Unit-I

Introduction to Python: History, Features, Applications, First Python Program, Variables, Data Types, Numbers, Operators, Input and Output statements.

Control Statements: Conditional Statements, A Word on Indentation, Looping Statements, the else Suite, break, continue, pass, assert, return.

Introduction to Python

Python is the world's most popular and fastest-growing computer programming language. It is a multi-purpose and high-level programming language.

Python was invented by Guido Van Rossum in the year 1989, but it was introduced into the market on 20th February 1991.

The Python programming language has been used by many people like

- Software Engineers
- Data Analysts
- Network Engineers
- Mathematicians
- Accountants
- Scientists and many more.

Using Python, one can solve complex problems in less time with fewer lines of code.

History

- Python laid its foundation in the late 1980s.
- The implementation of Python was started in December 1989 by Guido Van Rossum at CWI in Netherland.
- In February 1991, Guido 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 such as list comprehensions, garbage collection systems.
- On December 3, 2008, Python 3.0 (also called "Py3K") was released. It was designed to rectify the 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 the Amoeba Operating System.

The following programming languages influence Python:

- ABC language
- Modula-3

Why the Name Python?

There is a fact behind choosing the name Python. Guido van Rossum was reading the script of a popular BBC comedy series "Monty Python's Flying Circus". It was late on-air 1970s.

Van Rossum wanted to select a name which unique, sort, and little-bit mysterious. So he decided to select naming Python after the "Monty Python's Flying Circus" for their newly created programming language.

The comedy series was creative and well random. It talks about everything. Thus it is slow and unpredictable, which made it very interesting.

Python is also versatile and widely used in every technical field, such as

- Machine Learning
- Artificial Intelligence
- Web Development
- Mobile Application
- Desktop Application
- Scientific Calculation, etc.

Python Version List

Python programming language is being updated regularly with new features and supports. There are lots of update in Python versions, started from 1994 to current release.

A list of Python versions with its released date is given below.

```
     Version              Release 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          Feb. 13, 2009
    Python 3.1          June 26, 2009
    Python 3.2          Feb. 20, 2011
    Python 3.3          Sept. 29, 2012
    Python 3.4          March 17, 2014
    Python 3.5          Sept. 13, 2015
    Python 3.6          Dec. 23, 2016
    Python 3.7          June 27, 2018
    Python 3.7.3        March 25, 2019
```

Features of Python

Python provides many useful features which make it popular and valuable from the other programming languages. It supports object-oriented programming, procedural programming approaches and provides dynamic memory allocation. We have listed below a few essential features.

1. Easy to Learn and Use

Python is easy to learn as compared to other programming languages. Its syntax is straightforward and much the same as the English language.

There is no use of the semicolon or curly-bracket, the indentation defines the code block. It is the recommended programming language for beginners.

2. Expressive Language

Python can perform complex tasks using a few lines of code.

A simple example, the hello world program you simply type **print("Hello World")**.

It will take only one line to execute, while Java or C takes multiple lines.

3. Interpreted Language

Python is an interpreted language; it means the Python program is executed one line at a time. The advantage of being interpreted language, it makes debugging easy and portable

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. It enables programmers to develop the software for several competing platforms by writing a program only once.

5. Free and Open Source

Python is freely available for everyone. It is freely available on its official website www.python.org (http://www.python.org).

It has a large community across the world that is dedicatedly working towards make new python modules and functions. Anyone can contribute to the Python community.

The open-source means, "Anyone can download its source code without paying any penny."

6. Object-Oriented Language

Python supports object-oriented language and concepts of classes and objects come into existence. It supports inheritance, polymorphism, and encapsulation, etc.

The object-oriented procedure helps to programmer to write reusable code and develop applications in less code.

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.

It converts the program into byte code, and any platform can use that byte code.

8. Large Standard Library

It provides a vast range of libraries for the various fields such as machine learning, web developer, and also for the scripting.

There are various machine learning libraries, such as Tensor flow, Pandas, Numpy, Keras, and Pytorch, etc. Django, flask, pyramids are the popular framework for Python web development.

9. GUI Programming Support

Graphical User Interface is used for the developing Desktop application. PyQT5, Tkinter, Kivy are the libraries which are used for developing the web application.

10. Integrated

It can be easily integrated with languages like C, C++, and JAVA, etc. Python runs code line by line like C,C++ Java. It makes easy to debug the code.

11. Embeddable

The code of the other programming language can use in the Python source code. We can use Python source code in another programming language as well. It can embed other language into our code.

12. Dynamic Memory Allocation

In Python, we don't need to specify the data-type of the variable.

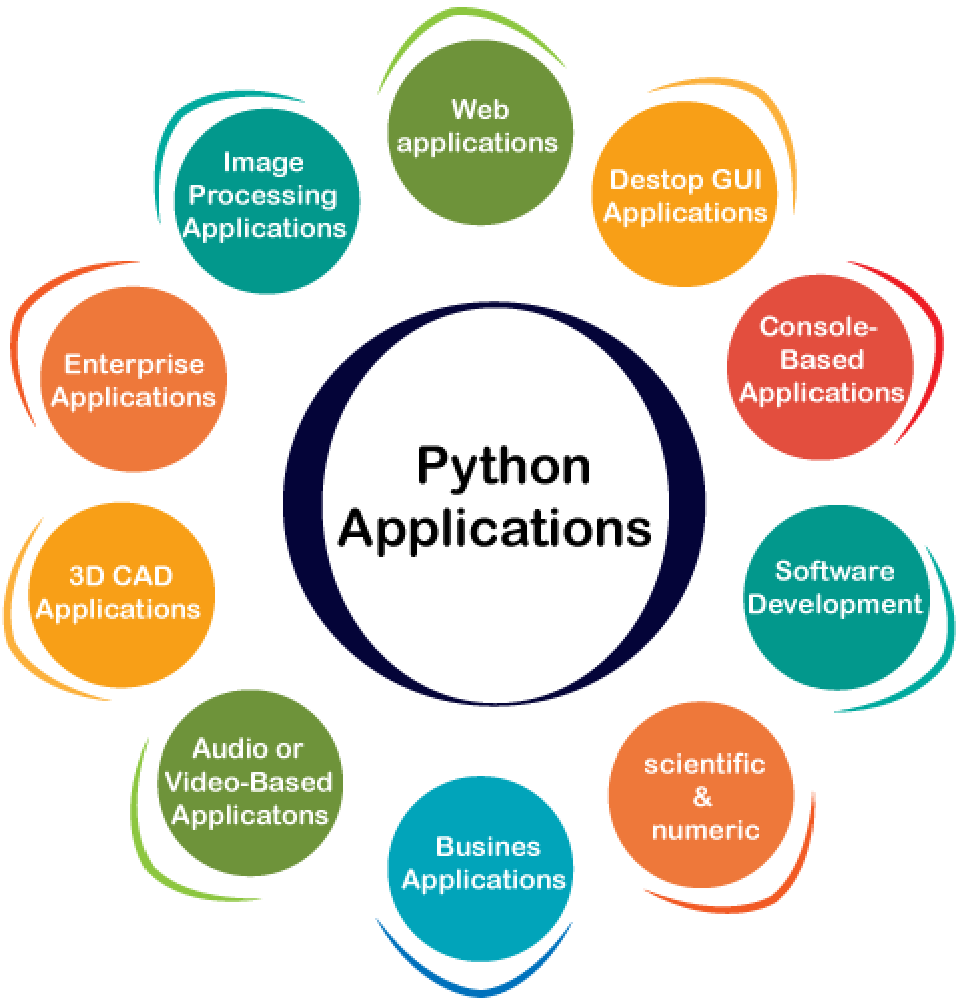
When we assign some value to the variable, it automatically allocates the memory to the variable at run time.

Suppose we are assigned integer value 15 to x, then we don't need to write **int x = 15**. Just write **x = 15**.

Applications of Python

Python is known for its general-purpose nature that makes it applicable in almost every domain of software development. Python makes its presence in every emerging field. It is the fastest-growing programming language and can develop any application.

Here, we are specifying application 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. One of Python web-framework named Django is used on Instagram. Python provides many useful frameworks, and these are given below:

- Django and Pyramid framework(Use for heavy applications)
- Flask and Bottle (Micro-framework)
- Plone and Django CMS (Advance Content management)

2. Desktop GUI Applications

The GUI stands for the Graphical User Interface, which provides a smooth interaction to any application. Python provides a Tk GUI library to develop a user interface. Some popular GUI libraries are given below.

- Tkinter or Tk
- wxWidgetM
- Kivy (used for writing multitouch applications)
- PyQt or Pyside

3. Console-based Application

Console-based applications run from the command-line or shell. These applications are computer program which are used commands to execute. This kind of application was more popular in the old generation of computers. Python can develop this kind of application very effectively.

It is famous for having **REPL**, which means the Read-Eval-Print Loop that makes it the most suitable language for the command-line applications.

Python provides many free library or module which helps to build the command-line apps. The necessary **IO libraries** are used to read and write. It helps to parse argument and create console help text out-of-the-box. There are also advance libraries that can develop independent console apps.

4. Software Development

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

- **SCons** is used to build control.
- **Buildbot** and **Apache** Gumps are used for automated continuous compilation and testing.
- **Round** or **Trac** for bug tracking and project management.

5. Scientific and Numeric

This is the era of Artificial intelligence where the machine can perform the task the same as the human. Python language is the most suitable language for Artificial intelligence or machine learning. It consists of many scientific and mathematical libraries, which makes easy to solve complex calculations.

Implementing machine learning algorithms require complex mathematical calculation. Python has many libraries for scientific and numeric such as Numpy, Pandas, Scipy, Scikit-learn, etc. If you have some basic knowledge of Python, you need to import libraries on the top of the code. Few popular frameworks of machine libraries are given below.

- SciPy
- Scikit-learn

- NumPy
- Pandas
- Matplotlib

6. Business Applications

Business Applications differ from standard applications. E-commerce and ERP are an example of a business application. This kind of application requires extensively, scalability and readability, and Python provides all these features.

