Course Name: Problem Solving Using Python
Course Code: DSC-C-BCA-352T

Unit - 1

PYTHON INTRODUCTION

What is Python?

- Python is a popular programming language. It was created by **Guido van Rossum**, and released in 1991.
- Python is one of the most popular programming languages.
- It’s simple to use, packed with features and supported by a wide range of libraries and frameworks.
- Its clean syntax makes it beginner-friendly.

Python is:

- A high-level language, used in web development, data science, automation, AI and more.
- Known for its readability, which means code is easier to write, understand and maintain.
- Backed by library support, so we don’t have to build everything from scratch, there’s probably a library that already does what we need.

Features in Python

- In this section we will see what are the features of Python programming language:

1. Free and Open Source

Python language is freely available at the official website. Download Python Since it is open-source, this means that source code is also available to the public. So you can download it, use it as well as share it.

2. Easy to code

Python is a high-level programming language. Python is very easy to learn the language as compared to other languages like C, C#, JavaScript , Java, etc. It is very easy to code in the Python language and anybody can learn Python basics in a few hours or days. It is also a developer-friendly language.

3. Easy to Read

As you will see, learning Python is quite simple. As was already established, Python's syntax is really straightforward. The code block is defined by the indentations rather than by semicolons or brackets.

4. Object-Oriented Language

One of the key features of Python is Object-Oriented programming. Python supports object-oriented language and concepts of classes, object encapsulation, etc.

5. GUI Programming Support

Graphical User interfaces can be made using a module such as PyQt5, PyQt4, wxPython, or Tk in Python. PyQt5 is the most popular option for creating graphical apps with Python.

6. High-Level Language

Python is a high-level language. When we write programs in Python, we do not need to remember the system architecture, nor do we need to manage the memory.

7. Python is a Portable language

Python language is also a portable language. For example, if we have Python code for Windows and if we want to run this code on other platforms such as Linux, Unix, and Mac then we do not need to change it, we can run this code on any platform.

8. Python is an Integrated language

Python is also an Integrated language because we can easily integrate Python with other languages like C, C++, etc.

9. Interpreted Language:

Python is an Interpreted Language because Python code is executed line by line at a time. like other languages C, C++, Java, etc. there is no need to compile Python code this makes it easier to debug our code. The source code of Python is converted into an immediate form called **bytecode**.

10. Large Standard Library

Python has a large standard library that provides a rich set of modules and functions so you do not have to write your own code for every single thing. There are many libraries present in Python such as regular expressions, unit-testing, web browsers, etc.

11. Dynamically Typed Language

Python is a dynamically-typed language. That means the type (for example- int, double, long, etc.) for a variable is decided at run time not in advance because of this feature we don't need to specify the type of variable.

12. Frontend and backend development

With a new project py script, you can run and write Python codes in HTML with the help of some simple tags <py-script>, <py-env>, etc. This will help you do frontend development work in Python like javascript. Backend is the strong forte of Python it's extensively used for this work cause of its frameworks like Django and Flask.

13. Allocating Memory Dynamically

In Python, the variable data type does not need to be specified. The memory is automatically allocated to a variable at runtime when it is given a value. Developers do not need to write int y = 18 if the integer value 15 is set to y. You may just type y=18.

Flavors of Python

- "Flavors of Python" generally refers to different implementations or distributions of the Python programming language, each with its own unique characteristics and use cases. These flavors often target specific platforms or integrate with other technologies.
- Here's a breakdown of some key Python flavors:

1. CPython:

This is the standard implementation of Python, **written in C**, and is the most widely used.

It's the reference implementation provided by the Python Software Foundation.

CPython is well-suited for general-purpose programming and can be used with **C language** applications.

2. Jython (or JPython):

Jython compiles Python code into Java bytecode, allowing it to run on the **Java Virtual Machine (JVM**).

This enables Python code to interact with Java libraries and frameworks.

It supports both static and dynamic compilation.

3. IronPython:

IronPython is a Python implementation targeting the **.NET framework** and Mono.

It allows Python code to be used with **C#** and other **.NET languages**.

4. PyPy:

PyPy is a **fast**, **Just-In-Time (JIT)** compiled implementation of Python.

It aims to improve Python's performance through JIT compilation.

•

5. Cython:

Cython is a language that is a superset of Python and allows for writing C extensions for Python.

It helps in optimizing Python code for performance-critical sections.

6. MicroPython:

MicroPython is a lean and efficient implementation of Python 3 designed for **microcontrollers** and **embedded systems**.

7. RubyPython:

RubyPython is a way to integrate Python code with **Ruby**, allowing them to interact.

8. Anaconda Python:

Anaconda is a popular distribution of Python, particularly for scientific computing and data science.

It includes a wide range of pre-installed packages and tools for **data analysis, machine learning, etc.**

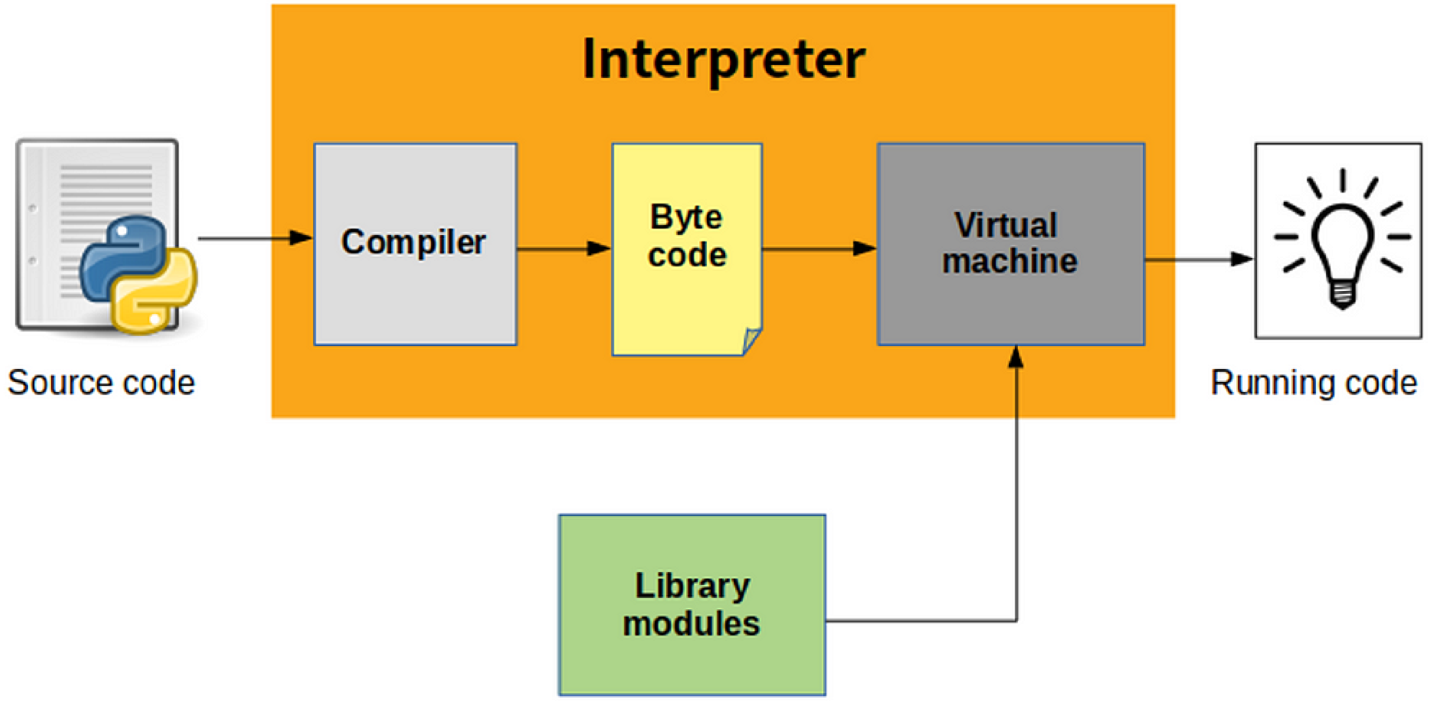
9. Stackless Python:

