2001
Python 2.3	July 29, 2003
Python 2.4	November 30, 2004
Python 2.5	September 19, 2006
Python 2.6	October 1, 2008
Python 2.7	July 3, 2010
Python 3.0	December 3, 2008
Python 3.1	June 27, 2009
Python 3.2	February 20, 2011
Python 3.3	September 29, 2012
Python 3.4	March 16, 2014
Python 3.5	September 13, 2015
Python 3.6	December 23, 2016

Python 3.7	June 27, 2018

• Python Features

Python provides lots of features that are listed below.

1) Easy to Learn and Use

Python is easy to learn and use. It is developer-friendly and high level programming language.

2) Expressive Language

Python language is more expressive means that it is more understandable and readable.

3) Interpreted Language

Python is an interpreted language i.e. interpreter executes the code line by line at a time. This makes debugging easy and thus suitable for beginners.

4) Cross-platform Language

Python can run equally on different platforms such as Windows, Linux, Unix and Macintosh etc. So, we can say that Python is a portable language.

5) Free and Open Source

Python language is freely available at https://www.python.org/. The source-code is also available. Therefore it is open source.

6) Object-Oriented Language

Python supports object oriented language and concepts of classes and objects come into existence.

7) Extensible

It implies that other languages such as C/C++ can be used to compile the code and thus it can be used further in our python code.

8) Large Standard Library

Python has a large and broad library and provides rich set of module and functions for rapid application development.

9) GUI Programming Support

Graphical user interfaces can be developed using Python.

10) Integrated

It can be easily integrated with languages like C, C++, JAVA etc.

• Python Applications

Python is known for its general purpose nature that makes it applicable in almost each domain of software development. Python as a whole can be used in any sphere of development.

Here, we are specifying applications areas where python can be applied.

1) Web Applications

We can use Python to develop web applications. It provides libraries to handle internet protocols such as HTML and XML, JSON, Email processing, request, beautifulSoup, Feedparser etc. It also provides Frameworks such as Django, Pyramid, Flask etc to design and delelop web based applications. Some important developments are: PythonWikiEngines, Pocoo, PythonBlogSoftware etc.

2) Desktop GUI Applications

Python provides Tk GUI library to develop user interface in python based application. Some other useful toolkits wxWidgets, Kivy, pyqt that are useable on several platforms. The Kivy is popular for writing multitouch applications.

3) Software Development

Python is helpful for software development process. It works as a support language and can be used for build control and management, testing etc.

4) Scientific and Numeric

Python is popular and widely used in scientific and numeric computing. Some useful library and package are SciPy, Pandas, IPython etc. SciPy is group of packages of engineering, science and mathematics.

5) Business Applications

Python is used to build Bussiness applications like ERP and e-commerce systems. Tryton is a high level application platform.

6) Console Based Application

We can use Python to develop console based applications. For example: **IPython**.

7) Audio or Video based Applications

Python is awesome to perform multiple tasks and can be used to develop multimedia applications. Some of real applications are: TimPlayer, cplay etc.

8) 3D CAD Applications

To create CAD application Fandango is a real application which provides full features of CAD.

9) Enterprise Applications

Python can be used to create applications which can be used within an Enterprise or an Organization. Some real time applications are: OpenErp, Tryton, Picalo etc.

10) Applications for Images

Using Python several application can be developed for image. Applications developed are: VPython, Gogh, imgSeek etc.

There are several such applications which can be developed using Python

• Execution of a python program:

Assume that we write a python program with the name first.py. here, first is the program name and the .py is the extension name. After typing the program, the next step is to compile the program using python compiler. The compiler converts the python program into another code called byte code.

Byte code represents a fixed set of instructions that represents all operations like arithmetic operations, comparison operations, memory related operations etc., which run on any operating system and hardware. It means the byte code instruction are system independent or platform independent. The size of each byte code instruction is 1 byte and hence they are called with the name **byte code**. These byte code instructions are contained in the file **first.pyc**. Here, the first.pyc file represents a python compiled file.

The next step is to run the program. If we directly give the byte code to the computer, it cannot execute them. The binary code is understandable to the machine, it is also called machine code.

It is therefore necessary to convert the byte code into machine code so that our computer can understand and execute it. For this purpose, we should use PVM(Python Virtual Machine).

PVM uses an interpreter which understands the byte code and converts it into machine code. PVM first understand the processor and operating system in our computer. Then it converts the byte code into machine code understandable to that processor and into that format understandable to that operating system. These machine code instructions are then executed by the processor and results are displayed.

An interpreter translates the program source code line by line. Hence, it is slow. The interpreter that is found inside the PVM runs the python program slowly.

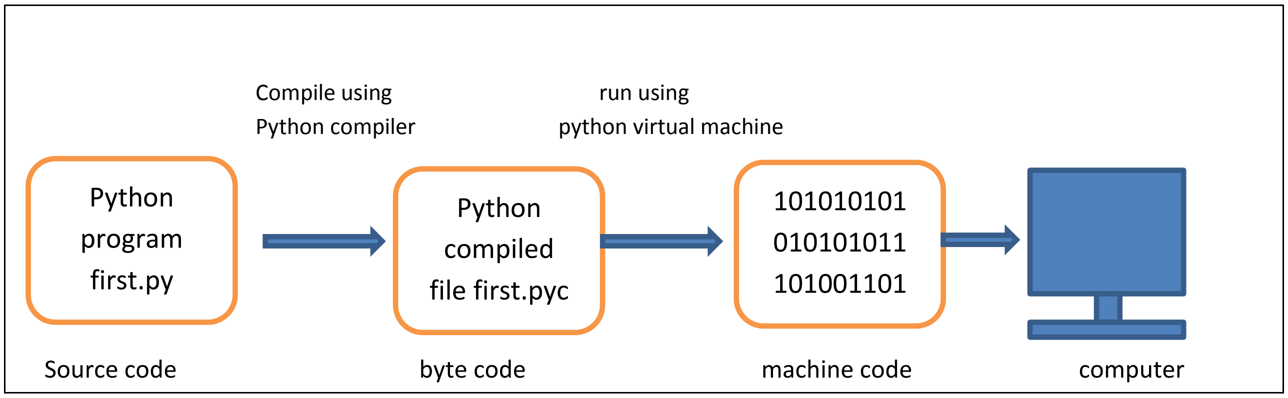


Fig: Steps of Execution of Python Program

Source code -> byte code -> machine code -> output

To separately create .pyc file from the source code, we can use the following command

```
c:\>python –m py_compile first.py
```

we want to see the byte code instructions that were created internally by the python compiler before they are executed by the PVM. For this purpose, we should spedify the dis module while using python command as:

```
c:\python –m dis first.py
```

- **Flavors of Python:**

CPython:

This is the standard python compiler implemented in C language. This is the Python software being downloaded and used by programmers directly from https:/www.python.org/download/. CPython is used to execute C and C++ functions and programs.

Jython:

This is earlier known as Jpython. This is the implementation of Python programming language which is designed to run on Java Platform. Jython compile first compiles the Python program into java byte code. This byte code is executed by Java Virtual Machine(JVM) to produce the output. This can be downloaded from http:/www.jython.org/.

IronPython:

This is another implementation of Python language for .NET framework. This is written in C# (C Sharp) language. This runs on CLR (Common Language Runtime).

PyPy:

This is Python implementation using Python language. Actually, PyPy is written in a language called RPython which was created in Python language. PyPy programs run very fast since there is a JIT(Just In Time) compiler added to the PVM.

RubyPython:

This is bridge between the Ruby and Python interpreters. It encloses a Python interpreter inside Ruby applications.

Pythonxy:

This is pronounced as Python xy and written as Python(X,Y). This is the Python implementation that get after adding scientific and engineering related packages.

AnacondaPython:

When Python is redeveloped for handling large-scale data processing, predictive analytics and scientific computing, it is called Anaconda Python.

- **Memory Management in Python:**

In C and C++, the programmer should allocate and deallocate (free) memory dynamically, during runtime. For example, to allocate memory,the programmer may use malloc() function and to deallocate the memory, he may use the free() function. But in Python, memory allocation and deallocation are done durin runtime automatically. The programmer need not allocate memory which creating objects or deallocate memory when deleting the objects. Python PVM will take care for this.

Everything is considered as an object in Python. For Example, strings are objects, Lists are objects, Functions are objects, Even modules are objects. For every object memory should be allocated. All objects are stored on a separate memory called ***heap.*** Heap is the memory which is allocated during runtime. The size of the heap memory depends on the RAM of our computer. For example, an integer number should be stored in memory in one way and a string should be stored in a different way.