Oddo is an example of the all-in-one Python-based application which offers a range of business applications. Python provides a **Tryton** platform which is used to develop the business application.

7. Audio or Video-based Applications

Python is flexible to perform multiple tasks and can be used to create multimedia applications. Some multimedia applications which are made by using Python are **TimPlayer**, **cplay**, etc. The few multimedia libraries are given below.

- Gstreamer
- Pyglet
- QT Phonon

8. 3D CAD Applications

The CAD (Computer-aided design) is used to design engineering related architecture. It is used to develop the 3D representation of a part of a system. Python can create a 3D CAD application by using the following functionalities.

- Fandango (Popular)
- CAMVOX
- HeeksCNC
- AnyCAD
- RCAM

9. Enterprise Applications

Python can be used to create applications that can be used within an Enterprise or an Organization. Some real-time applications are OpenERP, Tryton, Picalo, etc.

10. Image Processing Application

Python contains many libraries that are used to work with the image. The image can be manipulated according to our requirements. Some libraries of image processing are given below.

- OpenCV
- Pillow
- SimpleITK

Python Program Examples

Single-Line Statement

In [ ]:

```
print("Hello World")
```

Multi-line Statements

In [ ]:

```
college = "CMR Technical Campus"
branch = "Computer Science and Engineering"
code = "7R"
print("College name is: ", college, )
print("College code is: ", code, )
print("Department name is: ", branch)
```

Python Variables

- Variable is a name that is used to refer to **memory location**. **Python variable is also known as an identifier and used to hold value**.
- In Python, we don't need to specify the **type of variable** because Python is a infer language and smart enough to get variable type.
- Variable names can be a group of both the **letters** and **digits**, but they have to **begin with a letter or an underscore**.
- **It is recommended to use lowercase letters for the variable name**.

```
Example: Rahul and rahul both are two different variables.
```

Identifier Naming

Variables are the example of **identifiers**. An Identifier is used to identify the literals used in the program. The rules to name an identifier are given below.

- The **first character** of the variable must be an **alphabet or underscore (_)**.

- All the characters except the first character may be an **alphabet of lower-case(a-z)**, **upper-case (A-Z)**, **underscore**, or **digit (0-9)**.
- Identifier name must not contain any **white-space**, or **special character** (!, @, #, %, ^, *, &).
- Identifier name must not be similar to any **keyword** defined in the language.
- Identifier names are **case sensitive**.

```
for example, myname, and MyName is not the same.
```

- Examples of valid identifiers: a123, _n, n_9, etc.
- Examples of invalid identifiers: 1a, n%4, n 9, etc.

Declaring Variable and Assigning Values

Python does not bind us to declare a variable before using it in the application. It allows us to create a variable at the required time.

We don't need to declare explicitly variable in Python. When we assign any value to the variable, that variable is declared automatically.

The equal (=) operator is used to assign value to a variable.

Object References

It is necessary to understand how the Python interpreter works when we declare a variable. The process of treating variables is somewhat different from many other programming languages.

Python is the highly object-oriented programming language; that's why every data item belongs to a specific type of class. Consider the following example.

In [ ]:

```
print("John")
```

The Python object creates an integer object and displays it to the console. In the above print statement, we have created a string object. Let's check the type of it using the Python built-in **type()** function.

In [ ]:

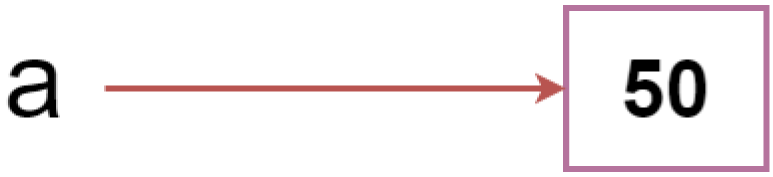
```
type("John")
```

In Python, variables are a symbolic name that is a reference or pointer to an object. The variables are used to denote objects by that name.

Let's understand the following example

In [ ]:

```
a = 50
print(a)
```

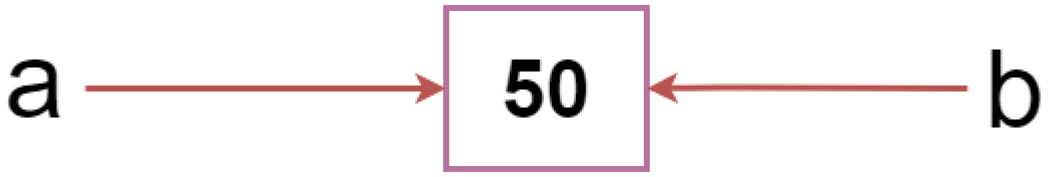


In the above image, the variable a refers to an integer object.

Suppose we assign the integer value 50 to a new variable b.

In [ ]:

```
a = 50
b = a
print(a)
print(b)
```

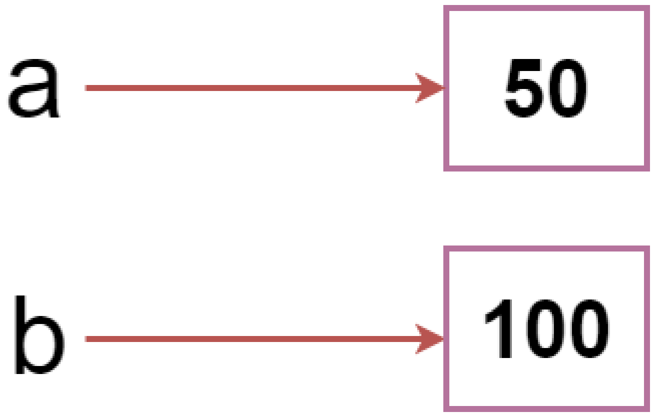


The variable b refers to the same object that a points to because Python does not create another object.

Let's assign the new value to b. Now both variables will refer to the different objects.

In [ ]:

```
a = 50
b =100
print(a)
print(b)
```



Python manages memory efficiently if we assign the same variable to two different values.

Object Identity

In Python, every created object identifies uniquely in Python. Python provides the guaranteed that no two objects will have the same identifier.

The built-in **id()** function, is used to identify the object identifier. Consider the following example.

In [ ]:

```
a = 50
b = a
print(id(a))
print(id(b))
# Reassigned variable a
a = 500
print(id(a))
```

We assigned the **b = a**, **a** and **b** both point to the same object. When we checked by the **id()** function it returned the same number. We reassign **a** to **500**; then it referred to the new object identifier.

Variable Names

We have already discussed how to declare the valid variable. Variable names can be any length can have uppercase, lowercase (A to Z, a to z), the digit (0-9), and underscore character(_).

Consider the following example of valid variables names.

In [ ]:

```
name = "Rajesh"
age = 20
marks = 80.50
print(name)
print(age)
print(marks)
```

Consider the following valid variables name.

In [ ]:

```
name = "A"
Name = "B"
naMe = "C"
NAME = "D"
n_a_m_e = "E"
_name = "F"
name_ = "G"
_name_ = "H"
na56me = "I"
print(name,Name,naMe,NAME,n_a_m_e, NAME, n_a_m_e, _name, name_,_name, na56me)
```

The multi-word keywords can be created by the following method

- **Camel Case** - In the camel case, each word or abbreviation in the middle of begins with a capital letter. There is no intervention of whitespace. For example - nameOfStudent, valueOfVaraible, etc.
- **Pascal Case** - It is the same as the Camel Case, but here the first word is also capital. For example - NameOfStudent, etc.
- **Snake Case** - In the snake case, Words are separated by the underscore. For example - name_of_student, etc.

Multiple Assignment

Python allows us to assign a value to multiple variables in a single statement, which is also known as multiple assignments.

We can apply multiple assignments in two ways, either by assigning a single value to multiple variables or assigning multiple values to multiple variables.

Consider the following example.

In [ ]:

```
#Assigning single value to multiple variables
x=y=z=50
print(x)
print(y)
print(z)
```

In [ ]:

```
#Assigning multiple values to multiple variables
a,b,c=5,10,15
print(a)
print(b)
print(c)
```

Python Variable Types

There are two types of variables in Python - Local variable and Global variable.

Local Variable

Local variables are the variables that declared inside the function and have scope within the function.

In [ ]:

```
# Declaring a function
def add():
    # Defining local variables. They has scope only within a function
    a = 20
    b = 30
    c = a + b
    print("The sum is:", c)
```

In [ ]:

```
# Calling a function
add()
print(a)
```

Global Variables

Global variables can be used throughout the program, and its scope is in the entire program. We can use global variables inside or outside the function.

A variable declared outside the function is the global variable by default. Python provides the **global** keyword to use global variable inside the function.

If we don't use the **global** keyword, the function treats it as a local variable.

In [ ]:

```
# Declare a variable and initialize it
x = 101

# Global variable in function
def mainFunction():
    # printing a global variable
    #global x
   # print(x)
    # modifying a global variable
    x = 'Welcome To CMR TC'
    print(x)
```

In [ ]:

```
mainFunction()
print(x)
```

In [ ]:

```
print(x)
```

Delete a variable

We can delete the variable using the del keyword.

Syntax: del variable_name

In [ ]:

```
# Assigning a value to x
x = 6
print(x)
# deleting a variable.
del x
print(x)
```

Maximum Possible Value of an Integer in Python

Unlike the other programming languages, Python doesn't have **long int** or **float** data types. **It treats all integer values as an int data type**.

Here, the question arises. What is the maximum possible value can hold by the variable in Python?

In [ ]:

```
# A Python program to display that we can store large numbers in Python

a = 10000000000000000000000000000000000000000000
a = a + 1
print(type(a))
print (a)
```

- Hence, **there is no limitation number by bits and we can expand to the limit of our memory**.
- **Python doesn't have any special data type to store larger numbers**.

Print Single and Multiple Variables in Python

We can print multiple variables within the single print statement.