Stackless Python is a modified version of **CPython** that provides features like **microthreads** and continuations.

10. RustPython:

RustPython is an implementation of Python written in **Rust**.

PVM(Python Virtual Machine)



Python Virtual Machine

- The Python Virtual Machine, often referred to as the Python interpreter, is responsible for executing Python code. It serves as an abstraction layer between the Python bytecode and the underlying hardware, providing a consistent environment for running Python programs across different platforms. The Python VM is implemented in CPython, the reference implementation of Python.

Architecture of the Python Virtual Machine

Here, is the the architecture of the Python Virtual Machine:

Bytecode Generation:

Before execution, Python source code is compiled into bytecode. This bytecode is a platform-independent intermediate representation of the Python code.

The Python VM loads bytecode from the compiled .pyc files or directly from memory if the code is generated dynamically.

Interpreter Loop:

The heart of the Python VM is the interpreter loop. It fetches bytecode instructions, decodes them, and executes them sequentially.

Python Object Model:

The Python VM maintains a Python object model to represent data types, such as integers, strings, lists, and custom objects. It manages the creation, manipulation, and destruction of these objects during execution.

Memory Management:

The Python VM handles memory allocation and garbage collection to manage the memory used by Python objects dynamically.

Python Virtual Machine (PVM)

- We know that computers understand only machine code that comprises 1s and Os. Since computer understands only machine code, it is imperative that we should convert any program into machine code before it is submitted to the computer for execution.
- For this purpose, we should take the help of a compiler.
- **A compiler normally converts the program source code into machine code.**
- A Python compiler does the same task but in a slightly different manner.
- It converts the program source code into another code, called ***byte code***.
- Each Python program statement is converted into a group of byte code instructions. Then what is byte code? Byte code represents the fixed set of instructions created by Python developers representing all types of operations. The size of each byte code instruction is 1 byte bits and hence these are called byte code instructions.

- Python organization says that there may be newer instructions added to the existing byte code instructions from time to time.
- We can find byte code instructions in the .pyc file. the role of virtual machine in converting byte code instructions into machine code:
- The role of **Python Virtual Machine (PVM)** is to convert the byte code instructions into machine code so that the computer can execute those machine code instructions and display the final output.
- To carry out this conversion, PVM is equipped with an interpreter.
- The interpreter converts the byte code into machine code and sends that machine code to the computer processor for execution. Since interpreter is playing the main role, often the Python Virtual Machine is also called an interpreter.

Memory Management and Garbage Collection in Python

- **Memory allocation:**
    - Python , memory allocation and deallocation are done during runtime automatically. The programmer need not allocate memory while creating objects or deallocate memory when deleting the objects. Python’s PVM will take care of such issues.
    - Everything is considered as an object in Python. For example, Strings , Lists, Functions and modules are objects.
    - For every objects, memory should be allocated.
    - Memory manager inside the PVM allocates memory required for objects created in a Python program.
    - All objects are stored on a separate memory called ***heap***.
    - Heap is the memory which is allocated during runtime.

Memory Management and Garbage Collection in Python

- **Garbage Collection** :
  - 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 garbage 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 used. When an object is reference twice, its reference count will be 2. when an object has some count, it is being used in the program and garbage collector will not remove it from memory.
  - When an object is found with a 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.

Comparisons between c and python

C	Python
C is procedural-oriented programming language. It does not contain the features like classes , objects inheritance, polymorphism, etc.	Python is object oriented language, It contains features like classes , objects inheritance, polymorphism, etc.
C programs execute faster.	Python programs are slower compared to C. PyPy flavor or Python programs run a bit faster but still slower than c.
It is compulsory to declare the datatypes of variables , arrays etc .	Type declaration is not required in Python.
C language type discipline is static and weak	Python type discipline is dynamic and strong.
Pointer concept is available in c.	Python does not use Pointers.
C does not have exception handling facility and hence C programs are weak.	Python handles exception and hence Python Program are robust.
C has do….. While, while and for loops.	Python has while and for loops.
C has switch statement.	Python does not have switch statement.

C	Python
The variable in for loop does not increment automatically.	The variable in the for loop increments automatically.
The programmer should allocate and deallocate memory using malloc(), calloc(), realloc() or free() functions.	Memory allocation and deallocation is done automatically by PVM.
C does not contain a garbage collector.	Automatic garbage collector is available in Python.
The array index should be positive integer.	Array index can be positive or negative integer number. Negative index represents locations from the end of the array.
C supports single and multi-dimensional arrays.	Python supports only single dimensional arrays. To work with multi-dimensional arrays, we should use third party applications like numpy.
Indentation of statements is not necessary in C.	Indentation is required to represent a block of statements.
A semicolon is used to terminate the statements in C and comma is used to separate expressions.	New line indicates end of the statements and semicolon is used as an expression separator.
C supports in-line assignments.	Python does not support in-line assignments.