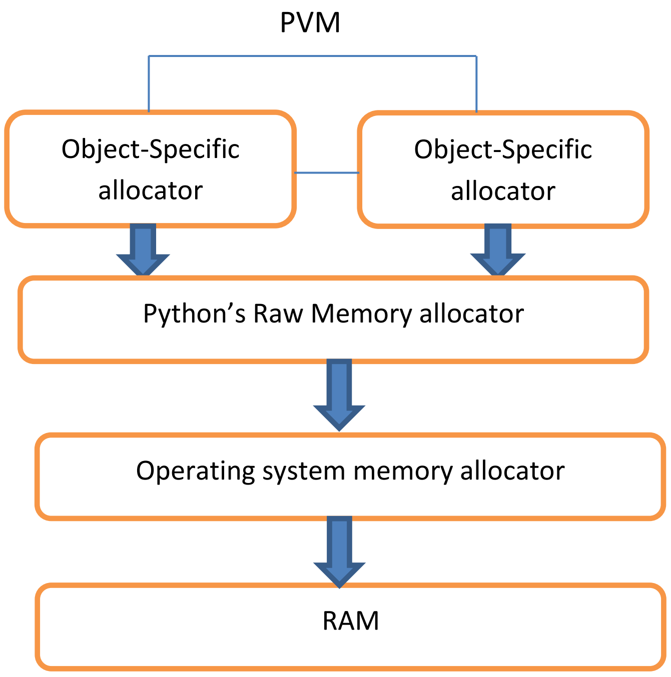

PVM
Object-Specific allocator
Object-Specific allocator
Python's Raw Memory allocator
Operating system memory allocator
RAM

Fig: Allocation of memory by python's virtual Machine (PVM)

- **<u>Garbage Collection in Python:</u>**

A module represents Python code that performs a specific task. Garbage Collector is a module in Python that is useful to delete objects from memory which are not used in the program. The module that represents the garbase collector is named as gc. Garbage collector in the simplest way to maintain a count for each object regarding how many times that object is referenced. When an object is referenced twice, its reference count will be 2. When an object has some count, it is being used in the program and hence garbage collector will not remove it from memory. When an object is found with reference count 0, garbage collector will understand that the object is not used by the program and hence it can be deleted from memory. Hence, the memory allocated for that object is deallocated or freed.

Garbage collector can detect reference cycles. A reference cycle is a cycle of references pointing to the first object from last object. For example, take three objects A, B and C. The object A refers to the object B whereas the object B holds a reference to the object C. Now if the object C refers to the first object A, it will form a reference cycle.

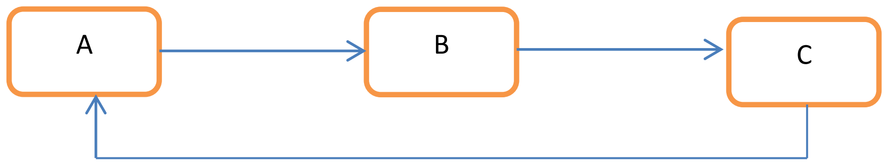


Fig: A reference Cycle of Three Objects

Garbage collector runs automatically. Python schedules garbage collector depending upon a number called *threshold.* This number represents the frequency of how many times the garbage collector removed the objects. We can know the threshold number by using the method get_threshold() of gc module.

- **<u>Getting help in Python:</u>**

When a programmer faces some doubt about how to use particular feature of the Python language, he can view the help. To get help, one can enter help mode by typing help() at python prompt. (>>> prompt).

```
>>>help()
help>topics
help>LOOPING
>>>help('LOOPING')
```

- **Writing Our First Python Program:**
    - **Executing a Python Program**
        - **Using Python’s command line window**
        - **Using Python’s IDLE graphics window**
        - **Directly from System prompt.**
- **Using Python’s Command Line Window:**
- **Using Python’s IDLE Graphics Window:**
- **Directly from System Prompt:**

2. DATATYPES IN PYTHON

- **Comments in Python**

  There are two types of comments in Python: single line comments and multi line comments.

  Single line comments:

  These comments start with a hash symbol (#) and are useful to mention that the entire line till the end should be treated as comment. Comments are non-executable statements. It means neither the Python compiler nor the PVM will execute them. For example.

```
# to find sum of two numbers
a=10 # store 10 into variable a
```

  Multiline comments:

  When we want to mark several lines as comment, we can write the previous block of code inside “”” (triple double quotes) or ‘’’(triple single quote) in the beginning and ending the block as:

```
“””
This is a program to find net salary of an employee
Based on the basic salary, pf, house rent allowance,
Dearness allowance and income tax.
“””
```

- **Docstrings**

  In fact, python supports only single line comments. Multi line comments are not available in Python. The triple double quotes or triple single quotes are actually not multi line comments but they are regular strings with the exception that they can span multiple lines. That means memory will be allocated to these strings internally. If these strings are not assigned to any variable, then they are removed from memory by the garbage collector and hence these can be used as comments.

  So, using “”” are ‘’’ or not recommended for comments by python people since they internally occupy memory and would waste time of the interpreter since the interpreter has to check them.

  If we write strings inside “”” or ‘’’ and if these strings are written as first statements in a module, function, class or a method, then these strings are called documentation strings or Docstrings. These Docstrings are useful to create an API documentation file from a Python program. An API (Application Programming Interface) documentation file is a text file or html file that what contains description of all the features of software, language or a product.

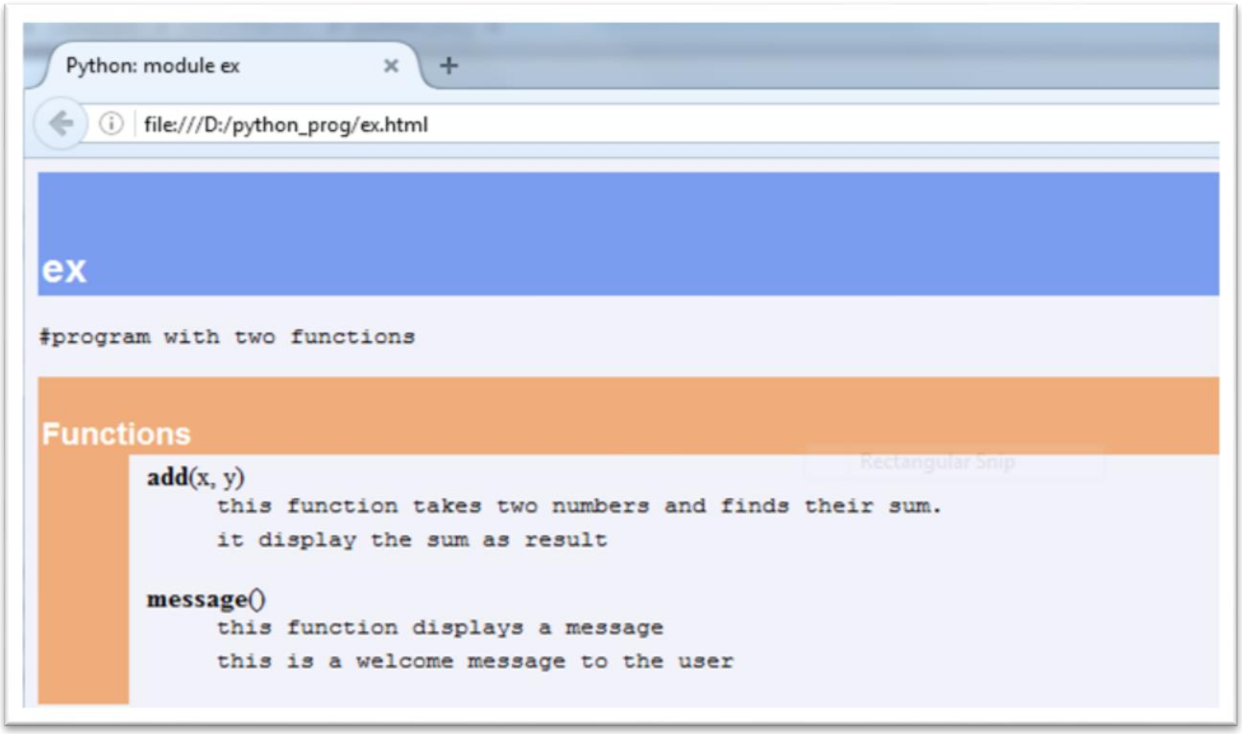
```
# program with two functions
def add(x,y):
   “””
   This function takes two numbers and finds their sum
   It displays the sum as result
   “””
   print(“Sum = “,(x+y))
 def message()
    ‘’’
    This function displays  a message
    This is a welcome message to the user.
    ‘’’
    print(“welcome to Python”)

# now call the functions
add(10,25)
message()
```

After typing this program, save it as ex.py and then execute it as:

```
d:\3102>python ex.py
sum = 35
welcome to Python
```

Now, we will execute this program once again to create API documentation file. For this purpose, we need a module ***pydoc.*** In the following command, we are using –m to represent a module and ***pydoc*** is the module name that is used by the python interpreter. After that –w indicates that an html file is to be created. The last word 'ex' represents the source file name.

```
d:\3102>python –m pydoc –w ex
sum = 35
welcome to Python
wrote to ex.html
```



ex

```
#program with two functions
```

Functions

```
add(x, y)
    this function takes two numbers and finds their sum.
    it display the sum as result

message()
    this function displays a message
    this is a welcome message to the user
```

Python Variables:

Python, a variable is seen as a tag (or name) that is tied to some value. For example, the statement.

a=1

Means the value ‘1’ is created first in memory and then a tag by the name ‘a’ is created to show that value as shown in.

$$\begin{tikzcd} a \arrow[r] & 1 \end{tikzcd}$$

Python considers the values (1 or 2 etc.) as ‘objects’. If we change the value of ‘a’ to a new value as:

a=2

then the tag is simply changed to the new value (or object), as shown below. Since the value ‘1’ becomes unreferenced object, it is removed by garbage collector.

$$\begin{tikzcd} a \arrow[r] & 2 & 1 \end{tikzcd}$$

b=a

Here, we are storing ‘a’ value into ‘b’. A new tag ‘b’ will be created that refers to ‘2’ as shown in below.

$$\begin{tikzcd} a \arrow[dr] & \\ & 2 \\ b \arrow[ur] & \end{tikzcd}$$

- **<u>Data types in Python:</u>**

A datatype represents the type of data stored into a variable or memory. The datatypes which are already available in Python language are called Build-in datatypes. The datatypes which can be created by the programmers are called User-defined datatypes.