In [ ]:

```
# printing single value
a = 5
print(a)
print((a))
```

In [ ]:

```
# printing multiple variables
a = 5
b = 6
print(a,b)
# separate the variables by the comma
print(1, 2, 3, 4, 5, 6, 7, 8)
```

More Examples on Variables

In [ ]:

```
counter=100
Miles=550.0
Name='cse'
value=None
print(type(counter))
print(type(Miles))
type(Name)
type(value)
```

In [ ]:

```
print(counter)
print(Miles)
print(Name)
print(value)
```

In [ ]:

```
x = 2
y = 5
xy = 'Hey'
print (x+y, xy)
```

In [ ]:

```
x = y = z = 1
print (x,y,z)
```

In [ ]:

```
x, y, z = 2, 3.0, 'CSE'
print(x,y,z)
print(type(x))
print(type(y))
print(type(z))
```

Python Data Types

Variables can hold values, and every value has a data-type. Python is a dynamically typed language; hence we do not need to define the type of the variable while declaring it. The interpreter implicitly binds the value with its type.

a=5

The variable **a** holds integer value five and we did not define its type. Python interpreter will automatically interpret variables **a** as an integer type.

In [ ]:

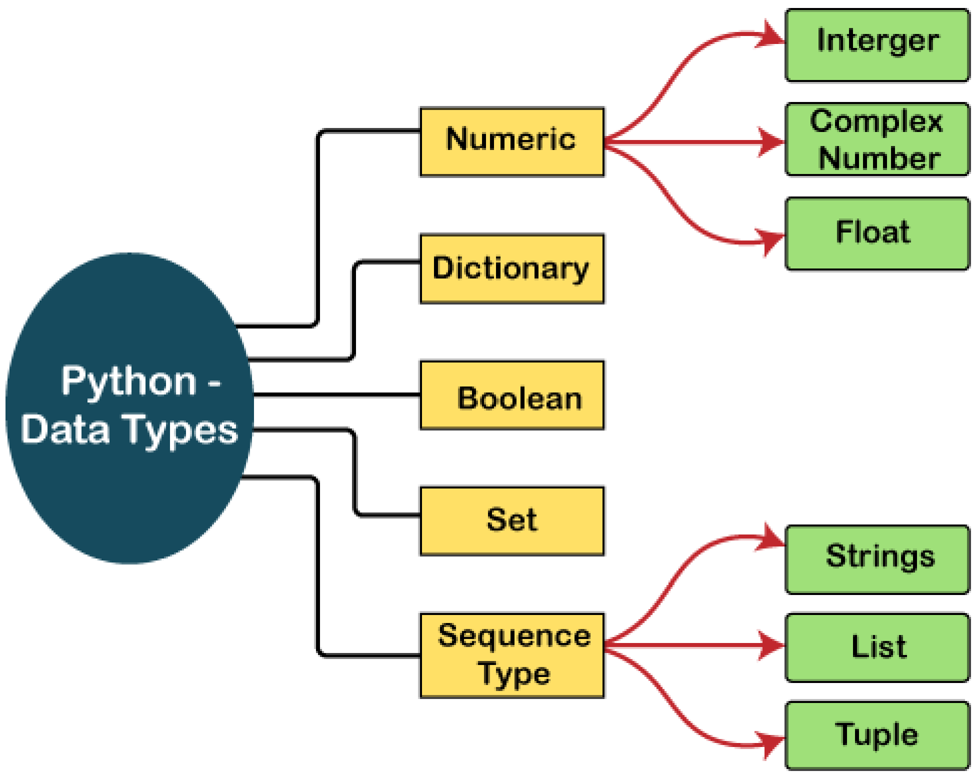
```
a=10
b="Hi Python"
c = 10.5
print(type(a))
print(type(b))
print(type(c))
```

Standard data types

A variable can hold different types of values. For example, a person's name must be stored as a string whereas its id must be stored as an integer.

Python provides various standard data types that define the storage method on each of them. The data types defined in Python are given below.

- **Numbers**
- **Sequence Type**
- **Boolean**
- **Set**
- **Dictionary**



Numbers

Number stores numeric values. The integer, float, and complex values belong to a Python Numbers data-type.

Python provides the **type()** function to know the data-type of the variable. Similarly, the **isinstance()** function is used to check an object belongs to a particular class.

Python creates Number objects when a number is assigned to a variable.

For example

In [ ]:

```python
a = 5
print("The type of a", type(a))

b = 40.5
print("The type of b", type(b))

c = 1+3j
print("The type of c", type(c))
print(" c is a complex number", isinstance(3,complex))
```

Python supports three types of numeric data:

Int - Integer value can be any length such as integers 10, 2, 29, -20, -150 etc. Python has no restriction on the length of an integer. Its value belongs to int

Float - Float is used to store floating-point numbers like 1.9, 9.902, 15.2, etc. It is accurate upto 15 decimal points.

complex - A complex number contains an ordered pair, i.e., x + iy where x and y denote the real and imaginary parts, respectively. The complex numbers like 2.14j, 2.0 + 2.3j, etc.

Sequence Type

String

The string can be defined as the sequence of characters represented in the quotation marks. In Python, we can use single, double, or triple quotes to define a string.

String handling in Python is a straightforward task since Python provides **built-in functions** and **operators** to perform operations in the string.

In the case of string handling, the operator **+** is used to **concatenate** two strings as the operation "hello"+" python" returns "hello python".

The operator * is known as a **repetition operator** as the operation "Python" *2 returns 'Python Python'.

In [ ]:

```python
str = "string using double quotes"
print(str)
s = '''''A multiline
string'''
print(s)
```

```
In [ ]:
```

```
#string handling
str1 = 'hello students' #string str1
str2 = ' how are you' #string str2
print (str1[0:2]) #printing first two character using slice operator
print (str1[4]) #printing 4th character of the string
print (str1*2) #printing the string twice
print (str1 + str2) #printing the concatenation of str1 and str2
```

List

Python Lists are similar to arrays in C. However, the list can contain **data of different types**. The items stored in the list are separated with a comma **(,)** and enclosed within square brackets **[]**.

We can use **slice [:] operators** to access the data of the list. The concatenation operator **(+)** and repetition operator **(*)** works with the list in the same way as they were working with the strings.

```
In [ ]:
```

```
list1  = [1, "hi", "Python", 2.5]
#Checking type of given list
print(type(list1))

#Printing the list1
print (list1)

# List slicing
print (list1[3:])

# List slicing
print (list1[0:2])

# List Concatenation using + operator
print (list1 + list1)

# List repetation using * operator
print (list1 * 3)
```

Tuple

A tuple is similar to the list in many ways. Like lists, tuples also contain the collection of the items of different data types. The items of the tuple are separated with a comma **(,)** and enclosed in parentheses **()**.

A tuple is a read-only data structure as we can't modify the size and value of the items of a tuple.

In [ ]:

```
tup  = ("hi", "Python", 2)
# Checking type of tup
print (type(tup))

#Printing the tuple
print (tup)

# Tuple slicing
print (tup[1:])
print (tup[0:1])

# Tuple concatenation using + operator
print (tup + tup)

# Tuple repatation using * operator
print (tup * 3)

# Adding value to tup. It will throw an error.
tup[2] = "hi"
```

Dictionary

Dictionary is an unordered set of a key-value pair of items. It is like an associative array or a hash table where each key stores a specific value. Key can hold any primitive data type, whereas value is an arbitrary Python object.

The items in the dictionary are separated with the comma **(,)** and enclosed in the curly braces **{}**.

In [ ]:

```
d = {1:'Jimmy', 2:'Alex', 3:'john', 4:'mike'}

# Printing dictionary
print (d)

# Accesing value using keys
print("1st name is "+d[1])
print("2nd name is "+ d[4])

print (d.keys())
print (d.values())
```

Boolean

Boolean type provides **two built-in values**, **True** and **False**. These values are used to determine the given statement **true** or **false**. It denotes by the **class bool**. **True** can be represented by any **non-zero value** or **'T'** whereas **false** can be represented by the **0** or **'F'**.

In [ ]:

```
# Python program to check the boolean type
print(type(True))
print(type(False))
print(false)
```

Set

Python Set is the **unordered collection of the data type**. It is **iterable**, **mutable**(can modify after creation), and **has unique elements**.

In set, the order of the elements is **undefined**; it may **return the changed sequence** of the element. The set is created by using a built-in function **set()**, or a sequence of elements is passed in the **curly braces** and separated by the **comma**. It can contain **various types of values**

In [ ]:

```
# Creating Empty set
set1 = set()

set2 = {'James', 2, 3,'Python'}

#Printing Set value
print(set2)

# Adding element to the set

set2.add(10)
print(set2)

#Removing element from the set
set2.remove(2)
print(set2)
```

Python Operators

The **operator** is a **symbol** that performs a certain **operation between two operands**.