Comparisons between Java and Python

Java	Python
Java is object-oriented programming language. Functional Programming features are introduced into Java 8.0 through lambda expressions.	Python blends the functional programming with object-oriented programming features. Lambdas are already available in Python.
Java Programs are verbose . It means they contain more number of lines.	Python programs are concise and compact. A big program can be written using very less number of lines.
It is compulsory to declare the datatypes of variables, arrays,etc. in JAVA	Type declaration is not required in Python.
Java languagetype discipline is static and weak.	Python type discipline is dynamic and strong.
Java has do… while ,while , for and for each loops.	Python has while and for loop.
Java has switch statement.	Python does not have switch statement.
The variable in for loop does not increment automatically.	The variable in the for loop increments automatically.

Java	Python
Memory allocation and deallocation is done automatically by JVM(Java Virtual Machine).	Memory allocation and deallocation is done automatically by PVM(Pyhton Virtual Machine).
Java supports single and multi-dimensional arrays.	Python supports only single dimensional arrays. To work with multi-dimensional arrays, we should use third party applications like numpy.
The array index should be positive integer.	Array index can be positive or negative integer number. Negative index represents locations from the end of the array.
Indentation of statements is not necessary in Java.	Indentation is required to represent a block of statements.
A semicolon is used to terminate the statements in Java and comma is used to separate expressions.	New line indicates end of the statements and semicolon is used as an expression separator.

Data Types in Python

- A datatype represents the type of data stored into a variable or memory.
- The datatypes which are already available in python language are called ***Built-in datatype*.**
- The datatypes which can be created by the programming are called ***User-defined datatypes***.
- **Built-In Datatypes:**
  - **None types,**
  - **Numeric types,**
  - **Converting the datatypes explicitly**
  - **Sequence in Python : Str, bytes, bytearray, list, tuple ,range**
  - **Sets : set, frozenset, Mapping types**

1. None types:

- 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 a python program, maximum of only one ‘None’ object is provided. One of the uses of ‘None’ is that it is used inside a function as a default value of the arguments. When calling the function, if no value is passed, then the default value will be taken as ‘None’. If some value is passed to the function, then that value is used by the function.
- In Boolean expressions, ‘None’ datatype represents ‘False’.
  - **Ex-1:**

    ```
    def check_return():
    pass
    print(check_return())
    ```
  - **Output:**

    ```
    None
    ```
  - **EX-2:**

    ```
    x = None

    #display x:
    print(x)

    #display the data type of x:
    print(type(x))
    ```
  - **Output:**

    ```
    None
    <class 'NoneType'>
    ```

2. Numeric types:

- 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 or fraction part.
- For example, 200, -50, 0, 9888999, etc. are treated as integer numbers.
- Now, lets store an integer number -57 into a variable.

a = -57

- Here, ‘a’ is called int type variable.
- Python has no limit for the size of an int datatype. It can store very large numbers conveniently.

Ex:

```
x = 20
#display x:

print(x)
#display the data type of x:

print(type(x))
```

- **Output:**

```
20
<class 'int'>
```

- **Float datatype:**

- The float datatypes 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.

```
num=55.67998
```

- Here num is float variable .

```
x= 22.55e3
```

Complex datatype:

- A complex datatype is a number that is written **in the form of a+bj or a+bJ.**
- Here 'a' represents the real part of the number and 'b' represents the imaginary part of the number.
- For example: 3+5j, -1-5.5J, 0.2+10.5J are complex number.

  C1= -1-5.5J
- Here, complex number -1-5.5J is assigned to the 'c1'.
- Also, we can add two complex numbers and displays their sum.
- **Ex:**

```
#python program to add two complex numbers.
C1 = 2.5 +2.5J
C2 = 3.0 -0.5 J
C3 = C1 + C2
Print("sum=", C3)
```

- **Output:**

```
s
```

3. Converting the datatypes explicitly

- Depending on the type of data, Python internally assumes the datatype for the variable.
- But sometimes, the programmer wants to convert one datatype into another type on his own. This is called type ***conversion or coercion***.
- This is possible by mentioning the datatype with parentheses.
- For example, to convert a number into integer type, we can write **int(num)**
  - Int(x) is used to convert the number x into int type. See the example:

    X= 15.56

    int(x) #will display 15
  - Float(x) is used to convert x into float type. For example,

    Num=15

    float (num) #will display 15.0
  - Complex(x) is used to convert x into complex number with real part x and imaginary part. For example,

    n=10

    Complex(n) # will display(10+0j)

bool Datatype:

- The bool datatype in Python represents boolean values. There are only two boolean values True or False that 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 internally to either True or False.
- **For example,**

```
a = 10
b = 20
if(a < b) : print("Hello")  # displays Hello.
```

- In the previous code, the condition a<b which is written after if is evaluated to True and hence it will execute print("Hello").

```
a = 10 > 5  # here 'a' is treated as bool type variable
Print(a)  # displays True

a = 5 > 10
print(a)  # displays False
```

4. Sequence in Python : Str, bytes, bytearray, list, tuple ,range

- 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 characters.
- Strings are enclosed in **single quotes** or **double quotes**. Both are valid.

  Str= “welcome” #here str is name of string type variable

  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 a book on Python which discusses all the topics of Core Python in a very lucid manner.”””

  Str2 = ‘’’this is a book on Python which discusses all the topics of topics of Core Python in a very lucid manner.’’’

- The slice operator represents 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.
- See the examples below:

1. **s ='welcome to Core Python' #this is the original string**

   print(s)

   welcome to Core Python
2. **print(s[0]) #display 0th character from s**

   w
3. **print(s[3:7]) #display from 3rd to 6th characters**

   come
4. **print(s[11:]) #display from 11th characters onwards till end**

   Core Python
5. **print(s[-1]) #display first character from the end**

   n

- The repetition operator is denoted by '*' symbol and useful to repeat the string for several times. For example s * n repeats the string for n times. See the example:
  - 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 (inclusive).
- 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)   #convert the list into bytes array
print(x[0])    # display 0th element. i.e 10
```

- We **cannot modify or edit any element in the bytes type array**.
- **For example,**
- x[0]=55 gives an error. Here we are trying to replace 0th element (ie. 10) by 55 which is not allowed.

- **Program :** A Python program to create a byte type array, read and display the elements of the array.