Built-in datatypes:

- **None Type**
- **Numeric types**
- **Sequence**
- **Sets**
- **Mappings**

1. **<u>The none type:</u>**

In python, the 'none' datatype represents an object that does not contain any value. In languages like java, it is called 'null' object. But in Python, it is called 'None' object. In Boolean expressions, 'None' datatype represents 'False'.

2. **<u>Numeric Types</u>**

The numeric types represent numbers. There are three sub types.

- ✓ int
- ✓ float
- ✓ complex

int Datatype:

The int datatype represents an integer number. An integer number is a number without any decimal point of fraction part. For example, 200, -50, 0, 9888998700, etc. are treated as integer numbers.

```
a=-57
a=20
```

float Datatype:

The float datatype represents floating point numbers. A floating point number is a number that contains a decimal point. For example, 0.5, -3.4567, 290.08, 0.001 etc are called floating point numbers.

num=55.67998

Complex Datatype

A complex number is a number that is written in the form of a+bj. For example, 3+5j, -1-5.5J, 0.2+10.5J are all complex numbers.

```
#python program to add two complex numbers
c1=2.5+2.5j
c2=3.0-0.5j
c3=c1+c2
print("sum = ",c3)

output
sum =  (5.5+2j)
```

bool Datatype:

The bool datatype in Python represents Boolean values. There are only two Boolean values True or False can be represented by this datatype. Python internally represents True as 1 and False as 0. A blank string like “” is also represented as false. Conditions will be evaluated to either true or false. For example,

```
a=10
b=20
if(a<b):print(“hello”)   # display hello
```

```
a=10>5   # here ‘a’ is treated as bool type variable
print(a)  #display true

a=5>10
 print(a)   #display false
```

3. Sequence in Python

Generally, a sequence represents a group of elements or items. For example, a group of integer numbers will form a sequence. There are six types of sequences in Python.

- str
- bytes
- bytearray
- list
- tuple
- range

str Datatype:

In python, str represents string datatype. A string is represented by a group of characters. Strings are enclosed in single quotes or double quote. Both are valid.

str=”welcome” or str=’welcome’ #here str is name of string type variable

we can also write strings inside “”” (triple double quotes) or ‘’’(triple single quotes) to span a group of lines including spaces.

```
str1=”””this is concept of Pyhon which
discusses about str datatype”””
```

The slice operator repreents square brackets [and] to retrieve pieces of a string. For example, the characters in a string are counted from 0 onwards. Hence str[0] indicates the 0th character or beginning character in the string.

```
s=’welcome to core python’  #this is original string
print(s)
welcome to core python
print(s[0])  #display 0th character from s
w
print(s[4])
o
print(s[3:7])  #display from 3rd to 6th characters
come
print(s[11:])  #display from 11th character onwards till end
core python
print(s[-1])    #display first character from the end
print(s*2)
welcome to core pythonwelcome to core python
```

bytes Datatype:

The bytes datatype represents a group of byte numbers just like an array does. A byte number is any positive integer from 0 to 255. Bytes array can store numbers in the range from 0 to 255 and it cannot even store negative numbers. For example,

```
elements=[10,20,0,40,15]       #this is a list of byte numbers
x=bytes(elements)              #converts the list into bytes array
print(x[0])                   #display 0th element, i.e. 10
```

we can not modify or edit any element in the bytes type array.
For example, x[0]=55 gives an error.

To create bytes type array with a group of elements and then display the elements using a for loop. In this program, we are using a for loop as:

```
for i in x: print(i)
```

bytearray Datatype:

The bytearray datatype is similar to bytes datatype. The difference is that the bytes type array cannot be modified but the bytearray type array can be modified.

```
elements=[10,20,30,40,50]
x=bytearray(elements)
x[0]=88  #replace 0th element by 88
```

list Datatype:

Lists in python are similar to arrays in C or Java. A list represents a group of elements. The main difference between a list and an array is that the list can store different types of elements but an array can store only one type of elements. Also, lists can grow dynamically in memory. But the size of arrays is fixed and they cannot grow at runtime.

```
list=[10,-20,15.5,’vijay’,”mary”]
```

We will create a list with different types of elements. The slicing operation like [0:3] reoresents elements from 0th to 2nd positions, i.e. 10,-20,15.5

```
print(list[0:3]
print(list[0])
print(list[1:3)
print(list[-2])
```

tuple Datatype

A tuple is similar to a list. A tuple contains a group of elements which can be of different types. The elements in the tuple are separated by commas and enclosed in parentheses(). Whereas the list elements can be modified, it is not possible to modify the tuple elements. That means a tuple can be treated as read-only list.

```
tpl=(10,-20,15.5,"vipul",'vipul')
```

range datatype

The range datatype represents a sequence of numbers. The numbers in the range are not modifiable. Generally, range is used for repeating a for loop for a specific number of times. To create a range of numbers, we can simply write:

```
r=range(10)
```

Here, the range object is created with the numbers starting from 0 to 9.

```
for i in r:print(i)
```

The above statement will display numbers from 0 to 9. We can use a starting number, an ending number and a step value in the range object as:

```
r=range(30,40,2)
```

This will create a range object with a starting number 30 and an ending number 39. The step size is 2.

```
for i in r: print(i)
```

will display numbers: 30,32,34,36,38.

```
>>> lst=list(range(10))
>>> print(lst)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> lst=list(range(30,40,2))
>>> print(lst)
[30, 32, 34, 36, 38]
```

4. Sets in Python

A set is an unordered collection of elements much like a set in Mathematics. The order of elements is not maintained in the sets. It means the elements may not appear in the same order as they are entered into the set. Moreover, a set does not accept duplicate elements. There are two sub types in sets.

- set datatype
- frozenset datatype

set datatype:

To create a set, we should enter the elements separated by commas inside curly braces { }

```
s={10,20,30,20,50}
>>> print(s)
{10, 20, 50, 30}
```

Observe that the set ‘s’ is not maintaining the order of the elements. We entered the elements in the order 10,20,30,20,50. But it is showing another order. Also, we repeated the element 20 in the set, but is has stored only one 20. We can use the set() function to create a set as;

```
>>> s={10,20,30,20,50}
>>> print(s)
{10, 20, 50, 30}
>>> ch=set("hello")
>>> print(ch)
{'h', 'e', 'o', 'l'}
>>> nch=set("vviipul")
>>> print(nch)
{'i', 'l', 'p', 'u', 'v'}
>>> print(nch)
{'i', 'l', 'p', 'u', 'v'}
>>> lst=[1,2,5,4,3]
>>> s=set(lst)
>>> print(s)
{1, 2, 3, 4, 5}
>>> s={1,2,5,4,3}
>>> print(s)
{1, 2, 3, 4, 5}
s.update([50,60])
>>> print(s)
{1, 2, 3, 4, 5, 50, 60}
>>> s.remove(3)
>>> print(s)
{1, 2, 4, 5, 50, 60}
```

frozenset datatype:

The frozenset datatype is same as the set datatype. The main difference is that the elements in the set datatype can be modified; whereas, the elements of frozenset cannot be modified.

```
>>> s={50,60,70,80,90}
>>> print(s)
{70, 80, 50, 90, 60}
>>> fs=frozenset(s)
>>> print(fs)
frozenset({80, 50, 70, 90, 60})
```

```
>>> fs=frozenset("abcdefg")
>>> print(fs)
frozenset({'g', 'a', 'f', 'b', 'e', 'd', 'c'})
```

However, update() and remove() methods will not work on frozensets since they cannot be modified or updated.