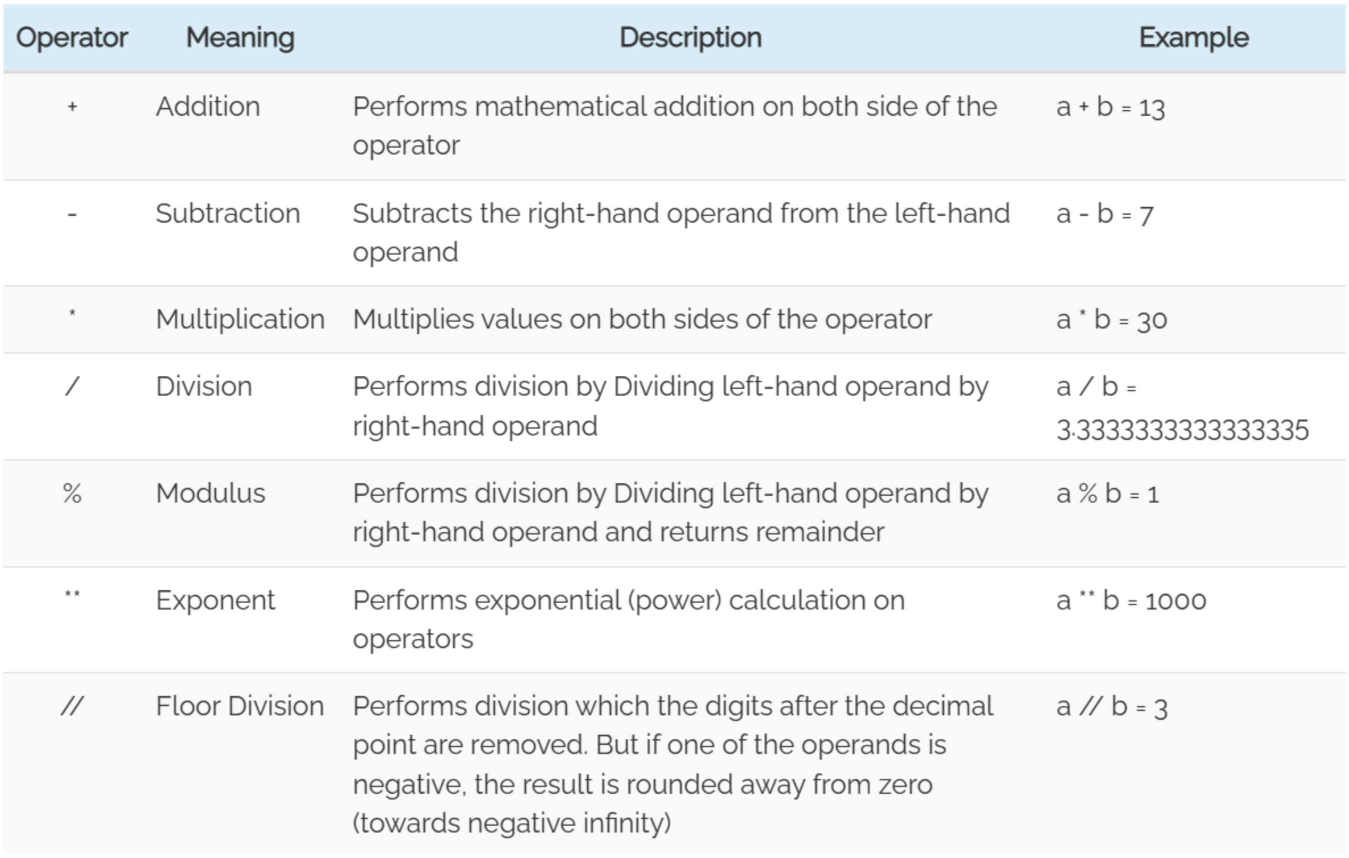
In a particular programming language, operators serve as the foundation upon which logic is constructed in a programme. The different operators that Python offers are listed here.

- **Arithmetic operators**
- **Comparison operators**
- **Assignment Operators**
- **Logical Operators**
- **Bitwise Operators**
- **Membership Operators**
- **Identity Operators**

Arithmetic Operators

Arithmetic operations between two operands are carried out using arithmetic operators. It includes the exponent (**) operator as well as the + (addition), - (subtraction), * (multiplication), / (divide), % (reminder), and // (floor division) operators.

To understand the example let's consider two variables **a** with value **10** and **b** with value **3**.



Operator	Meaning	Description	Example
+	Addition	Performs mathematical addition on both side of the operator	a + b = 13
-	Subtraction	Subtracts the right-hand operand from the left-hand operand	a - b = 7
*	Multiplication	Multiplies values on both sides of the operator	a * b = 30
/	Division	Performs division by Dividing left-hand operand by right-hand operand	a / b = 3.3333333333333335
%	Modulus	Performs division by Dividing left-hand operand by right-hand operand and returns remainder	a % b = 1
**	Exponent	Performs exponential (power) calculation on operators	a ** b = 1000
//	Floor Division	Performs division which the digits after the decimal point are removed. But if one of the operands is negative, the result is rounded away from zero (towards negative infinity)	a // b = 3

In [ ]:

```
a = 10
b = 3
print(f"a + b = {a + b}")
print(f"a - b = {a - b}")
print(f"a * b = {a * b}")
print(f"a / b = {a / b}")
print(f"a % b = {a % b}")
print(f"a ** b = {a ** b}")
print(f"a // b = {a // b}")
```

Comparison Operators

Comparison operators **compare** the values of the **two operands** and return a **true** or **false** Boolean value in accordance. The comparison operators are also known as **Relational Operators**. The following table lists the comparison operators.

To understand the example let's consider two variables **a** with value **10** and **b** with value **3**.

Operator	Meaning	Description	Example
<	Less than	Returns True if left-hand side value is smaller than right-hand side value, otherwise returns False .	a < b (False)
<=	Less than or Equal to	Returns True if left-hand side value is smaller than or equal to right-hand side value, otherwise returns False .	a <= b (False)
>	Greater than	Returns True if left-hand side value is larger than right-hand side value, otherwise returns False .	a > b (True)
>=	Graeter than or Equal to	Returns True if left-hand side value is larger than or equal to right-hand side value, otherwise returns False .	a >= b (True)
==	Equal to	Returns True if left-hand side value is equal to right-hand side value, otherwise returns False .	a == b (False)
!=	Not equal to	Returns True if left-hand side value is not equal to right-hand	a != b

In [ ]:

```
a = 10
b = 3
print(a < b)
print(a <= b)
print(a > b)
print(a >= b)
print(a == b)
print(a != b)
```

Assignment Operators

The **right expression's value** is assigned to the **left operand** using the assignment operators. The following table provides a description of the assignment operators.

To understand the example let's consider two variables **a** with value **10** and **b** with value **3**.

Operator	Meaning	Description	Example
=	Assignment	Assigns right-hand side value to left-hand side variable	a = 10
+=	Add and Assign	Adds right operand to the left operand and assign the result to left operand	a += b ≈ (a = a + b)
-=	Subtract and Assign	Subtracts right operand from the left operand and assign the	a -= b

In [ ]:

```
a = 10
b = 3
a += b
print(f"a += b => {a}")
a -= b
print(f"a -= b => {a}")
a *= b
print(f"a *= b => {a}")
a /= b
print(f"a /= b => {a}")
a %= b
print(f"a %= b => {a}")
a **= b
print(f"a **= b => {a}")
a //= b
print(f"a //= b => {a}")
```

Logical Operators

The **assessment of expressions** to **make decisions** typically makes use of the logical operators. The following logical operators are supported by Python.

To understand the example let's consider two variables **a** with value **10** and **b** with value **3**.

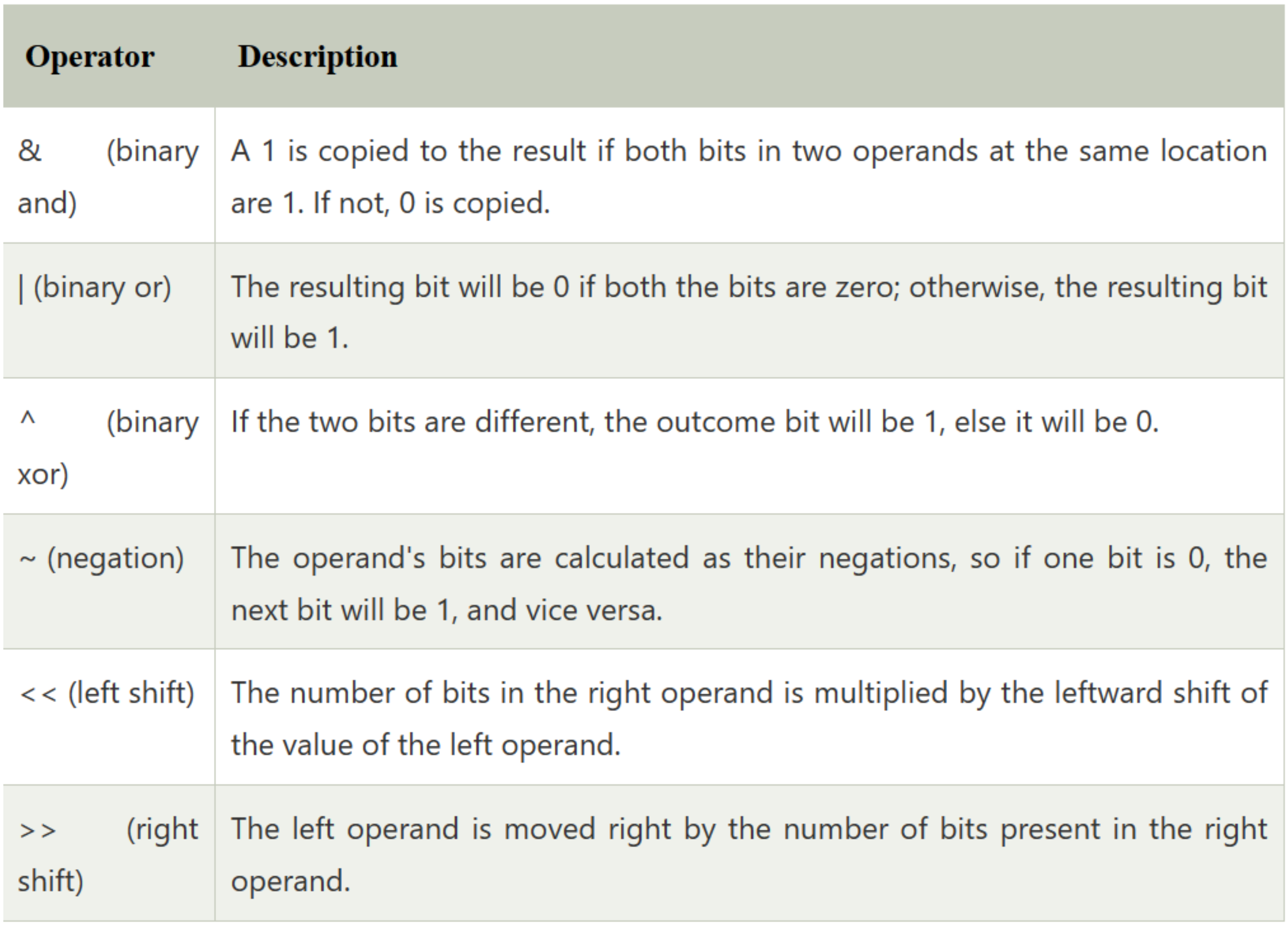
Operator	Meaning	Description	Example
and	Logical AND	Returns `True` if all the conditions are True, otherwise returns `False` .	a < b and a > c
or	Logical OR	Returns `False` if all the conditions are False, otherwise returns `True` .	a < b or a > c
not	Logical NOT	Returns `True` if the condition is False, otherwise returns `False` .	not a > b

In [ ]:

```
a = 10
b = 3
c = 20
print(a < b and a > c)
print(a < b or a > c)
print(not a > b)
```

Bitwise Operators

The bitwise operators are used to performs **bit by bit operation**. There are **six bitwise operators** in Python. The following table presents the list of bitwise operations in Python along with their description.