```
#program to understand bytes type array
#create a list of byte numbers
elements= [10, 20, 0, 40, 15]

#convert the list into bytes type array
X= bytes (elements)

# retrieve elements from x using for loop and display
For i in x: print(i)
```

- **Output:** C:\> python bytes.py

```
10
20
0
40
15
```

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**. It means an element or all the elements of the bytearray type can be modified.
- To create a bytearray type array, we can use the function bytearray as:

```
elements = [30, 20, 0, 40, 15]  #this is list of byte numbers
X= bytearray(elements)  #convert the list into bytearray type array
print(x[0])   #display 0th element,  i.e. 30
```

- We can modify or edit the elements of the bytearray.
- For example, we can write:

```
x[0] =88  #replace 0th  element by 88
X[1] =99 #replace  1st  element by 99
```

- We will write program to create a bytearray type array and then modify the first two elements. Then we will display all the elements using a for loop.

- **Program :** A Python program to create a bytearray type array and retrieve elements.

```
#program to understand bytearray type array
#create a list of byte numbers
elements=[10, 20, 0, 40, 15]

#convert the list into bytearray type array
X= bytearray(elements)

#modify the first two elements of x
X[0] = 88
X[1] = 99

#retrieve elements from x using for loop and display
for i in x: print(i)
```

- **Output**: c:\>python bytearray.py

```
88
99
04
0
15
```

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 **a 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.
- Lists are represented using **square brackets [ ]** and the elements are written in [ ].
- separated by commas. For example,

  list= [10, 20, 15.5, 'Vijay', "Mary"]
- will create a list with different types of elements.
- The **slicing operation** like [0: 3] represents elements from 0th to 2nd positions, i.e. 10, 20, 15.5.

```
>>> list = [10, -20, 15.5, ‘Vijay’, "Mary"]
>>> print(list)
 [10, -20, 15.5, ‘Vijay', "Mary”]

>>> print (list[0])
10
>>> print (list[1:3])
[-20, 15.5]
>>> print (list[-2])
Vijay
>>> print (list * 2)
[10, -20, 15.5, ‘Vijay’, "Mary“, 10, -20, 15.5, ‘Vijay’, "Mary“]
```

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 a **read-only list**. Let's create a tuple as:

  tpl = (10, -20, 15.5, 'Vijay', "Mary")
- The individual elements of the tuple can be referenced using square braces as tpl[0], tpl[1], tpl[2], ... Now, if we try to modify the 0th element as:

  tpl [0] = 99

  This will result in error.

- The slicing operations which can be done on lists are also valid in tuples.
- See the general operations on the tuple :

```
>>> tpl = (10, 20, 15.5, Vijay', "Mary")
 >>> print(tpl)
(10, -20, 15.5, 'Vijay', 'Mary')

>>> print(tpl[0])
10

>>> print (tpl [1:3])
(-20, 15.5)

>>> print (tp1*2)
 (10, -20, 15.5, 'Vijay', 'Mary', 10, 20, 15.5. 'Vijay', 'Mary')

>>> print (tpl [-2])
Vijay

>>> tpl [0] = 99
        Traceback (most recent call last):
        File "cpyshell#6>", line 1, in <module>
        tpl. [0] = 99
TypeError: 'tuple' object does not support item assignment
```

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 in created with the numbers starting from 0 to 9. We can display these numbers using a for loop as:

```
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. It means the numbers in the range will increase by 2 every time. So, the for loop

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

- will display numbers: 30, 32, 34, 36, 38.
- Let's create a list with a range of numbers from 0 to 9 as:

```
lst = list(range(10))
print(lst)   # will display: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

5. Sets: set, frozenset, Mapping types

- 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)   #may display {50, 10, 20, 30}
```

- Please observe that the set 's' is not maintaining the order of the elements. We entered the elements in the order 10, 20, 30, 20 and 50. But it is showing another order. Also, we repeated the element 20 in the set, but it has stored only one 20. We can use the set function to create a set as:

```
ch= set("Hello")
print(ch)   #may display{‘H’,’e’,’l’,’o’}
```

- Here, a set 'ch' is created with the characters H,e,l,o. Since a set does not store duplicate elements, it will not store the second l.

- We can convert a list into a set using the set() function as:

  lst = [1,2,5,4,3]

  s=set(lst)

  print(s) #may display {1, 2, 3, 4, 5}

- Since sets are unordered, we cannot retrieve the elements using indexing or slicing operations.
- For example, the following statements will give **error messages**:

  print(s[0]) #indexing, display 0th element

  print(s[0:2]) # slicing, display from 0 to 1st elements

- The update() method is used to add elements to a set as:

  S.update([50,60])

  Print(S) # may display {1,2,3, 4, 5, 50, 60}

- On the other hand, the remove() method is used to remove any particular element from a set as:

  s.remove(50)

  print(s) # may display {1, 2, 3, 4, 5, 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**.
- We can create a frozenset by passing a set to frozenset() function as:

```
s = {50, 60, 70, 80, 90}
Print(s)    #may display{80, 90, 50, 60, 70}


fs = frozenset(s)  # create frozenset fs
print(fs) # may display frozenset ({80, 90, 50, 60, 70})
```

- Another way of creating a frozenset is by passing a string (a group of characters) to the frozenset() function as:

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

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

Mapping Types

- A map represents **a group of elements in the form of key value pairs**
- 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.
- All the elements should be enclosed inside curly brackets {}.
- We can create a dictionary by typing the roll numbers and names of students. Here, roll numbers are keys and names will become values. We write these key value pairs inside curly braces as:

```
d={10: ‘Kamal', 11: 'Pranav’, 12: 'Hasini', 13: 'Anup', 14: ‘Reethu’}
```

- Here, d is the name of the dictionary. 10 is the key and its associated value is 'Kamal'. The next key is 11 and its value is 'Pranav'. Similarly 12 is the key and 'Hasini' is it value. 13 is the key and 'Anup' is the value and 14 is the key and 'Reethu' is the value.
- We can create an empty dictionary without any elements as:

```
d={}
```

- Later, we can store the key and values into d as:

  d[10]='kamal'

  d[11] ='Pranav'
- In the preceding statements, 10 represents the key and Kamal' is its value. Similarly, 11 represents the key and its value is 'Pranav'. Now, if we write

  print(d)
- It will display the dictionary as:

  {10: 'Kamal', 11: 'Pranav'}

- We can perform various operations on dictionaries. To retrieve value upon giving the key. we can simply mention d[key]. To retrieve only keys from the dictionary, we can use the method keys() and to get only values, we can use the method values().
- We can update the value of a key, as: d[key] = newvalue. We can delete a key and corresponding value, using del module.