5. Mapping Types in Python

A map represents a group of elements in the form of key value pairs so that when the key is given, we can retrieve the value associated with it. The dict datatype is an example for a map. The ‘dict’ represents a ‘dictionary’ that contains pairs of elements such that the first element represents the key and the next one becomes its value. The key and its value should be separated by a colon (:) and every pair should be separated by a comma.

```
>>> d={10:'vipul',11:'pranav',12:'haresh'}
>>> print(d)
{10: 'vipul', 11: 'pranav', 12: 'haresh'}
print(d[10])
vipul
>>> del d[12]
>>> print(d)
{10: 'vipul', 11: 'pranav'}
```

---datatype over--

- **Escape chatacters in Strings:**

Escape Character	**Meaning**
\	New Line continuation
\\	Display a single line
\”	Display a double quote
\n	New line
\r	Enter
\t	Horizontal tap space
\v	Vertical tab

```
>>> str="this is \nPython"
>>> print(str)
this is
Python
```

- **Determining the Datatype of a Variable**

```
>>> a=10
>>> print(type(a))
<class 'int'>
>>> a=15.5
>>> print(type(a))
<class 'float'>
>>> a='hello'
>>> print(type(a))
<class 'str'>
>>> a="hello"
>>> print(type(a))
<class 'str'>
>>> a=(1,2,3,4)
>>> print(type(a))
<class 'tuple'>
>>> a=[1,2,3,4]
>>> print(type(a))
<class 'list'>
>>> a={1,2,3,4}
>>> print(type(a))
<class 'set'>
```

Extra:

```
>>> str='Vipul'
>>> str[0].isupper()
True
>>> str[1].isupper()
False
```

- **User Defined Datatypes:**

The datatype which are created by the programmers are called ‘user-defined’ data types. For example, an array, a class, or a module is user defined datatypes.

- **Constants in Python:**
  A constant is similar to a variable but its value cannot be modified or changed in the course of the program execution. A constant cannot allow changing its value. For example, in Mathematics, 'pi' value is 22/7 which never changes and hence it is a constant. In language like C and Java, defining constants is possible. But in Python, that is not possible. A programmer can indicate a variable as constant by writing its name in all capital letters. For example, MAX_VALUE is a constant. But its value can be changed.

- **Identifiers and Reserved Words:**
  An identifiers is a name that is given to a variable or function or class etc. Identifiers can include letters, numbers and the underscore character (_). They should always start with a nonnumeric characters. Special symbols such as ?, #, $, %, and @ are not allowed in identifiers. Some examples for identifiers are salary,name11,gross_income etc.

  We should also remember that Python is a case sensitive programming language. It means capital letters and small letters are identified separately by Python. For example, the names 'num' and 'Num' are treated as different names.

  Reserved words are the words that are already reserved for some particular purpose in the python language. The names of these reserved words should not be used as identifiers.

and	del	from	nonlocal	try
as	elif	global	not	while
assert	else	if	or	with
break	except	import	pass	yield
class	exec	in	print	False
continue	finally	is	raise	True
def	for	lambda	return	

- **Naming Conventions in Python:**

  - **Packages:**
    Package names should be written in all lower case letters. When multiple words are used for a name, we should separate them using an underscore (_).

- **Modules:**
  Modules name should be written in all lower case letters. When multiple words are used for a name, we should separate them using an underscore (_).
- **Classes:**
  Each word of a class name should start with a capital letters. Python's built-in class names use all lowercase words. When a class represents exception, then its name should end with a word 'Error'.
- **Global variables or module level variable:**
  Global variables names should be all lower case letters. When multiple words are used for a name, we should separate them using an underscore.
- **Functions:**
  Function names should be all lower case letters.
- **Methods:**
  Method names should be all lower case letters.

3. Operators in Python

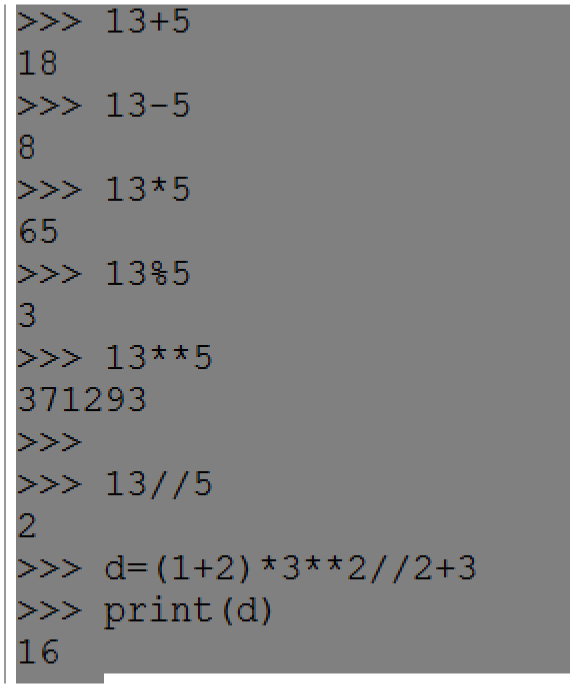
- Arithmetic operators
- Assignment operators
- Unary minus operator
- Relational operator
- Logical operators
- Boolean operators
- Bitwise operators
- Membership operators
- Identity operators

- **Arithmetic operators**

Let's assume a=13 and b=5

Operator	**Meaning**	**Example**	**Result**
+	Addition operator.	a+b	18
-	Subtraction operator	a-b	8
*	Multiplication operator	a*b	65
/	Division operator	a/b	2.6

%	Modulus operator	a%b	3
**	Exponent operator. Gives the value of a to the power of b.	a**b	371293
//	Integer division. Performs division and gives only integer quotient.	a//b	2



```
>>> 13+5
18
>>> 13-5
8
>>> 13*5
65
>>> 13%5
3
>>> 13**5
371293
>>>
>>> 13//5
2
>>> d=(1+2)*3**2//2+3
>>> print(d)
16
```

- **<u>Assignment operators:</u>**
  These operators are useful to store the right side value into a left side variable.
  Assume that the values x=20,y=10 and z=5

Operator	Example	Meaning	Result
=	z=x+y	Assignment operator. Stores right side value into left side variable. i.e. x+y is stored into z.	z=30
+=	z+=x	Addition assignment operator. z=z+x	z=25
-=	z-=x	Subtraction assignment operator. z=z-x	z=-15
*=	z*=x	Multiplication assignment operator. z=z*x	z=100
/=	z/=x	Division assignment operator. z=z/x	z=0.25
%/	Z%=x	Modulus assignment operator. z=z%x	z=5

**=	z**=y	Exponentiation assignment operator.	z=9765625
//=	z//=y	Floor division assignment operator.	z=0

```
>>> a=b=1
>>> print(a,b)
1 1
>>> a=1;b=2
>>> print(a,b)
1 2
>>> a,b=1,2
>>> print(a,b)
1 2
```

- **Unary minus operators:**

The unary minus operator is denoted by the symbol minus (-). When this operator is used before a variable, its value is negated. That means if the variable value is positive, it will be converted into negative and vice versa.

```
>>> n=10
>>> print(-n)
-10
>>> num=-10
>>> num=-num
>>> print(num)
10
```

- **Relational operators:**

Relational operators are used to compare two quantities. We can understand whether two values are same or which value one is bigger or which one is lesser, etc.

Assuming a=1, and b=2

Operators	**Example**	**Meaning**	**Result**
>	a>b	Greater than operator	False
>=	a>=b	Greater than or equal operator.	False
<	a<b	Less than operator	True
<=	a<=b	Less than or equal	True
==	a==b	Equals operator	False

!=	a!=b	Not equals operators	True

```
a=1;b=2
if(a>b):
   print("yes")
else:
   print("no")
```

```
>>> 1<2<3<4
True
>>> 1<2>3<4
False
>>> 4>2>=2>1
True
```

10<x>20 #display false

Here, 10 is less than 15 is true. But 15 is greater than 20 is false. Since we are getting true and false, the total result will be false. So, the point is this: in the chain of relational operators, if we get all true, then only the final result will be true. If any comparison yields false, then we get false as the final result.

- **<u>Relational operators:</u>**

Logical operators are useful to construct compound conditions. A compound condition is a combination of more than one simple condition. Each of the simple condition is evaluated to True or False and then the decision is taken to know whether the total condition is true or false. We should keep in mind that in case of logical operators false indicates 0 and true indicates any other number. Let's take x=1 and y=2 .

Operators	**Example**	**Meaning**	**Result**
And	x and y	And operator. if x is false , it returns x, otherwise it returns y.	2
Or	x or y	Or operators. If x is false, it returns y, otherwise it returns x.	1
Not	not x	Not operator. if x is false, it returns true, otherwise false.	False

```
x=1;y=2;z=3
if(x<y and y<z): print('yes')
else: print('no')
```

- **Boolean operators:**

  We know that there are two 'bool' type literals. They are True and False.