Operator	Description
& (binary and)	A 1 is copied to the result if both bits in two operands at the same location are 1. If not, 0 is copied.
| (binary or)	The resulting bit will be 0 if both the bits are zero; otherwise, the resulting bit will be 1.
^ (binary xor)	If the two bits are different, the outcome bit will be 1, else it will be 0.
~ (negation)	The operand's bits are calculated as their negations, so if one bit is 0, the next bit will be 1, and vice versa.
<< (left shift)	The number of bits in the right operand is multiplied by the leftward shift of the value of the left operand.
>> (right shift)	The left operand is moved right by the number of bits present in the right operand.

Example

if a = 7
b = 6
then, binary (a) = 0111
binary (b) = 0110

hence, a & b = 0110
a | b = 0111
a ^ b = 0001
~ a = 1000

In [1]:

```
a = 10
b = 15
print(f"{bin(a)} & {bin(b)} = {bin(a & b)}")
print(f"{bin(a)} | {bin(b)} = {bin(a | b)}")
print(f"{bin(a)} ^ {bin(b)} = {bin(a ^ b)}")
print(f"~{bin(a)} = {bin(~a)}")
print(f"{bin(a)} << 2 = {bin(a << 2)}")
print(f"{bin(a)} >> 2 = {bin(a >> 2)}")
```

```
0b1010 & 0b1111 = 0b1010
0b1010 | 0b1111 = 0b1111
0b1010 ^ 0b1111 = 0b101
~0b1010 = -0b1011
0b1010 << 2 = 0b101000
0b1010 >> 2 = 0b10
```

In [ ]:

```
c=8
print(f"~{bin(c)} = {bin(~c)}")
```

In [ ]:

```
d=15
print(f"~{bin(d)} = {bin(~d)}")
```

In [ ]:

```
c=8
print(f"{bin(c)} << 4 = {bin(c << 4)}")
print(f"{bin(c)} >> 4 = {bin(c >> 4)}")
```

Membership Operators

In Python, the membership operators are used to **test** whether a **value is present** in a **sequence**. Here the sequence may be **String**, **List**, or **Tuple**. There are **two membership operators** in Python.

Operator	Meaning	Description	Example
in	in	Returns `True` if it finds a variable in the specified sequence, otherwise returns `False`.	a in list
not in	not in	Returns `True` if it does not finds a variable in the specified sequence, otherwise returns `False`.	a not in list

In [ ]:

```
list = [1, 4, 10, 40, 5]
a = 10
b = 3
print(a in list)
print(b in list)
print(a not in list)
print(b not in list)
print('s' in 'CMR Technical Campus')
```

Identity Operators

In Python, identity operators are used to **comparing** the **memory locations** of two **objects** or **variables**. There are **two identity operators** in Python.

Operator	Meaning	Description	Example
is	is identical	Returns `True` if the variables on either side of the operator point to the same object otherwise returns `False`.	a is b
is not	is not identical	Returns `False` if the variables on either side of the operator point to the same object otherwise returns `True`.	a is not b

In [ ]:

```
a = 10
b = 3
print(a is b)
print(a is not b)
```

In [ ]:

```
a = 10
b = 3
print(a,b)
print(a is b)
a=b
print(a,b)
print(a is b)
print(a is not b)
```

More Examples on Operators

In [ ]:

```
age = 16
if(age>=18):
       print('Your account has been created!')
else:
       print('Sorry, please come back later')
```

In [ ]:

```
age = 25
if(age<=60):
       print('You are eligible to get a permanent Driving Lisence!')
else:
       print('Sorry, You are not eligible for permanent Driving Lisence')
```

In [ ]:

```
a = 2 #10
b = 3 #11
```

In [ ]:

```
a=2
b=3
print(a&b)
print(bin(a&b))
```

In [ ]:

```
a=2
b=3
print(a|b)
print(bin(a|b))
print(a^b)
print(bin(a^b))
print(~a,~b)
print (a & b)
print (bin(a&b))
```

In [ ]:

```
5 >> 1
```

0000 0101 -> 5

Shifting the digits by 1 to the right and zero padding

0000 0010 -> 2

In [ ]:

```
5 << 1
```

0000 0101 -> 5

Shifting the digits by 1 to the left and zero padding

0000 1010 -> 10

In [ ]:

```
#identity operators
x1 = 5
y1 = 5
print(x1 is y1)
x2 = 'Hello'
y2 = 'Hello'
print(x2 is y2)
x3 = [1,2,3]
y3 = [1,2,3]
print(x3 is y3)

print(x1 is not y1)

print(x2 is y2)

print(x3 is y3)
```

In [ ]:

```
#membership operator
x = 'Hello world'
y = [1,'a',2,'b']

print('H' in x)

print('hello' not in x)

print(1 in y)

print('a' in y)
```

Built-in Functions

Python comes loaded with pre-built functions

Conversion from one system to another

Conversion from hexadecimal to decimal is done by adding prefix **0x** to the hexadecimal value or vice versa by using built in **hex( )**, Octal to decimal by adding **prefix 0** to the octal value or vice versa by using built in function **oct( )**.

In [2]:

```
hex(170)
```

Out[2]:

```
'0xaa'
```

In [3]:

```
0xAA
```

Out[3]:

```
170
```

In [4]:

```
oct(8)
```

Out[4]:

```
'0o10'
```

In [5]:

```
0o10
```

Out[5]:

```
8
```

In [6]:

```
chr(99)
```

Out[6]:

```
'c'
```

In [7]:

```
ord('b')
```

Out[7]:

```
98
```

Simplifying Arithmetic Operations

round( ) function rounds the input value to a specified number of places or to the nearest integer.

In [8]:

```
print (round(5.6231))
print (round(4.55892, 2))
```

```
6
4.56
```

complex( ) is used to define a complex number and **abs( )** outputs the absolute value of the same.

In [9]:

```
c =complex('5+2j')
print (abs(c))
```

```
5.385164807134504
```

divmod(x,y) outputs the quotient and the remainder in a tuple.

In [10]:

```
divmod(9,2)
```

Out[10]:

```
(4, 1)
```

isinstance( ) returns True, if the first argument is an instance of that class. Multiple classes can also be checked at once.

In [11]:

```
print (isinstance(1, int))
print (isinstance(1.0,int))
print (isinstance(1.0,(int,float)))
```

```
True
False
True
```

pow(x,y,z) can be used to find the power x^y also the mod of the resulting value with the third specified number can be found i.e. : (x^y % z).

In [12]:

```
print (pow(3,3))
print (pow(3,3,5))
```

```
27
2
```

In [13]:

```
#Write a program to evaluate the expression x=yz-w+r*t/q
y=2
z=5
w=3
r=6
t=8
q=4
x=y**z-w+r*t/q
print(x)
```

```
41.0
```

In [14]:

```
#Write a program to prompt the user to enter the number of working Hours and rate per ho
#Findout the gross pay.
hour=input("Enter the number of hours worked:")
rate=input("Enter the rate per Hour:")
H=int(hour)
R=int(rate)
Gross_Pay=H*R
print(Gross_Pay)
```

```
Enter the number of hours worked:6
Enter the rate per Hour:3000
18000
```

In [15]:

```
#Write a program which prompts the user for a Celsius temperature, convert the temperatu
#Fahrenheit, and print out the converted temperature.
cel=input("Enter the Temprature in celsius:")
c= float(cel)
f=c*9/5+32
print(f)
```

```
Enter the Temprature in celsius:25
77.0
```

In [19]:

```
print("Welcome to pythons class")
print("ABC")
print('A','B','C')
print('a','b','c', sep = "$")
print("welcome to pythons class", end=";")

x = 10
y = 20

# My output should be like The sum of 10 and 20 is 30.

#print("The sum of {} and {} is {}".format(x,y,x+y))
print(f"The sum of {x} and {y} is {x+y}")
```

```
Welcome to pythons class
ABC
A B C
a$b$c
welcome to pythons class;The sum of 10 and 20 is 30
```

Input and Output statements

In Python, input and output operations (OI Operations) are performed using two built-in functions. Following are the two built-in functions to perform output operations and input operations.

- **print( )** - Used for output operation.
- **input( )** - Used for input operations.

Output Operations using print() function

The built-in function **print( )** is used to output the given data to the standard output device.

Displaying a message using print( ) function

In Python, using the print( ) function we can display a text message. The print( ) takes a string as an argument and displays the same.

When we use the print( ) function to display a message, the string can be enclosed either in the **single quotation** or **double quotation**.

In [17]:

```
print('Welcome to CMR Technical Campus')

print("Department of Computer Science and engineering")
```

```
Welcome to CMR Technical Campus
Department of Computer Science and engineering
```

Displaying a variable value using print( ) function

In Python, the built-in function print( ) function is also used to display variables value. The following example shows that how to display variable value using print( ) function.

In [18]:

```
num1 = 10   #variale num1 with value 10
print(num1)
```

```
10
```

Formatted print( ) function to display a combination of message and value

The built-in function **print( )** is also used to display the combination of message and variable value.

In Python, we can use the **formatted string** that is prefixed with character **"f"**. In the formatted string, the variable values are included using curly braces **({ })**.

In [20]:

```
num1 = 10   #variale num1 with value 10
print('The value of variable num1 is ',num1)
```

```
The value of variable num1 is  10
```

In [21]:

```
num1 = 10   #variale num1 with value 10
print(f"The value of variable num1 is {num1}")
```

```
The value of variable num1 is 10
```

Note: Always the return value of **print( )** function is **None**.