- For example del d[11] will delete a key with 11 and its value. demonstrates these operations:

```
>>> d = {10: Kamal', 11: 'Pranav', 12: 'Hasini', 13: Anup', 14: 'Reethu‘}
>>> print(d)
{10: 'Kamal', 11: 'Pranav', 12: 'Hasini', 13: 'Anup', 14: 'Reethu‘}

>>> print(d[11])
Pranav

>>> print(d.keys())
dict_keys([10, 11, 12, 13, 14])

>>> print(d.values())
dict_values(['Kamal', 'Pranav', 'Hasini', 'Anup', 'Reethu'])

>>> d[10] = 'Hareesh'
>>> print (d)
(10: 'Hareesh', 11: 'Pranav', 12: 'Hasini', 13: 'Anup', 14: 'Reethu'}

>>> del d[11]
>>> print (d)
(10: 'Hareesh', 12: 'Hasini', 13: 'Anup', 14: 'Reethu')
```

Determining the Datatype of a Variable

- To know the datatype of a variable or object, we can use the type() function.
- For example, type(a) **displays the datatype** of the variable 'a'.

```
a = 15
print(type(a))
<class 'int'>
```

- Since we are storing an integer number 15 into variable 'a', the type of 'a' is assumed by the Python interpreter as 'int'. Observe the last line. It shows class 'int'.
- It means the variable 'a' is an object of the class 'int'. It means 'int' is treated as a class.
- Every datatype is treated as an object internally by Python.
- In fact, every datatype, function, method, class, module, lists, sets, etc. are all objects in Python.

```
>>> a = 15
>>> print (type (a))
<class 'int'>

>>>> a = 15.5
>>>>> print (type (a))
<class 'float'>

>>> s = 'Hello'
>>> print (type(s))
<class 'str'>

>>> print (type("Hai"))
<class 'str'>

>>> lst = [1, 2, 3, 4]
>>> print(type( print (type (type(lst))
<class 'list'>

>>> t = (1, 2, 3, 4)
>>> print (type(t))
<class 'tuple'>
```

Python Fundamentals and Control Flow Statements

- **Tokens:**
- In Python, tokens are the smallest individual units of a program that are meaningful to the interpreter.
- These tokens are categorized into several types:
  - **Keywords**
  - **Identifiers**
  - **Literals**
  - **Operators**
  - **Punctuators (Delimiters)**

Keywords:

- These are reserved words in Python with **predefined meanings and functionalities.**
- 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.
- The following are the reserved words available in Python:

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		

Identifiers:

- An identifier **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 character. 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 and hence represent different variables. shows examples of a variable, an operator and a literal:
- This symbol stores right side literal into left side variable. Thus it is performing an operation. Hence it is called **'operator'**.

  salary = 15000.75

- This is a **variable**. Its name is 'salary'. Hence this name 'salary' is 'identifier'
- This is the float number stored into variable. Hence it is called 'floating point **literal**'.

Literals:

- A literal **is a constant value** that is stored into a variable in a program.
- Observe the following statement:

  a = 15
- Here, 'a' is the variable into which the constant value '15' is stored.
- Hence, the value 15 is called 'literal'. Since 15 indicates integer value, it is called ‘**integer literal**'.
- The following are different types of literals in Python:
  - **Numeric literals**
  - **Boolean literals**
  - **String literals**

- **Numeric Literals**
- These literals represent numbers.
- Please observe the different types of numeric literals available in Python, as shown in Table:
- Table : Numeric Literals

Examples	Literal name
450, -15	Integer literal
3.14286, -10.6, 1.25E4	Float literal
0x5A1C	Hexadecimal literal
0557	Octal literal
0B110101	Binary literal
18+3J	Complex literal

- **Boolean Literals**

  Boolean literals are the **True or False values stored into a bool type** variable.

- **String Literals**
- **A group of characters is called a string literal**.
- These string literals are enclosed in single quotes (‘) or double quotes (") or triple quotes (‘’’ or """).
- In Python, there is no difference between single quoted strings and double quoted strings.
- Single or double quoted strings should end in the same line as:

  ```
  S1=‘This is first Indian book'
  s2= "Core Python"
  ```

- We can use escape character like \n inside a string literal.

  Example:

  ```
  str= “this is \n python “
  Print(str)
  ```

  Output:

  ```
  This is
  python
  ```

Punctuators (Delimiters)

- These are the symbols that used in Python to organize the structures, statements, and expressions. Some of the Punctuators are: [ ] { } ( ) @ -= += *= //= **== = , etc.

Operators:

- These are symbols that perform operations on operands (variables or literals). Examples include:
- **Arithmetic Operators:** +, -, *, /, %, **, //
- **Comparison Operators:** ==, !=, <, >, <=, >=
- **Logical Operators:** and, or, not
- **Assignment Operators:** =, +=, -=, *=

Arithmetic Operators

These operators are used to perform mathematical operations.

Operator	Name	Example
+	Addition	x + y
-	Subtraction	x - y
*	Multiplication	x * y
/	Division	x / y
%	Modulus	x % y
**	Exponentiation	x ** y
//	Floor division	x // y

Comparison Operators

- These operators compare two values and return a Boolean result.

Operator	Name	Example
==	Equal	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

Logical Operators

- These operators are used to combine conditional statements.

Operator	Description	Example
and	Returns True if both statements are true	x < 5 and x < 10
or	Returns True if one of the statements is true	x < 5 or x < 4
not	Reverse the result, returns False if the result is true	not(x < 5 and x < 10)

Bitwise Operators

- These operators perform operations on binary digits.

Operator	Name	Description
&	AND	Sets each bit to 1 if both bits are 1
|	OR	Sets each bit to 1 if one of two bits is 1
^	XOR	Sets each bit to 1 if only one of two bits is 1
~	NOT	Inverts all the bits
<<	Zero fill left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off
>>	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off

Assignment Operators

- These operators are used to assign values to variables.

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
//=	x //= 3	x = x // 3
**=	x **= 3	x = x ** 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

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 declared using these operators.
- There are two membership operators as shown here:
  - in
  - not in

operator	Description	Example

in	Returns True if the element is found in the sequence, False otherwise.	x=[“apple”,”orange”]
		print(“apple” in x)
		o/p: true

not in	Returns True if the element is not found in the
	sequence, False otherwise.
		x=[“apple”,”orange”]
		print(“pineapple” not in x)
		o/p: true

Identity operator

- These operator **compare the memory locations of two objects**.
- It is possible to know whether the two objects are same or not.