  Assume that x=True and y=False

Operators	Example	Meaning	Result
and	x and y	Boolean and operators.	False
or	x or y	Boolean or operator.	True
not	not x	Boolean not operator.	False

```
>>> a=True
>>> b=False
>>> a and a
True
>>> a and b
False
>>> b and b
False
>>> b or b
False
>>> b or a
True
```

- **Bitwise operators:**
  - Bitwise Complement Operator(~)
  - Bitwise AND operator(&)
  - Bitwise OR operator (|)
  - Bitwise Left Shift operator(<<)
  - Bitwise Right Shift operator (>>)

- **Membership operators:**

  The membership operators are useful to test for membership in a sequence such as strings, lists, tuples or dictionaries. For example, if an element is found in the sequence or not can be asserted using these operators. There are two membership operators.

  - in
  - not in

  The in operator:

  This operator returns true if an element is found in the specified sequence. If the element is not found in the sequence, then it returns false.

  The not in operator:

  This works in reverse manner for 'in' operator. this operator returns true if an element is not found in the sequence. If the element is found, then it returns false.

```
names=["vipul","rakesh","dinesh","dhamo"]
for name in names:
   print(name,end =" ")
```

```
postal={'delhi':110001,'chennai':600001,'amreli':365601}
for city in postal:
   print(city,postal[city],end=" ")
```

```
x=10
list=[10,20,30,40,50]
if(x in list):
   print("x is present")
else:
   print("x is not present")
```

```
x=100
list=[10,20,30,40,50]
if(x not in list):
   print("x is not present")
else:
   print("x is present")
```

- **identity operators:**

  These operators compare the memory locations of two objects. Hence, it is possible to know whether the two objects are same or not. The memory location of an object can be seen using the id() function. This function returns an integer number, called the ***identity number*** that internally represents the memory location of the object. For example, id(a).

```
>>> a=25
>>> b=25
>>> id(a)
1374868912
>>> id(b)
1374868912
>>> c=a+b
>>> id(c)
1374869312
```

There are two identity operators:

- is
- is not

The is operator

The 'is' operator is useful to compare whether two objects are same or not. It will internally compare the identity number of the objects. If the identity numbers of the objects are same, it will return True, otherwise, it return False.

The is not operator

The is not operator returns True, if the identity numbers of two objects being compared are not same. If they are same, then it will return false.

```
a=10
b=10
if(a is b):
   print("a and b have same identity")
else:
   print("a and b do not have same identity")
```

```
a=[10,20,30]
b=[10,20,30]
if(a is b):
   print("a and b have same identity")
else:
   print("a and b do not have same identity")
```

```
a=[10,20,30]
b=[10,20,30]
if(a==b):
   print("a and b are same")
else:
   print("a and b do not have same")
```

4. INPUT AND OUTPUT

The purpose of a computer is to process data and return results. It means that first of all, we should provide data to the computer. The data given to the computer is called *input*. The results returned by the computer are called *output*.

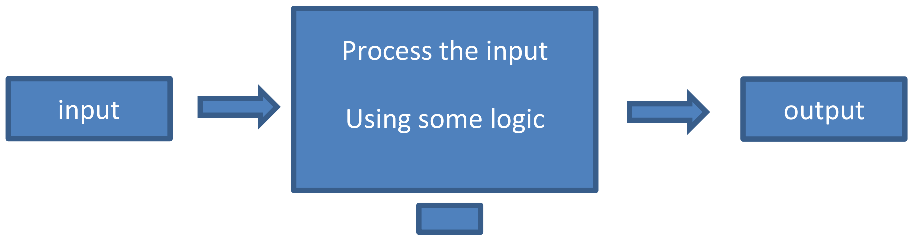


Fig: Processing input by the computer

To provide input to a computer, python provides some statements which are called input statements. Similarly, to display the output, there are output statements available in python. We should use some logic to convert the input into output.

- Output statements
- Input statements

Output statements:

To display output or result, python provides the print() function.

The print() statement:

When the print() function is called simply, it will throw the cursor to the next line. It means that a blank line will be displayed.

```
Print(“hello”)
Hello
Print(3*’hello’)
Hellohellohello
Print(“city name = “+”Mumbai”)
City name=Mumbai
Print(“city name=”,”mumbai”)
City name= Mumbai
```

The print(variable list) statement

```
a,b=2,4
print(a)
2
Print(a,b)
2 4
Print(a,b,sep=”,”)
2,4
Print(a,b,sep=”:”)
2:4

print("hello",end='\t')
print("Dear",end='\t')
print("how r u?",end='\t')
```

The print(object) statement

```
>>> lst=[10,'A','Hello']
>>> print(lst)
[10, 'A', 'Hello']
```

The print("string",variable list) statement

```
>>> a=2
>>> print(a,"is even number")
2 is even number
>>> print('you typed ',a,'as input')
you typed  2 as input
```

the print(formatting string) statement

```
 print("formatted string" % (variable list))

>>> x=10
>>> print("value=%i"%x)
value=10
>>> print("value=%d"%x)
value=10
>>> x,y=10,15
>>> print("x=%i y=%d"%(x,y))
x=10 y=15

>>> name="vipul"
>>> print("name=%s"%name)
name=vipul
>>> print("name %20s"%name)
name                vipul
>>> print("name %-20s"%name)
name vipul
>>> print("name (%20s)"%name)
name (               vipul)

>>> print("name %c %c "%(name[0],name[1]))

>>> print("name %s"%(name[0:3]))

>>> num=123.456789
>>> print("the value is : %f"%num)
the value is : 123.456789
>>> print("the value is : %8.2f"%num)
the value is :   123.46
```

```
>>> n1,n2,n3=1,2,3
>>> print("number1={0}".format(n1))
number1=1
>>> print("number1={0},number2={1},number3={2}".format(n1,n2,n3))
number1=1,number2=2,number3=3
>>> print("number1={1},number2={0},number3={2}".format(n1,n2,n3))
number1=2,number2=1,number3=3

>>> name,salary='ravi',12500.75
>>> print("hello {0}, your salary is {1}".format(name,salary))
hello ravi, your salary is 12500.75
>>> print("hello {n}, your salary is {s}".format(n=name,s=salary))
hello ravi, your salary is 12500.75
>>> print("hello {:s}, your salary is {:.2f}".format(name,salary))
hello ravi, your salary is 12500.75
>>> print("hello %s, your salary is %.2f"%(name,salary))
hello ravi, your salary is 12500.75
```

<u>input statements:</u>

to accept input from keyboard, python provides the input() function.

```
>>> str=input()    #this will wait till we enter a string
vipul
>>> print(str)
vipul
```

it is a better idea to display a message to the user so that the user understands what to enter.

```
>>> str=input("Enter your name: ")
Enter your name: vipul
>>> print(str)
Vipul
```

Once the value comes into the variable ‘str’ it can be converted into ‘int’ or ‘float’ etc.

```
>>> str=input("Enter a number: ")
Enter a number: 125
>>> x=int(str)
>>> print(x)
125
```

```
>>> a=input("Enter no1 :")
Enter no1 :100
>>> b=input("Enter no2 :")
Enter no2 :200
>>> c=a+b
>>> c
'100200'
>>> print(c)
100200
>>> c=int(a)+int(b)
>>> c
300

>>> a=float(input("Enter percentage : "))
Enter percentage : 78.85
>>> print(a)
```

Prog1:
A Python program to accept a string from keyboard and display it.

```
#accepting a string from keyboard
str=input("Enter a string: ")
print("u enterered:",str)
```

```
Enter a string: ksc
u enterered: ksc
```

Prog2:
A Python program to accept a float from keyboard and display it.

```
#accepting float number from keyboard
x=float(input("Enter a number: "))
print("U enterered: ",x)
```

Prog3:
A Python program to accept two integer numbers from keyboard.

```
#accepting two numbers from keyboard
x=int(input("Enter first number: "))
y=int(input("Enter second number: "))
print("U entered: ",x,y,sep=",")
```

Prog4:
A Python program to accept two integer numbers and sum it.

```
#accepting two numbers from keyboard
x=int(input("Enter first number: "))
y=int(input("Enter second number: "))
print("The sum of ",x,"and ",y," is ",x+y)
```

or

```
#accepting two numbers from keyboard
x=int(input("Enter first number: "))
y=int(input("Enter second number: "))
print("The sum of {} and {} is {}".format(x,y,x+y))
```

```
#accepting two numbers from keyboard
x=int(input("Enter first number: "))
y=int(input("Enter second number: "))
print("The sum of {0} and {1} is {2}".format(x,y,x+y))
print("The multiplication of {0} and {1} is {2}".format(x,y,x*y))
print("The subtraction of {0} and {1} is {2}".format(x,y,x-y))
print("The division of {0} and {1} is {2}".format(x,y,x/y))
```

prog5:A python program to accept 3 integers in the same line and display their sum.

```
        #accepting 3 numbers separated by space
        var1,var2,var3=[int(x) for x in input("Enter three numbers:").split()]
        print("sum = ",var1+var2+var3)
```

In the previous statement, the input() function will display the message ‘Enter three numbers” to the user. When the user enters three values, they are accepted as strings. These strings are divided wherever a space is found by split() method. So, we get three strings as elements of a list. These strings are read by for loop and converted into integers by the int() function. These integers are finally stored into var1,var2,var3.