Input Operations using input() function

The Python provides a built-in function **input( )** to perform input operations. The input( ) function can be used either with some message or without a message.

When input( ) function is used without a message, it simply prompts for the input value. When the user enters the input value it reads the entered value as a string and assigns it to the left-hand-side variable.

In [23]:

```
number_1 = input()
print(f'The number you have entered is {number_1}')
```

```
cmr
The number you have entered is cmr
```

In [24]:

```
number_1 = input('Enter any number: ')
print(f'The number you have entered is {number_1}')
```

```
Enter any number: 123
The number you have entered is 123
```

Note: Always the **input( )** function reads input value as **string** value only.

To read the value of any other data type, there is no input function in Python. Explicitly we need to convert to the required data type using type casting.

Reading a single character in Python

The Python does not provide any direct function to read a single character as input. But we can use the index value to extract a single character from the user entered input value.

In [28]:

```
ch = input('Enter any data: ')[2:4]
print(f'The character entered is {ch}')
```

```
Enter any data: 217R1A0514
The character entered is 7R
```

Note: When we run the above piece of code, the input( ) function reads complete data entered by the user but assigns an only first character to the variable ch.

Control Statements

In Python, the default execution flow of a program is a sequential order. But the sequential order of execution flow may not be suitable for all situations. Sometimes, we may want to **jump** from line to another line, we may want to **skip** a part of the program, or sometimes we may want to execute a **part of the program** again and again. To solve this problem, Python provides control statements.

In Python, the control statements are the statements which will tell us that in which order the instructions are getting executed. The control statements are used to control the order of execution according to our requirements. Python provides several control statements, and they are classified as follows.

- **Selection Control Statements (Decision Making Statements)**.
- **Iterative Control Statements (Looping Statements)**.

Selection Control Statements

In Python, the selection statements are also known as decision making statements or branching statements. The selection statements are used to select a part of the program to be executed based on a condition. Python provides the following selection statements.

- if statement
- if-else statement

- elif statement

if statement

In Python, we use the if statement to test a condition and decide the execution of a block of statements based on that condition result.

The if statement checks, the given condition then decides the execution of a block of statements. If it is True, then the block of statements is executed and if it is False, then the block of statements is ignored.

if%20statement.png

Syntax:

if condition:

```
Statement_1
Statement_2
Statement_3
...
```

When we define an if statement, the block of statements must be specified using **indentation** only. The **indentation** is a series of white-spaces. Here, the number of white-spaces is variable, but all statements must use the identical number of white-spaces.

In [4]:

```python
num = int(input('Enter any number: '))
if (num % 5 == 0):
    print(f'Given number {num} is divisible by 5')
    print('This statement belongs to if statement')
print('This statement does not belongs to if statement')
```

```
Enter any number: a

---------------------------------------------------------------------------
-
ValueError                                Traceback (most recent call las
t)
Cell In[4], line 1
----> 1 num = int(input('Enter any number: '))
      2 if (num % 5 == 0):
      3     print(f'Given number {num} is divisible by 5')

ValueError: invalid literal for int() with base 10: 'a'
```

In [6]:

```python
#even or not
num = int(input("enter the number?"))
if num%2 == 0:
    print("Number is even")
```

```
enter the number?4
Number is even
```

In [7]:

```python
#print the largest of the three numbers
a = int(input("Enter a? "));
b = int(input("Enter b? "));
c = int(input("Enter c? "));
if a>b and a>c:
    print("a is largest");
if b>a and b>c:
    print("b is largest");
if c>a and c>b:
    print("c is largest");
```

```
Enter a? 2
Enter b? 1
Enter c? 5
c is largest
```

if-else statement

In Python, we use the if-else statement to test a condition and pick the execution of a block of statements out of two blocks based on that condition result.

The if-else statement checks the given condition then decides which block of statements to be executed based on the condition result. If the condition is True, then the true block of statements is executed and if it is False, then the false block of statements is executed.

if-else%20statement.png

Syntax:

if condition:

```
        Statement_1
        Statement_2
        Statement_3
        ...
```

else:

```
        Statement_4
        Statement_5
        ...
```

In [ ]:

```
# testing whether a given number is Even or Odd
num = int(input('Enter any number : '))
if num % 2 == 0:
    print(f'The number {num} is a Even number')
else:
    print(f'The number {num} is a Odd number')
```

In [ ]:

```
#Program to check whether a person is eligible to vote or not.
age = int (input("Enter your age? "))
if age>=18:
    print("You are eligible to vote !!");
else:
    print("Sorry! you have to wait !!");
```

elif statement

The elif statement enables us to check multiple conditions and execute the specific block of statements depending upon the true condition among them. We can have any number of elif statements in our program depending upon our need. However, using elif is optional.

The elif statement works like an if-else-if ladder statement in C. It must be succeeded by an if statement.

elif%20statement.png

Syntax:

if condition_1:

```
Statement_1
Statement_2
Statement_3
...
```

elif condition_2:

```
Statement_4
Statement_5
Statement_6
...
```

else:

```
Statement_7
Statement_8
...
```

In [9]:

```
# Python code to illustrate elif statement
choice = input(f'Which game do you like, Press\nC - Cricket\nH - Hokey: ')
if choice == 'C':
    print('You are a Cricketer!')
elif choice == 'H':
    print('You are a Hockey player!')
else:
    print('You are not interested in Sports')
```

```
Which game do you like, Press
C - Cricket
H - Hokey: A
You are not interested in Sports
```

In [ ]:

```
#program for number position
number = int(input("Enter the number?"))
if number==10:
    print("number is equals to 10")
elif number==50:
    print("number is equal to 50");
elif number==100:
    print("number is equal to 100");
else:
    print("number is not equal to 10, 50 or 100");
```

In [ ]:

```
#program grade calculation
marks = int(input("Enter the marks? "))
if marks > 85 and marks <= 100:
   print("Congrats ! you scored grade A ...")
elif marks > 60 and marks <= 85:
   print("You scored grade B + ...")
elif marks > 40 and marks <= 60:
   print("You scored grade B ...")
elif (marks > 30 and marks <= 40):
   print("You scored grade C ...")
else:
   print("Sorry you are fail ?")
```

More Examples

In [ ]:

```
x = 12
if x > 10:
    print('hello')
```

In [ ]:

```
x = 5
if x >10:
    print('hello')
```

In [10]:

```
x = input()
x1=int(x)
if x1 > 10:
    print ("hello")
else:
    print ("world")
```

```
abs

---------------------------------------------------------------------------
-
ValueError                                Traceback (most recent call las
t)
Cell In[10], line 2
      1 x = input()
----> 2 x1=int(x)
      3 if x1 > 10:
      4     print ("hello")

ValueError: invalid literal for int() with base 10: 'abs'
```

In [ ]:

```
x = input()
x1=int(x)
y = input()
y1=int(y)
if x1 > y1:
    print ("x1>y1")
elif x1 < y1:
    print ("x1<y1")
else:
    print("x1=y1")
```

In [ ]:

```
x = 10
y = 12
if x > y:
    print ("x>y")
elif x < y:
    print ("x<y")
    if x==10:
        print("x=10")
    else:
        print ("invalid")
else:
    print ("x=y")
```

In [11]:

```
#Write a program to find out the given Year is Leap Year or Not.
year=input("Enter the Year:")
x=int(year)
if x%4 ==0:
    print("The Given Year is a Leap Year:")
else:
    print("The Given Year is not a Leap Year:")
```

```
Enter the Year:2023
The Given Year is not a Leap Year:
```

In [ ]:

```
#Write a program to prompt the user for hours and rate per hour using input to compute g
#Pay should be the normal rate for hours up to 40 and time-and-a-half for the hourly rat
#all hours worked above 40 hours.
hrs = input("Enter Hours:")
h = float(hrs)
rate = input("Enter the Rate per Hour:")
rate1=float(rate)
rate2=rate1*1.5
if h>40 :
    pay=(40.0*rate1+(h-40)*rate2)
    print(pay)
else :
    pay=h*rate1
    print(pay)
```

- Write a program to prompt for a score between 0.0 and 1.0. If the score is out of range, print an error. If the score is between 0.0 and 1.0, print a grade using the following table: Score Grade >= 0.9 is A, >= 0.8 is B, >= 0.7 is C, >= 0.6 is D, < 0.6 is F

In [ ]:

```
score = input("Enter Score: ")
s = float(score)

if s >= 0.0 and s < 0.6:
    print('F')

elif s >= 0.6 and s < 0.7:
    print('D')

elif s >= 0.7 and s < 0.8:
    print('C')

elif s >= 0.8 and s < 0.9:
    print('B')

elif s >= 0.9 and s <= 1.0:
    print('A')

else:
    print('Error: input not within range')
```

Iterative Control Statements in Python

In Python, the **iterative statements** are also known as **looping statements** or **repetitive statements**. The iterative statements are used to execute a **part of the program repeatedly** as long as the given condition is **True**.

Using iterative statements **reduces the size of the code**, **reduces the code complexity**, **makes it more efficient**, and **increases the execution speed**.

Python provides the following iterative statements.