Operator	Description	Example
is	Returns true if both variables are the same object	x is y
is not	Returns true if both variables are not the same object	x is not y

- Example: Is operator

```
x = ["apple", "banana"]
y = ["apple", "banana"]

print(x is y)
```

# returns False because x is not the same object as y, even if they have the same content

```
print(x == y)
```

# to demonstrate the difference betweeen "is" and "==": this comparison returns True because x is equal to y

- **O/p: false**

  true

- Example: is not operator

```
x = ["apple", "banana"]
y = ["apple", "banana"]

print(x is not y)
```

# returns True because x is not the same object as y, even if they have the same content

```
print(x != y)
```

# to demonstrate the difference betweeen "is not" and "!=": this comparison returns False because x is equal to y

o/p: true

false

Output Statements

- Python uses the **print()** function **to display output.**
- **Example:**

```
# Printing a string literal
print("This is a simple message.")

# Printing variables
x = 10
y = 20
print("The value of x is:", x)
print("The sum of x and y is:", x + y)

# Using f-strings for formatted output (Python 3.6+)
name = "Alice"
age = 30
print(f"My name is {name} and I am {age} years old.")

# Customizing print behavior with 'sep' and 'end'
print("Hello", "World", sep="-") # Separates arguments with a hyphen
```

Input Statements

- To accept input from keyboard, Python provides the **input() function**. This function takes a value from the keyboard and returns it as a string.
- **Example:**

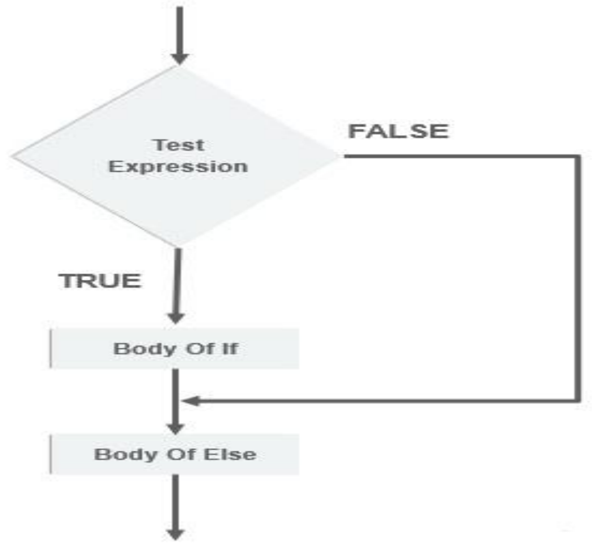
```
# Basic input
name = input("Enter your name: ")
print("Hello, " + name + "!")

# Input and type conversion
age_str = input("Enter your age: ")
age_int = int(age_str) # Convert string input to an integer
print("You are", age_int, "years old.")


# input integer and float
age = int(input("enter your age"))
Percentage = float(input(" enter your percentage"))
```

Control Statements

- Control statements are statements which control or change the flow of execution.
- The following are the control statements avaliable in python:
  - 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

if statement:

- This statement is used to execute one or more statement depending on whether a condition is True or not.
- Condition is tested . If condition is True, then the statements given after colon(:) are executed.
- If condition is False. Then the statements mentioned after colon are not executed.



- **Syntax:**

```
If condition:
      statements
```

- **Example:**

```
num=1
If num==1:
      print("One")
```

- **O/p:**

```
      One
```

- **Example:**

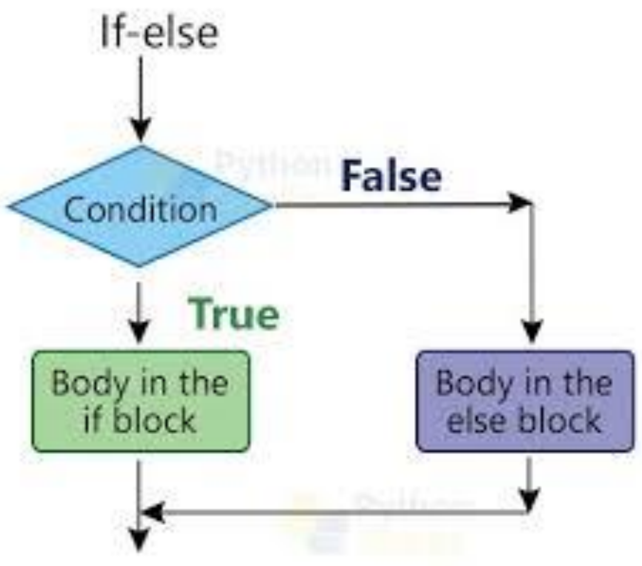
```
a=5
if a>2:
      print(a,"is greater")
```

- **O/p:**

```
      5 is greater
```

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.



- **Syntax:**

```
if condition:
   # Code to execute if the condition is True
   statement_1
   statement_2
else:
   # Code to execute if the condition is False
   statement_A
   statement_B
```

Example:

```
x = 10
If x % 2 == 0:
      print(x, " is even number")
else:
      print(x, "is odd number")
```

Output:

```
10 is even number
```

Example:

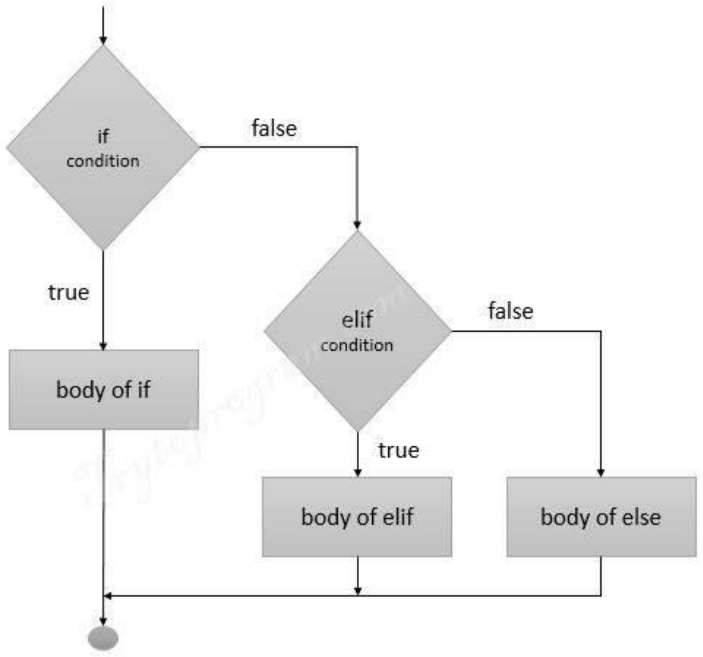
```
x = -5
If x > 0 :
      print(x, " is positive number")
else:
      print(x, "is negative number")
```

Output:

```
5 is negative snumber
```

if…elif…else statement

- The programmer has to test multiple conditions and execute statements depending on those conditions.