The split() method by default splits the values where a space is found. Hence, while entering the numbers, the user should separate them using a space. The square brackets [] around the total expression indicates that the input is accepted as elements of a list.

```
#accepting 3 numbers separated by space
var1,var2,var3=[int(x) for x in input("Enter three numbers:").split(',')]
print("sum = ",var1+var2+var3)
```

prog6: A Python program a group of strings from keyboard

```
lst=[x for x in input('Enter strings: ').split(',')]
print('you entered:\n',lst)
```

The eval() function takes a string and evaluates the result of the string by taking it as a Python expression. For example, let’s take a string : “a+b-4” where a=5,b=10. If we pass the string to the eval() function, it will evaluate the string and returns the result.

```
a,b=5,10
result=eval("a+b-4")
print(result)
```

prog7: Evaluating an expression entered from keyboard.

```
#using eval() along with input() function
x=eval(input("Enter an expression:"))
print("result = %d"%x)
```

output:

```
Enter an expression:10+15-4
result = 21
```

prog8: A Python program to accept a list and display it

```
lst=eval(input("Enter a list:"))
print("list = ",lst)
```

output

```
Enter a list:["ajay","vipul","rushi"]
list =  ['ajay', 'vipul', 'rushi']
```

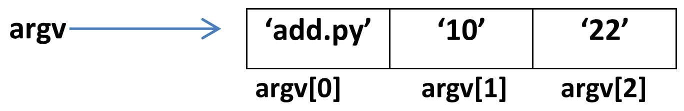
- **<u>Command Line Arguments:</u>**

We can design out program in such a way that we can pass inputs to the program when we give run command. We can supply the two numbers as input to the program at the time of running the program at command prompt as:

```
c:\python add.py 10 22
```

```
sum=32
```

These arguments are stored by default in the form of strings in a list with the name 'argv' which is available in sys module. Since argv is a list that contains all the values passed to the program, argv[0] represents the name of the program, argv[1] represents the first value, argv[2] represents the second value and so on.



```
#to display command line args. save this as cmd.py.
import sys

n=len(sys.argv)  #n is the number of arguments
args=sys.argv   # args list contains arguments
print("no. of command line args=",n)
print("the args are:",args)
print("The args one by one:")
for a in args:
   print(a)
```

prog9:A Python program to find sum of two numbers using command line arguments.

```
#to add two numbers.
import sys
#coverts args into integers and add them
sum=int(sys.argv[1])+int(sys.argv[2])
print('Sum = ',sum)
```

prog10: A Python program to find the sum of even numbers using command line arguments.

```
#to find sum of even numbers
import sys
#read command line arguments except the program name
args=sys.argv[1:]
print(args)

sum=0
#find sum of even arguments
for a in args:
   x=int(a)
   if x%2==0:
      sum+=x

print('sum of evens = ',sum)
```

prog11: A program to calculate the area of circle.

```
import math
r=float(input("Enter radius-"))
area=math.pi*r**2
print("Area of circle = ",area)
print('Area of circle {:0.2f}'.format(area))
```

5. Control Statements

- **<u>Control Statements:</u>**
  Control statements are statements which control or change the flow of execution.
    - if statement
    - if…else statement
    - if…elif…else statement
    - while loop
    - for loop
    - else suite

- break statement
- continue statement
- pass statement
- assert statement
- return statement

The if Statement:

This statement is used to execute one or more statement depending on whether a condition is true or not.

Syntax:

```
if condition:
    statements
```

```
#understanding if statement
num=1
if num==1:
   print("one")
   print("second")
   print("third")
```

Indentation in if…statements:

```
if x==1:
__print('a')
__print('b')
__if y==2:
____print('c')
____print('d')
print('end')
```

The if…else Statement:

The if..else statement executes a group of statements when a condition is True; otherwise, it will execute another group of statements. The syntax of if…else statement is given below:

```
if condition:
    statements1
else:
    statements2
```

```
x=int(input("Enter a number: "))
if x%2==0:
   print(x," is even number")
else:
   print(x," is odd number")
```

The if…elif…else Statement:

Sometimes, the programmer has to test multiple conditions and execute statements depending on those conditions.

```
if condition1:
   Statements1
elif condition2:
  statements2
elif condition3:
  statements3
else:
  statements4
```

prog: A Python program to enter three no. from user and check biggest number from it.

```
a=int(input("Enter number1="))
b=int(input("Enter number2="))
c=int(input("Enter number3="))
if a>b and a>c:
   print("a is biggest")
elif b>a and b>c:
   print("b is biggest")
else:
   print("c is biggest")
```

prog: A Python program to know if a given number is zero,positive, or negative. (Lab)

The while loop:

A statement is executed only one from top to bottom. For example, 'if' is a statement that is executed by Python interpreter only once. But a loop is useful to execute repeatedly. For example, while and for loops in Python. They are useful to execute a group of statements repeatedly several times.

```
while condition:
   statements
x=1
while x<=10:
   print(x)
   x+=1
print("End")
```

prog: A Python program to display even numbers between 100 and 200.

```
          x=100
         while x>=100 and x<=200:
                   print(" ",x,end="")
                   x+=2
```

prog: A Python program to display even number between m to n

```
m,n=[int(i) for i in input("Enter minimum and maximum range:").split(",")]
x=m
if x%2!=0:
   x=x+1
while x>=m and x<=n:
   print(" ",x,end="")
   x+=2
```

The for loop:

The for loop is useful to iterate over the elements of a sequence. It means, the for loop can be used to execute a group of statements repeatedly depending upon the number of element in the sequence. The for loop can work with sequence like string, list, tuple, range etc.

```
For var in sequence:
  statements
```

prog: A python program to display characters of a string using for loop.

```
str='hello'
for ch in str:
   print(ch)
```

prog: A Python program to display odd numbers from 1 to 10 using range object.

```
for i in range(1,10,2):
    print(i)
```

prog: A Python program to display numbers from 10 to 1 in descending order.

```
for i in range(10,0,-1):
    print(i)
```

prog: A Python program to display and find the sum of a list of numbers using for loop

```
list=[10,20,30,40,50]
sum=0
for i in list:
    print(i)
    sum=sum+i
print("The sum is = ",sum)
```

Nested Loop:

```
for i in range(3):
    for j in range(4):
        print('i=',i,'\t','j=',j)
```

prog: A Python program to print Pyramid using Nested for loop.

```
for i in range(1,11):

    for j in range(1,i+1):

        print('* ',end='')

    print()
```

The else suite

for with else	**while with else**

for(var in sequence):
    statements
	while(condition):
	    statements

else:
    statements
	else:
	    statements

The statements written after 'else' are called suite. The else suite will be always execute irrespective of the statements in the loop are executed or not.

```
for i in range(5):
   print("Yes")
else:
   print("No")
```

it, means the for loop statement is executed and also the else suite is executed. For example,

```
for i in range(0):
   print("Yes")
else:
   print("No")
```

So, the point is this: the else suite is always executed. But then where is this else suite useful? Sometimes, we write programs where searching for an element is done in the sequence. When the element is not found, we can indicate that in the else suite easily. In this case, else with for loop or while loop - is very convenient.

<u>The break statement</u>

The break statement can be used inside a for loop or while loop to come out of the loop. When 'break' is executed, the python interpreter jumps out of the loop to process the next statement in the program.

Prog: A Python program to search for an element in the list of elements.

```
group1=[1,2,3,4,5]
search=int(input("Enter element to search: "))
for element in group1:
   if search==element:
      print("Element found in group1")
      break
else:
   print("Element not found in group")
```

<u>The continue statement</u>

The continue statement is used in a loop to go back to the beginning of the loop. It means, when continue is executed, the next repetition will start.

<u>The pass statement:</u>

The pass statement does not do anything. It is used with 'if' statement or inside a loop to represent no operation. We use pass statement when we need a statement syntactically but we do not want to do any operation.

Prog: A Program to know that pass does nothing.

```
x=0
while x<10:
   x+=1
   if x>5:
      pass
   print('x=',x)
print("Out of loop")
```

A more meaningful usage of the 'pass' statement is to inform the Python interpreter not to do anything when we are not interested in the result.

Prog: A Python program to retrieve only negative numbers from a list of numbers.

```
num=[1,2,3,-4,-5,-6,-7,8,9]
for i in num:
   if(i>0):
      pass
   else:
      print(i)
```

<u>The assert statement</u>

The assert statement is useful to check if a particular condition is fulfilled or not. The sysntax is as follows.