- **while statement**
- **for statement**
- **Loop Control Statements**

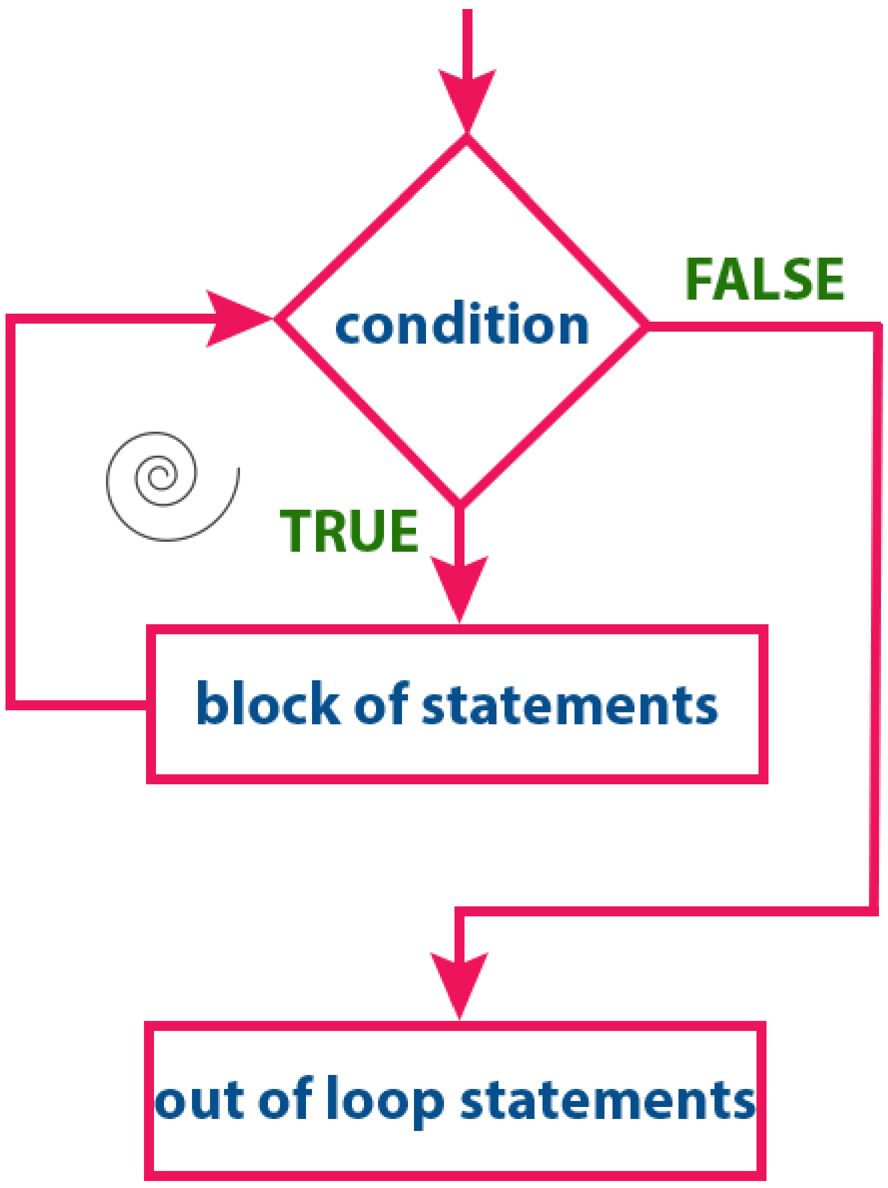
while statement

In Python, the **while** statement is used to execute a **set of statements repeatedly**. In Python, the while statement is also known as **entry control loop statement** because in the case of the while statement, first, the given condition is verified then the execution of statements is determined based on the condition result.

Syntax:

while condition:

```
    Statement_1
    Statement_2
    Statement_3
    ...
```



When we define a while statement, the block of statements must be specified using **indentation** only. The indentation is a series of white-spaces. Here, the number of white-spaces may variable, but all statements must use the identical number of white-spaces.

In [1]:

```
# Python program to show how to use a while loop
counter = 0
# Initiating the loop
while counter < 10: # giving the condition
    counter = counter + 3
    print("CMR Technical Campus")
```

```
CMR Technical Campus
CMR Technical Campus
CMR Technical Campus
CMR Technical Campus
```

In [2]:

```
count = int(input('How many times you want to say "CSE Department": '))
i = 1
while i <= count:
    print('CSE Department')
    i += 1
print('Job is done! Thank you!!')
```

```
How many times you want to say "CSE Department": 12
CSE Department
CSE Department
CSE Department
CSE Department
CSE Department
CSE Department
CSE Department
CSE Department
CSE Department
CSE Department
CSE Department
CSE Department
Job is done! Thank you!!
```

Note: When we use a while statement, the value of the variable used in the condition must be modified otherwise while loop gets into the infinite loop.

while statement with 'else' clause

In Python, the **else clause** can be used with a while statement. The else block is gets executed whenever the condition of the while statement is evaluated to false. But, if the while loop is terminated with **break** statement then else doesn't execute.

In [ ]:

```
#Python program to show how to use else statement with the while loop
counter = 0

# Iterating through the while loop
while (counter < 10):
    counter = counter + 3
    print("CMR Technical Campus") # Executed untile condition is met
# Once the condition of while loop gives False this statement will be executed
else:
    print("Code block inside the else statement")
```

```
In [ ]:
```

```python
# Here, else block is gets executed because break statement does not executed
count = int(input('How many times you want to say "CSE Department": '))
i = 1
while i <= count:
    if count > 10:
        print('I cann\'t say more than 10 times!')
        break
    print('CSE Department')
    i += 1
else:
    print('This is else block of while!!!')
print('Job is done! Thank you!!')
```

for statement

In Python, the for statement is used to iterate through a **sequence** like a **list**, a **tuple**, a **set**, a **dictionary**, or a **string**.

The for statement is used to **repeat the execution of a set of statements** for **every element of a sequence**.

the **variable is stored with each element** from the **sequence for every iteration**.

Syntax:

for **variable** in **sequence**:

```
        Statement_1
        Statement_2
        Statement_3
        ...
```

```
In [3]:
```

```python
# Python code to illustrate for statement with List
my_list = [1, 2, 3, 4, 5]
for value in my_list:
    print(value)
print('Job is done!')
```

```
1
2
3
4
5
Job is done!
```

In [4]:

```python
# Python program to show how the for loop works

# Creating a sequence which is a tuple of numbers
numbers = [4, 2, 6, 7, 3, 5, 8, 10, 6, 1, 9, 2]

# variable to store the square of the number
square = 0

# Creating an empty list
squares = []

# Creating a for loop
for value in numbers:
    square = value ** 2
    squares.append(square)
print("The list of squares is", squares)
```

```
The list of squares is [16, 4, 36, 49, 9, 25, 64, 100, 36, 1, 81, 4]
```

In [5]:

```python
# Python code to illustrate for statement with Tuple
my_tuple = (1, 2, 3, 4, 5)
for value in my_tuple:
    print(value)
print('Job is done!')
```

```
1
2
3
4
5
Job is done!
```

In [6]:

```python
# Python code to illustrate for statement with Set
my_set = {1, 2, 3, 4, 5}
for value in my_set:
    print(value)
print('Job is done!')
```

```
1
2
3
4
5
Job is done!
```

In [7]:

```python
# Python code to illustrate for statement with Dictionary
my_dictionary = {1:'Rama', 2:'Seetha', 3:'Heyaansh', 4:'Gouthami', 5:'Raja'}
for key, value in my_dictionary.items():
    print(f'{key} --> {value}')
print('Job is done!')
```

```
1 --> Rama
2 --> Seetha
3 --> Heyaansh
4 --> Gouthami
5 --> Raja
Job is done!
```

In [8]:

```python
# Python code to illustrate for statement with String
for item in 'Python':
    print(item)
print('Job is done!')
```

```
P
y
t
h
o
n
Job is done!
```

In [9]:

```python
# Python code to illustrate for statement with Range function
for value in range(1, 6):
    print(value)
print('Job is done!')
```

```
1
2
3
4
5
Job is done!
```

In [ ]:

```python
# Python program to iterate over a sequence with the help of indexing

tuple_ = ("Python", "Loops", "Sequence", "Condition", "Range")

# iterating over tuple_ using range() function
for iterator in range(len(tuple_)):
    print(tuple_[iterator].upper())
```

for statement with 'else' clause

In Python, the **else clause** can be used with a for a statement. The else block is gets executed whenever the **for statement is does not terminated with a break statement**. But, if the for loop is terminated with

In [ ]:

```
# Here, else block is gets executed because break statement does not executed
for item in 'Python':
    if item == 'x':
        break
    print(item)
else:
    print('else block says for is successfully completed!')
print('Job is done!!')
```

In [ ]:

```
# Python program to show how if-else statements work

str1 = 'Python Loop'

# Initiating a loop
for s in str1:
    # giving a condition in if block
    if s == "o":
        print("If block")
    # if condition is not satisfied then else block will be executed
    else:
        print(s)
```

In [ ]:

```
# Here, else block does not gets executed because break statement terminates the loop.
for item in 'Python':
    if item == 'y':
        break
    print(item)
else:
    print('else block says for is successfully completed!')
print('Job is done!!')
```

In [ ]:

```
# Python program to show how to use else statement with for loop

# Creating a sequence
tuple_ = (3, 4, 6, 8, 9, 2, 3, 8, 9, 7)

# Initiating the loop
for value in tuple_:
    if value % 2 != 0:
        print(value)
# giving an else statement
else:
    print("These are the odd numbers present in the tuple")
```

Loop Control Statements

Statements used to **control loops and change the course of iteration** are called control statements.

All the objects produced within the local scope of the loop are deleted when execution is completed.

- **Break statement**
- **Continue statement**
- **Pass statement**

Break statement

This command terminates the loop's execution and transfers the program's control to the statement next to the loop.

In [10]:

```
# Python program to show how the break statement works

# Initiating the loop
for string in "CMR Technical Campus":
    if string == 'T':
         break
    print('Current Letter: ', string)
```

```
Current Letter:  C
Current Letter:  M
Current Letter:  R
Current Letter:
```

Continue statement

This command **skips the current iteration of the loop**.

The statements following the continue statement are not executed once the Python interpreter reaches the continue statement.

In [11]:

```python
# Python program to show how the continue statement works

# Initiating the loop
for string in "CMR Technical Campus":
    if string == "a" or string == "c" or string == "e":
        continue
    print('Current Letter:', string)
```

```
Current Letter: C
Current Letter: M
Current Letter: R
Current Letter:  
Current Letter: T
Current Letter: h
Current Letter: n
Current Letter: i
Current Letter: l
Current Letter:  
Current Letter: C
Current Letter: m
Current Letter: p
Current Letter: u
Current Letter: s
```

Pass statement

The pass statement is used when a statement is syntactically necessary, but no code is to be executed.