Syntax:

```
if condition1:
    # Code to execute if condition1 is True
  Statement 1
elif condition2:
    # Code to execute if condition1 is False and condition2 is True
    Statement 2
elif condition3:
    # Code to execute if condition1 and condition2 are False, and condition3 is True
     Statement 3
else:
   # Code to execute if all preceding conditions are False
   Statement 4
```

Example:

```
num= -5
If num == 0:
        print(num, “is zero”)
elif num > 0:
        print(num, “is positive”)
else:
        print(num, “is negative”)
```

O/p:

```
-5 is negative
```

Example:

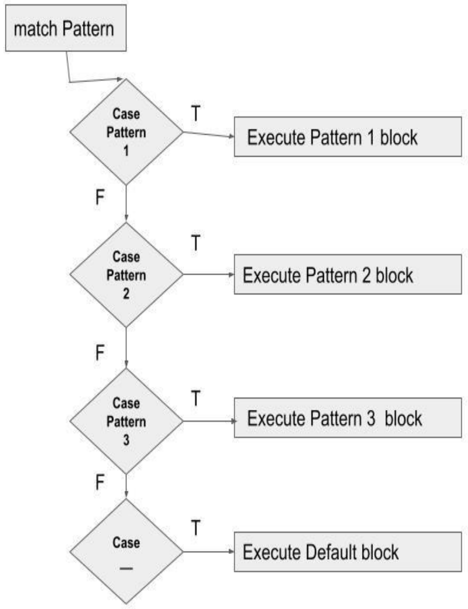
```
a,b,c = 12,45,3
If a> b and a > c:
        print(a, “is greater”)
elif b> a and b>c:
        print(b, “is greater”)
else:
        print(c , “is greater”)
```

O/p:

```
45 is greater
```

Match case statement

- The match statement is used to perform different actions based on different conditions.
- Instead of writing **many** if..else statements, you can use the match statement.
- The match statement selects one of many code blocks to be executed.
- This is how it works:
- The match expression is evaluated once.
- The value of the expression is compared with the values of each case.
- If there is a match, the associated block of code is executed.



Syntax:

```
match expression:
  case x:
    code block
  case y:
    code block
  case z:
    code block
```

Example:

```
day = 4
match day:
  case 1:
    print("Monday")
  case 2:
    print("Tuesday")
  case 3:
    print("Wednesday")
  case 4:
    print("Thursday")
  case 5:
    print("Friday")
  case 6:
    print("Saturday")
  case 7:
    print("Sunday")
```

Output:

```
Thursday
```

- **Wildcard Pattern (_):** Acts as a default case, matching any value if no other pattern matches. It should be placed as the last case.
- **OR Patterns (|):** Combine multiple patterns with the logical OR operator.

Example:

```
day = 6
match day:
case 1 | 2 | 3 | 4 | 5:
    print("Weekday")
case 6 | 7:
     print("Weekend")
```

Output:

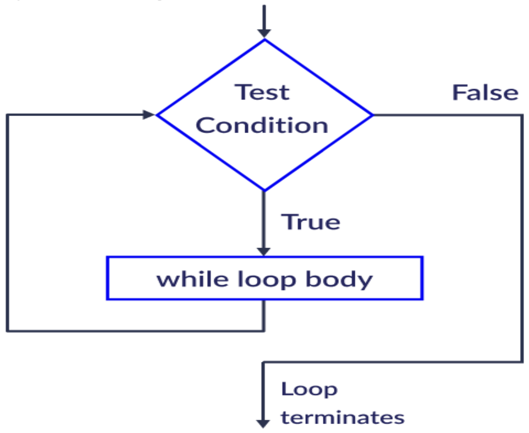
```
Weekend
```

loops:

- In Python, loops are used to repeatedly execute a block of code.
- **A loop allows to execute a statement or a group of statements multiple times as long as the condition is true.**
- while ,
- for ,
- infinite,
- nested,
- else suite
- Pass statement,
- Assert Statement,
- Return Statement

While loop:

- The while loop is useful to execute a group of statements several times repeatedly depending on whether a condition is True or False.



Syntax:

```
while condition:
        statements
```

Example:

```
x=1
while x<=5:
        print(x)
        x+=1
print("end")
```

OUTPUT:

```
1
2
3
4
5
end
```

Example:

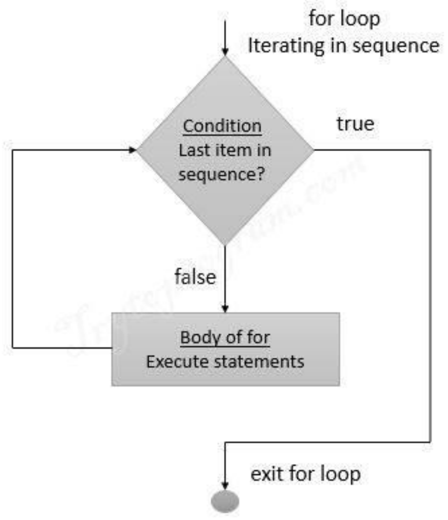
```
x=1
while x<=5:
        print(x)
        x+=1
        print("end")
```

OUTPUT:

```
1
end
2
end
3
end
4
end
5
end
```

For loop

- The for loop can be used to execute a group of statements repeatedly depending upon the number of elements in the sequence.
- The for loop can work with sequence like string, list, tuple, range , etc.



syntax:

```
for var in sequence:
        statements
```

Example:

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

Output:

```
1
3
5
7
9
```

Example:

```
lst =[10,20.5,’A’,’America’]
for i in lst:
    print(i)
```

Output:

```
10
20.5
A
America
```

Infinite loops:

- An infinite loop is a loop that executes forever.
- Infinite loops are drawback in a program because when the user is caught in a infinite loop, they cannot understand how to come out of the loop. So, it is always recommended to avoid infinite loops in any program.

Example:

```
i=1
while i<=10
    print(i)
```

i.e. the initial 'i ' value is displayed first, but it is never incremented to reach 10. hence loop will always display 1 and never terminates.

Such a loop is called 'Infinite loop'.

Nested loops

- In python , It is possible to write one loop inside another loop.
- We can write a for loop inside a while loop or a for loop inside another for loop. Such loops are called ‘**Nested loops**’.

```
rows = 5
for i in range(1, rows + 1):
    for j in range(1, i + 1):
        print('*',end=' ')
    print(‘  ‘)
```

Output:

```
*
* *
* * *
* * * *
* * * * *
```

```
rows = 5
for i in range(1, rows + 1):
    for j in range(1, i + 1):
        print(j,end=' ')
    print(‘  ‘)
```

Output:

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

Else suite loop

- In python, it is possible to use ‘else’ statement along with for loop and while loop.