Assert expression, message

In the above syntax, the 'message' is not compulsory. Let's take an example. If we want to assure that the user should enter only a number greater than 0, we can use assert statement as:

assert x>0, "wrong input entered"

prog:A program to assert that the user enters a number greater than zero.

```
x=int(input('Enter a number greater than 0:'))
assert x>0,"Wrong input entered"
print("u entered: ",x)
```

The 'assertion error' shown in the above output is called an exception. An exception is an error that occurs during runtime. To avoid such type of exceptions, we can handle them using 'try…except' statement. After 'try', we use 'assert' and after 'except', we write the exception name to be handled.

```
x=int(input('Enter a number greater than 0:'))
try:
   assert(x>0)   #exception may occur here
   print("u entered : ",x)
except AssertionError:
   print("wrong input entered")
```

The return statement

A function represents a group of statements to perform a task. The purpose of a function is to perform some task and in many cases a function returns the result. A function starts with the keyword ***def*** that represents the definition of the function. After, 'def', the function should be written.

```
def sum(a,b):
   function body
```

```
def sum(a,b):
    print(a+b)
```

sum(5,10)

In the above program, the sum() function is not returning the computed result. In some cases, it is highly useful to write the function as if is returning the result. This is done using 'return' statement.

```
return expression
```

```
def sum(a,b):
   return a+b  #result is returned from here

#call sum() and 5 and 10
#get the returned result into res
res=sum(5,10)
print('the result is',res)
```

prog: A Python program for Fibonacci series.

```
#program to display fibonacci series
n=int(input('how many fibonaccis? '))
f1=0
f2=1
c=2
if n==1:
   print(f1)
elif n==2:
   print(f1,'\n',f2,sep='')
else:
   print(f1,'\n',f2,sep='')
   while c<n:
      f=f1+f2
      print(f)
      f1=f2
      f2=f
      c+=1
```

6. Arrays in Python

An array is an object that stores a group of elements of same datatype. The main advantage of any array is to store and process a group of elements easily.

Creating an Array:

Arrayname=array(type code,[elements])

The type code ‘i’, represents integer type array where we can store integer numbers. If the type code is ‘f’ then it represents float type array where we can store numbers with decimal point.

Typecode	C Type	Minimum size in bytes
‘b’	Signed integer	1
‘B’	Unsigned integer	1
‘i’	Signed integer	2
‘I’	Unsigned integer	2
‘f’	Floating point	4
‘d’	Double precision floating point	8
‘u’	Unicode character	2

a = array(‘i’,[4,6,2,9])   -> integer type array

a = array(‘d’,[1.5,-2.2,3,5.75])   -> float type array

Importing the array module:

Import array

When we import the array module, we are able to get the ‘array’ class of that module that helps us to create an array.

a=array.array(‘i’,[4,2,6,8])

or

import array as ar

a=ar.array(‘i’,[4,2,6,8])

or

from array import *

Observe the ‘*’ symbol that represents ‘all’. The meaning of this statement is this: import all(classes, objects, variables etc) from the array module into program. So there is no need to mention the module name before our array name while creating it.

```
import array
a=array.array('i',[2,4,6,8])
print("the array elements are")
for element in a:
   print(element)
```

```
from array import *
a=array('u',['a','b','c'])
print("the array elements are")
for element in a:
   print(element)
```

```
from array import *
x=array('i',[10,20,30,40,50,60,70])
for i in x[2:5]:                    #  :2   0:     -2:     -2:      :-2
   print(i)
```

```
name=u'\u0915\u094b\u0930 \u092a\u0948\u0925\u0964\u0928'
print(name)

name=u'\u0A95\u0AAE\u0ABE\u0AA3\u0AC0'
print(name)
```

String and Character

```
str='corepython'
print(str)
print(str[0:9:1])  #access string from 0th to 8th element in steps of 1
print(str[0:9:2])  #access string from 0th to last character
print(str[::])     #access string from 0th to last character
print(str[::2])    #access entire string in steps of 2
print(str[2::])    #access string from str[2] to ending
print(str[-4:-1])  #access from str[-4] tp str[-2] from left to right in str
print(str[-6::])   #access from str[-6] till end of the string
print(str.upper()) #changing case of a string
print(str.lower())
print(str.swapcase())
print(str.title())
```

output

```
corepython
corepytho
crpto
corepython
```

```
crpto
repython
tho
python
COREPYTHON
corepython
COREPYTHON
Corepython
```

7. Functions

- Functions are important in programming because they are used to process data, make calculations or perform any task which is required in the software development.
- Once a function is written, it can be reused as and when required. So functions are also called reusable code.
- Functions provide modularity for programming. A module represents a part of the program. Usually, a programmer divides the main task into smaller sub tasks called modules. To represents each module, the programmer will develop a separate function.

Difference between a Function and a Method.

A function is called using its name. When a function is written inside a class, it becomes a 'method'. A method is called using one of the following ways.

```
objectname.methodname()
classname.methodname()
```

Defining a Function

We can define a function using the keyword *def* followed by function name. After the function name, we should write parentheses() which may contain paramerters. Consider the syntax of function.

Function definition.

```
def functionname(para1,para2,…):
    """function docstring"""
    Statements
```

Example:

```
def sum(a,b):
     “””This function finds sum of two numbers”””
     c=a+b
     print(c)
```

Calling function

sum(5,10)

Returning Results from a function

```
def sum(a,b):
    return a+b  #result is returned from here

#call sum() and 5 and 10
#get the returned result into res
res=sum(5,10)
print('the result is',res)
```

prog: A function to test whether a number is even or odd

```
#a function to test whether a number is odd or even
def even_odd(num):
    """to know num is even or odd"""
    if num%2==0:
        print(num," is even")
    else:
        print(num," is odd")

#call the function
even_odd(10)
even_odd(13)
```

prog: A Python program to calculate factorial values of numbers

```
# a function to calculate factorial value
def fact(n):
    """to find factorial value"""
    prod=1
    while n>=1:
        prod*=n
        n-=1
    return prod

#display factorial of first 10 numbers
#call fact() function and pass the numbers from 1 to 10
for i in range(1,11):
    print("factorial of {} is {}".format(i,fact(i)))
```

Returning Multiple Values from a Function

```
# a function that returns multiple results
def sum_sub_mul_div(a,b):
    c=a+b
    d=a-b
    e=a*b
    f=a/b
    return c,d,e,f

#get resuts from sum_sub_mul_div() function and store into t
t=sum_sub_mul_div(10,5)

#display the results using for loop
print("The results are")
for i in t:
    print(i,end=", ")
```

Functions are first class objects:

In Python functions are considered as first class objects. It means we can use functions as perfect objects. In fact when we create a function, the Python interpreter internally creates an object. Since functions are objects, we can pass a function to another function just like we pass an object(or value) to a function.

```
#assign a function to a varible
def display(str):
   return 'Welcome'+str

#assign function to variable x
x=display(" To ksc")
print(x)
```

Formal and Actual Arguments:

When a function is defined, it may have some parameter. These parameters are useful to receive values from outside of the function. They are called ‘formal arguments’. When we call the function, we should pass data or values to the function. These values are called ‘actual arguments’.

```
def sum(a,b):   #a,b are formal arguments
   c=a+b
   print(c)

#call the function
x=10;y=15
sum(x,y)       #x,y are actual arguments
```

The actual arguments used in a function call are of 4 types:

- **Positional Arguments**
- **Keyword Arguments**
- **Default Arguments**
- **Variable length arguments**

1. **Positional Arguments:**
   These are the arguments passed to a function in correct positional order. Here, the number of arguments and their positions in the function definition should match exactly with the number and position of the argument in the function call.

Prog: A Python program to understand the positional arguments of a function.

```
#positional arguments demo
def attach(s1,s2):
   """to join s1 and s2 and display total string"""
   s3=s1+s2
   print("total string: "+s3)

#call attach() and pass 2 strings
attach("new","york")    #positional arguments
```

Keyword Arguments:

Keyword arguments are arguments that identify the parameters by their names. For example, the definition of a function that displays grocery item and its price can be written as:

Prog:A Python program to understand the keyword argument of a function.

```
#keyword arguments demo
def grocery(item,price):
   """to display the given arguments"""
   print("Item=",item)
   print("Price=",price)

#call grocery() and pass 2 arguments
grocery(item="sugar",price=48.50) #keyword arguments
grocery(price=88.00,item="Oil")  #keyword arguments
```

Default Arguments:

We can mention some default value for the function parameters in the definition.