In [12]:

```python
# Python program to show how the pass statement works
for string in "CMR Technical Campus":
    pass
print( 'Last Letter:', string)
```

```
Last Letter: s
```

More Examples on Control Statement

In [ ]:

```
# List of integer numbers
numbers = [1, 2, 4, 6, 11, 20]

# variable to store the square of each num temporary
sq = 0

# iterating over the given list
for val in numbers:
    # calculating square of each number
    sq = val * val
    # displaying the squares
    print(sq)
```

In [ ]:

```
# Program to print the sum of first 5 natural numbers

# variable to store the sum
sum = 0

# iterating over natural numbers using range()
for val in range(1, 6):
    # calculating sum
    sum = sum + val

# displaying sum of first 5 natural numbers
print(sum)
```

In [13]:

```
for val in range(5):
    print(val)
else:
    print("The loop has completed execution")
```

```
0
1
2
3
4
The loop has completed execution
```

In [ ]:

```
for num1 in range(3):
    for num2 in range(10, 14):
        print(num1, ",", num2)
```

In [ ]:

```
num = 1
# loop will repeat itself as long as
# num < 10 remains true
while num < 10:
    print(num)
    #incrementing the value of num
    num = num + 3
```

In [ ]:

```
i = 1
j = 5
while i < 4:
    while j < 8:
        print(i, ",", j)
        j = j + 1
        i = i + 1
```

In [ ]:

```
# program to display all the elements before number 88
for num in [11, 9, 88, 10, 90, 3, 19]:
    print(num)
    if(num==88):
        print("The number 88 is found")
        print("Terminating the loop")
        break
```

In [ ]:

```
#program to print odd numbers between 1 and 20.
for num in range(1,20):
    # Skipping the iteration when number is even
    if num%2 == 0:
        continue
    # This statement will be skipped for all even numbers
    print(num)
```

In [ ]:

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print("Factors of ", n, " are ", x,",",int(n/x))
            break
```

In [ ]:

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print("Factors of ", n, " are ", x,",",int(n/x))
            break
    else:
        print(n, " is prime")
```

In [ ]:

```
num = 10

if num < 20:
    print("Given number is less than 20")
else:
    pass
```

Python else suite

In Python, it is possible to use **'else'** statement along with **for loop** or **while loop**.

The else suite will be always executed irrespective of the statements in the loop are executed or not.

for loop with else syntax

for(var in sequence)

```
statements
```

else:

```
statements
```

while loop with else syntax

while(condition):

```
statements
```

else:

```
statements
```

In [ ]:

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

In [ ]:

```
#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 group')
        break #come out of for loop
else:
    print('Element not found in group1') #this is else suite
```

In [ ]:

```
#Write a python program to check whether given number is Prime number or not?
num = int(input("Enter a number: "))
if num > 1:
   for i in range(2,num):
       if (num % i) == 0:
            print(num,"is not a prime number")
            print(i,"times",num//i,"is",num)
            break
   else:
        print(num,"is a prime number")
else:
    print(num,"is not a prime number")
```

In [ ]:

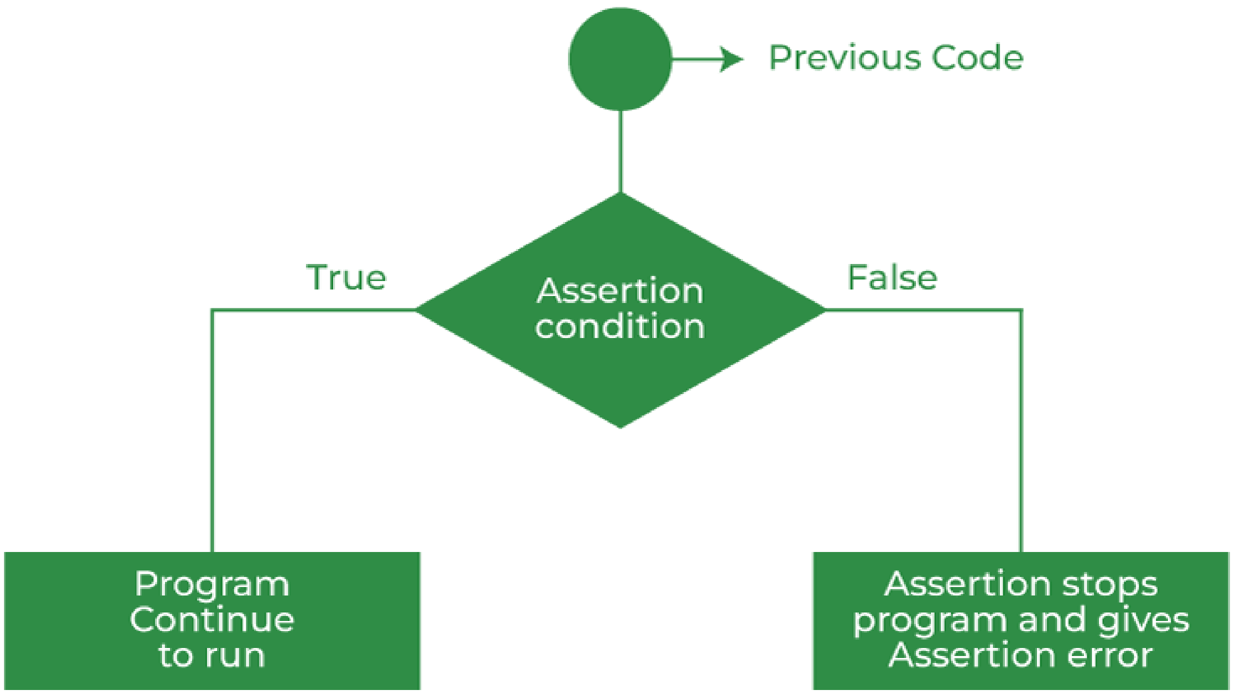
```
#Write a python program to list of Prime numbers
lower = int(input("Enter starting number : "))
upper = int(input("Enter ending number: "))
for num in range(lower,upper + 1):
   if num > 1:
       for i in range(2,num):
           if (num % i) == 0:
               break
       else:
          print(num,end='\t')
```

Python Assert Statement

In Python, the **assert statement** is used to **continue the execute if the given condition evaluates to True**.

If the **assert condition** evaluates to **False**, then it raises the **AssertionError exception** with the **specified error message**.

Assertions are simply **boolean expressions** that check if the conditions return **true** or **not**. If it is **true**, the program **does nothing** and **moves to the next line of code**. However, if it's **false**, the **program stops** and **throws an error**.



Syntax:

Case-1: without Error message

assert condition

Case-2: with Error message

assert condition, error message

Using assert without Error Message

In [ ]:

```
def avg(marks):
    assert len(marks) != 0
    return sum(marks)/len(marks)

mark1 = []
print("Average of mark1:",avg(mark1))
```

In [ ]:

```
x = 10
assert x > 0
print('x is a positive number.')
```

Using assert with error message

In [ ]:

```
x = 0
assert x > 0, 'Only positive numbers are allowed'
print('x is a positive number.')
```

In [ ]:

```
def square(x):
    assert x>=0, 'Only positive numbers are allowed'
    return x*x

n = square(2) # returns 4
n = square(-2) # raise an AssertionError
```

In [ ]:

```
def avg(marks):
    assert len(marks) != 0,"List is empty."
    return sum(marks)/len(marks)

mark2 = [55,88,78,90,79]
print("Average of mark2:",avg(mark2))

mark1 = []
print("Average of mark1:",avg(mark1))
```

In [ ]:

```
def square(x):
    assert x>=0, 'Only positive numbers are allowed'
    return x*x

try:
    square(-2)
except AssertionError as msg:
    print(msg)
```

In [ ]:

```
def KelvinToFahrenheit(Temperature):
   assert (Temperature >= 0),"Colder than absolute zero!"
   return ((Temperature-273)*1.8)+32

print(KelvinToFahrenheit(273))
print(int(KelvinToFahrenheit(505.78)))
print (KelvinToFahrenheit(-5))
```

In [ ]:

```
# initializing list of foods temperatures
batch = [ 40, 26, 39, 30, 25, 21]

# initializing cut temperature
cut = 26

# using assert to check for temperature greater than cut
for i in batch:
    assert i >= 26, "Batch is Rejected"
    print (str(i) + " is O.K" )
```

Python return statement

A **return** statement is used to **end the execution of the function call** and **“returns”** the **result (value of the expression following the return keyword) to the caller**. The statements after the return statements are not executed.

If the **return statement is without any expression**, then the special value **None** is returned. A return statement is overall used to **invoke a function** so that the passed statements can be executed.

Note: Return statement can not be used outside the function.

Syntax:

def fun():

```
statements
.
.
return [expression]
```

In [ ]:

```
def cube(x):
   r=x**3
   return r
```

In [ ]:

```
def myfunction():
  return 3+3

print(myfunction())
```

In [ ]:

```
def myfunction():
  return 3+3
  print("Hello, World!")

print(myfunction())
```

In [ ]:

```
# Python program to
# demonstrate return statement

def add(a, b):

    # returning sum of a and b
    return a + b

def is_true(a):

    # returning boolean of a
    return bool(a)

# calling function
res = add(2, 3)
print("Result of add function is {}".format(res))

res = is_true(2<5)
print("\nResult of is_true function is {}".format(res))
```

In [ ]:

```
# A Python program to return multiple
# values from a method using tuple

# This function returns a tuple
def fun():
    str = "CMR Technical Campus"
    x = 20
    return str, x;  # Return tuple, we could also
                    # write (str, x)

# Driver code to test above method
str, x = fun() # Assign returned tuple
print(str)
print(x)
```

In [ ]:

```
# A Python program to return multiple
# values from a method using list

# This function returns a list
def fun():
    str = "CMR Technical Campus"
    x = 20
    return [str, x];

# Driver code to test above method
list = fun()
print(list)
```

In [ ]:

```
# A Python program to return multiple
# values from a method using dictionary

# This function returns a dictionary
def fun():
    d = dict();
    d['str'] = "CMR Technical Campus"
    d['x']   = 20
    return d

# Driver code to test above method
d = fun()
print(d)
```

In [ ]:

```

```