For with else

```
for var in sequence:
    statements
else:
    statements
```

Example:

```
for i in range(5):
    print(“yes”)
else:
    print(“no”)
```

Output:

```
yes
yes
yes
yes
yes
no
```

while with else

```
while condition:
    statements
else:
    statements
```

Example:

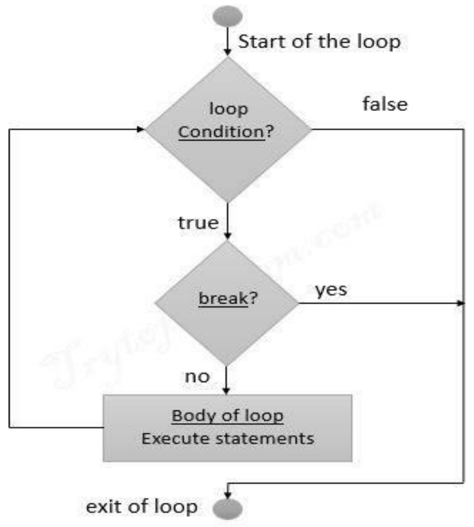
```
count = 0
while count < 3:
    print(count)
    count += 1
else:
    print("While loop finished normally.")
```

Output:

```
0
1
2
While loop finished normally.
```

Break statement:

- 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 access the next statement** int the program.



Syntax:

```
break
```

Example:

```
for num in range(10):
    if num == 5:
        print("Loop stopped at:", num)
        break
    print("Current number:", num)

print("Loop finished.")
```

Output:

```
Current number: 0
Current number: 1
Current number: 2
Current number: 3
Current number: 4
Loop stopped at: 5
Loop finished.
```

Example:

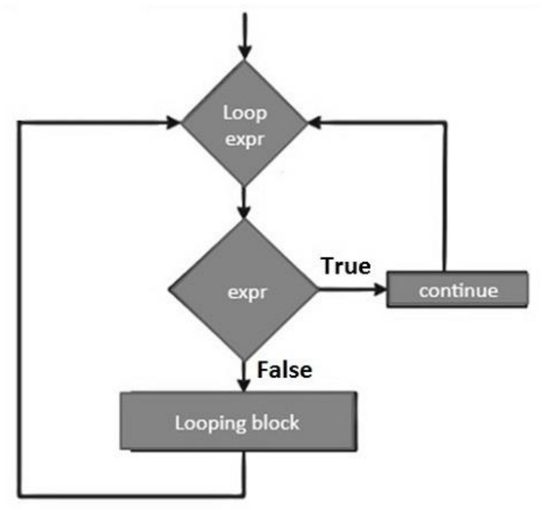
```
i=1
while i<=10:
        print(i)
        if i==5:
                break
        i+=1
```

Output:

```
1 2 3 4 5
```

continue statement

- The continue statement is used in a loop **to go back to the beginning of the loop.**
- The continue statement in Python is a control flow statement used within loops (both for and while loops). Its purpose is to skip the remaining code within the current iteration of the loop and immediately jump to the beginning of the next iteration.



Syntax:

```
while True:
   ...
   if x == 10:
      continue
   print(x)
```

Example:

```
     i=0
     While i< 5:
          i=i+1
          If i==3:
               Continue
          Print(i)
```

Output:

```
1 2 4 5
```

pass statement

- 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.
- 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.

Example:

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

Output:

```
-4
-5
-6
-7
```

#**Example** of using pass in an empty function

```
def fun():
  pass
# Call the function
fun()
```

assert statement

- The assert statement is useful **to check if a particular condition is fulfilled or not**.
- The assert statement in Python is used for debugging purposes to check if a given condition is true. If the condition evaluates to False, an AssertionError is raised.

syntax:

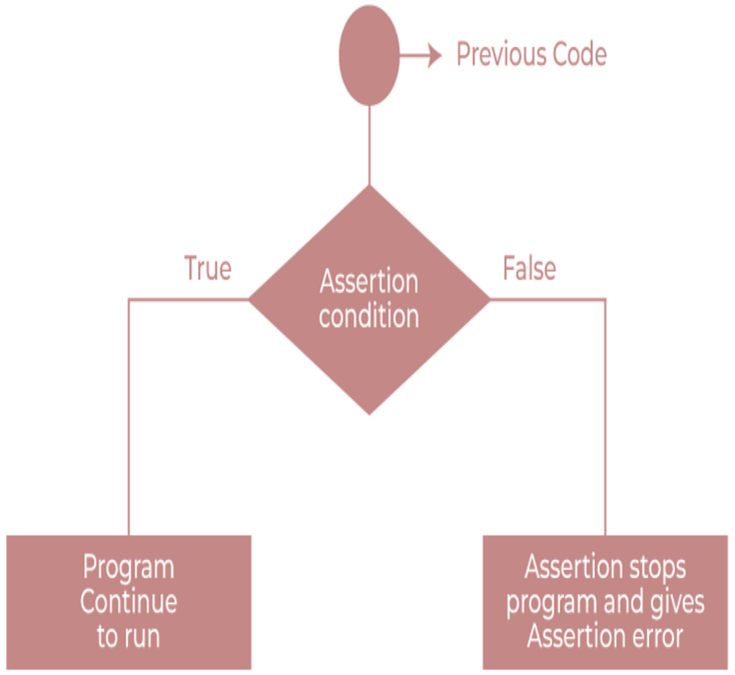
```
assert condition, message
```

example:

```
assert x>0, “wrong input entered”
```

Example:

```
a = 4
b = 0

# using assert to check for 0
print("The value of a / b is : ")
assert b != 0, "Zero Division Error"
print(a / b)
```



Example:

```
x= int(input(“enter greater than 0:”))
assert x > 0 , “wrong input entered”
print(“number is:”, x)
```

output:

```
enter greater than 0: 5
number is: 5

enter greater than 0: -5
wrong input entered
```

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 result.
- A function starts with the keyword def that represents the definition of the function.
- After ‘def’ , the function should be written.
- For example: def sum(a,b):

```
        function body
    Sum(5,10)
```

Syntax:

return expression

- In some cases, it is highly useful to write the function as if it is returning the result. This is done using ‘return’ statement. Return statement is used inside a function to return some value to the calling place.

Example:

```
def sum(a,b):
        return a + b
res= sum(5,10)
print(“The result is’,’res)
```

Output:

```
The result is 15
```