Prog: A Python program to understand the use of default arguments in a function.

```
#default arguments demo
def grocery(item,price=88.00):
   """to display the given arguments"""
   print("item=%s"%item)
   print("price=%.2f"%price)

#call grocery() and pass 2 arguments
grocery(item="sugar",price=48.50) #keyword arguments
grocery(item="Oil")  #keyword arguments
```

Variable length Arguments:

Sometimes the programmer does not know how many values a function may receive. In that case, the programmer cannot decide how many arguments to be given in the function definition. For example, if the programmer is writing a function to add two numbers, he can write:

add(a,b)

But, the user who is using this function may want to use this function to find sum of three numbers.

add(10,15,20)

Then the add() function will fail and error will displayed. If the programmer wants to develop a function that can accept 'n' arguments, that is also possible in Python. For this purpose, a variable length argument is used in the function definition. The variable length argument is written with a '*' symbol before it in the function definition. Here, farg is the formal argument and *args represents variable length argument.

```
def add(farg,*args):
def add(farg,*args):   #*args can take 0 or more values
   """to add given numbers"""
   print("Formal arguments = ",farg)
   sum=0
   for i in args:
      sum+=i
   print("sum of all numbers= ",(farg+sum))
#call add() and pass arguments
add(5,10)
add(15,20,25,30)
```

Local and Global Variables:

When we declare a variable inside a function, it becomes a local variable. A local variable is a variable whose scope is limited only to that function where it is created. That means the local variable value is available only in that function and not outside of that function.

```
#local variable in a function
def myfunction():
   a=1    #this is local var
```

```
    a+=1
    print(a)

myfunction()
print(a) #error, not available
```

When a variable is declared above a function, it becomes global variable. Such variable are available to all the functions which are written after it.

```
#global variable in a function
a=1 #this is global var
def myfunction():
    b=2     #this is local var
    print("a = ",a)  #display global var
    print("b = ",b)  #display local var

myfunction()
print(a) #available
print(b) #error,not available
```

<u>Recursive Functions:</u>
A function that calls itself is known as 'recursive function'. For example, we can write the factorial of 3 as:

```
#recursive function to calculate factorial
def factorial(n):
    """to find factorial of n"""
    if n==0:
        result=1
    else:
        result=n*factorial(n-1)
    return result
#find factorial values for first 10 numbers
for i in range(1,11):
    print("Factorial of {} is {}".format(i,factorial(i)))
```

The Special Variable __name__

When a program is executed in python, there is a special variable internally created by the name '_name_'. This variable stores information regarding whether the program is executed as an individual program or as a module. When the program is executed directly, the Python interpreter stores the value ‘_main_’ into this variable. When the program is imported as a module into another program, then Python interpreter stores the module name into this variable. Thus, by observing the value of the variable ‘_name_’ we can understand how the program is executed.

one.py

```
#Python program to display a message, save this as one.py
def display():
   print("Hello Python")

if __name__== '__main__':
   display()
   print("This code is run as a program")
else:
   print("This code is run as a module")
```

output:

```
Hello Python
This code is run as a program
```

two.py

```
#in this program one.py is imported as a module. save this as two.py
import one
one.display() #call module one's display() function
```

output:

```
This code is run as a module
Hello Python
```

Creating our own Modules in Python

A module represents a group of classes, methods, functions and variables. While we are developing software, there may be several classes, methods and functions. We should first group them depending on their relationship into various modules and later use these modules in the other programs. It means, when a module is developed, it can be reused in any program that needs that module.

In Python, we have several built-in modules like sys, io, time etc. Just like these modules, we can also create our own modules and use them 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.

```
#save this code as employee.py
#to calculate dearness allowance
def da(basic):
   """da is 80% of basic salary"""
   da=basic*80/100
   return da

#to calculate house rent allowance
def hra(basic):
   """hra is 15% of basic salary"""
   hra=basic*15/100
   return hra

#to calculate provident fund amount
def pf(basic):
   """pf is 12% of basic salary"""
   pf=basic*12/100
   return pf

#to calculate income tax payable by employee
def itax(gross):
   """tax is calculated at 10% on gross"""
   tax=gross*0.1
   return tax
```

Now, we have our own module by the name ‘employee.py’. this module contains 4 functions which can be used in any program by importing this module. To import the module, we can write.

```
import employee
```

In this case, we have to refer to the functions by adding the module name as employee.da(), employee.hra(), employee.pf(), employee.itax(). This is a bit cumbersome and hence we use another type of import statement as:

```
from employee import *
```

```
#using employee module to calculate gross and net salaries of an employee

from employee import *

#calculate gross salary of employee by taking basic

basic=float(input("enter basic salary: "))

#calculate gross salary

gross=basic+da(basic)+hra(basic)

print("YOur gross salary :{:10.2f}".format(gross))

#calculate net salary

net=gross-pf(basic)-itax(gross)

print("Your net salary : {:10.2f}".format(net))
```

8. Lists and Tuples

List:

A list is similar to an array that consists of a group of elements or items.

Prog: A Python program to create lists with different types of elements

```
#a general way to create list

#create a list with integer numbers

num=[10,20,30,40,50]

print("TOtal list=",num)

print("First = %d, Last=%d"%(num[0],num[4]))

#create a list with strings

names=["Vipul","Rakesh","Ravi","Kamlesh"]

print("Total List=",names)

print("First = %s, Last=%s"%(names[0],names[3]))
```

Creating List Using range()

```
lst=list(range(4,9,2))

print(lst)



lst2=range(5,10)

for i in lst2:

    print(i,", ",end="")

print()
```

Updating The elements of a List

```
lst=list(range(1,5))
print(lst)

lst.append(9)
print(lst)

lst[1]=8  #update 1st element of lst
print(lst)

lst[1:3]=10,11   #update 1st and 2nd elements of lst
print(lst)

lst.remove(9)   #delete 9 from lst
print(lst)

del lst[1]    #delete 1st element from lst
print(lst)

lst.reverse()
print(lst)
```

prog: A Python program to understand list processing methods.

```
#Python's list methods
num=[10,20,30,40,50]

n=len(num)
print("No of elements in num: ",n)

num.append(60)
print("Num after appending 60: ",num)

num.insert(0,5)
print("Num after insering 5 at 0th position: ",num)

num1=num.copy()
print("Newly created list num1: ",num1)

num.extend(num1)
print("num after appending num1:",num)

n=num.count(50)
print("No of times 50 found in the list num: ",n)

num.remove(50)
print("Num after removing 50:",num)

num.pop()
print("num after removing ending element:",num)

num.sort()
print("num after sorting:",num)

num.reverse()
print("Num after reversing:",num)

num.clear()
print("num after removing all elements:",num)
```

<u>Tuple:</u>

A tuple is a Python sequence which stores a group of elements or items. Tuples are similar to lists but the main difference is tuple are immutable whereas lists are mutable. Since tuples are immutable, once we create a tuple we cannot modify its elements. Hence we cannot perform operations like append(), extend(), insert(), remove(), pop(), clear().

tup1=()   #empty tuple

tup2=(10)   # tuple with one element

tup3=(10,20,-30.1,40.5,’hydrabad’,’new delhi’)

tup4=1,2,3,4   #no braces

Basic operations on Tuples:

The 5 Basic operations: finding length, concatenation, repetition, membership and iteration operations can be performed on any sequence may be it is a string, list, tuple or dictionary.

```
student=(10,"hiren",50,50,65,61,70)
print(len(student))

fees=(25000.00,)*4
print(fees)

student1=student+fees
print(student1)

name="hiren"
print(name in student)
```

Inserting Elements in a Tuple:

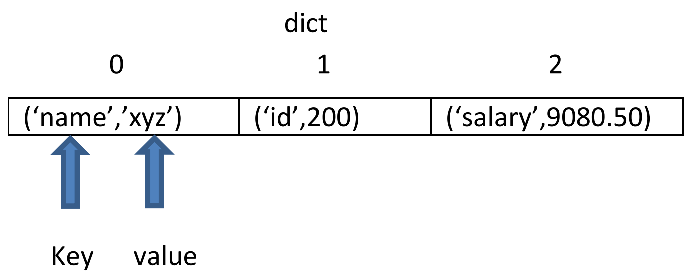
```
#inserting a new element into a tuple
names=('a','b','c','d')
print(names)

#accept new name and position number
lst=[input("Enter a new name:")]
new=tuple(lst)
pos=int(input("Enter position no:"))

#copy from 0th to pos-2 into another tuple names1
names1=names[0:pos-1]

#concatenate new element at pos-1
```

```
names1=names1+new

#concatenate the ramaining elements of names from pos-1 till end
names=names1+names[pos-1:]
print(names)
```

Dictionaries:

A dictionary represents a group of elements arranged in the form of key-value pairs. In the dictionary, the first element is considered as ‘key’ and the immediate next element is taken as its ‘value’. The key and its value are separated by a colon.

```
dict={‘name’:’xyz’,’id’:200,’salary’:9080.50}
```



```
dict={'Name':'xyz','Id':200,'Salary':9080.50}
print("name of employee=",dict['Name'])
print("id number=",dict['Id'])
print("salary=",dict['Salary'])
```

- **Operations on Dictionaries:**

```
n=len(dict)
print("No of key pairs=",n)
```

We can modify the existing value of a key by assigning a new value.

```
dict[‘Salary’]=10500.00
```

prog: A Python program to retrieve keys, values, and key value pairs from a dictionary.

```
dict={'Name':'xyz','Id':200,'Salary':9080.50}
print(dict)
print("key in dict=",dict.keys())    #display only keys
print("Values in dict=",dict.values())  #display only values
print("items in dict=",dict.items())    #display both key and value pairs as tuple
```

prog: A Python program to create a dictionary and find the sum of values.

```
dict=eval(input("Enter elements in {}: "))

#find the sum of values
s=sum(dict.values())
print("Sum of values in the dictionary: ",s)
```

Output:

```
Enter elements in {}: {'A':10,'B':20,'C':30}
Sum of values in the dictionary:  60
```