Python Tutorial

What is python?

Python is a simple, general purpose, high level, and object-oriented programming language.Python is an interpreted scripting language also.
Python programming language (latest Python 3) is being used in web development, Machine Learning applications, along with all cutting-edge technology in Software Industry. Python Programming Language is very well suited for Beginners, also for experienced programmers with other programming languages like C++ and Java.

History Of Python –

Python was developed by Guido van Rossum in the late eighties and early nineties at the National Research Institute for Mathematics and Computer Science in the Netherlands.

Python 3.10. 0 is the newest major release of the Python programming language, and it contains many new features and optimizations.

Why Used Python?

Python is currently the most widely used multi-purpose, high-level programming language. The various areas of Python use are given below.

- Data Science
- Data Mining
- Desktop Applications
- Console-based Applications
- Mobile Applications
- Software Development
- Artificial Intelligence
- Web Applications
- Enterprise Applications
- 3D CAD Applications
- Machine Learning
- Computer Vision or Image Processing Applications.
- Speech Recognitions

Why Learn Python?

Python provides many useful features to the programmer. These features make it most popular and widely used language. We have listed below few-essential feature of Python.

- Easy to use and Learn
- Expressive Language
- Interpreted Language
- Object-Oriented Language
- Open-Source Language
- Extensible
- Learn Standard Library
- GUI Programming Support - Tkinter
- Integrated
- Embeddable
- Dynamic Memory Allocation
- Wide Range of Libraries and Frameworks

Python Popular Frameworks and Libraries

Python has wide range of libraries and frameworks widely used in various fields such as machine learning, artificial intelligence, web applications, etc. We define some popular frameworks and libraries of Python as follows.

Web development (Server-side) - Django, Flask, Pyramid, CherryPy
GUIs based applications - Tkinter, PyGTK, PyQt, PyJs, etc.
Machine Learning - TensorFlow, PyTorch, **Scikit-learn**, Matplotlib, Scipy, etc.
Mathematics - Numpy, Pandas, etc.

Difference between Python and Java?

Parameter	**Python**	**Java**
Code	Python has less lines of code.	Java has longer lines of code.
Framework	Compared to JAVA, Python has lower number of Frameworks. Popular ones are DJango, Flask.	Java has large number of Frameworks. Popular ones are Spring, Hibernate, etc.
Syntax	Syntax is easy to remember almost similar to human language.	Syntax is complex as it throws error if you miss semicolon or curly braces.
Key Features	Less line no of code, Rapid deployment and dynamic typing.	Self memory management, Robust, Platform independent
Speed	Python is slower since it uses interpreter and also determines the data type at run time.	Java is faster in speed as compared to python.
Databases	Python’s database access layers are weaker than Java’s JDBC. This is why it rarely used in enterprises.	(JDBC)Java Database Connectivity is most popular and widely used to connect with database.
Machine Learning Libraries	Tensorflow, Pytorch.	Weka, Mallet, Deeplearning4j, MOA

How to download and install python in MAC/Windows OS?

Download The python.

Step 1 – Type Python in crome browser.
Step 2 – Click on python official website.
https://www.python.org/
Step 3 – Keep the mouse cursor in downloads.
Step 4 – If you are using widows OS then click on windows
Step 5 – If you are using mac OS then click on Mac.
Step 5 – Select the latest version.
Step 6 – Download.

Install Python

Step 1 – Go to the download folder
Step 2 – Select python
Step 3 – double click on python folder
Step 4 – Choose the Path.
Step 5 – After all steps click on done.

Check Python is install or not.

Step 1 – Open cmd or terminal.
Step 2 – Type python --version

Virtual environment

virtualenv is used **to manage Python packages for different projects**. Using virtualenv allows you to avoid installing Python packages globally which could break system tools or other projects. You can install virtualenv using pip.

```
# Create a virtual environment to isolate our package dependencies locally
        python3 -m venv env
        source env/bin/activate

 # On Windows use
        `env\Scripts\activate`
```

Installing Jupyter Noteboook / VS Code/PyCharm for Windows

Step 1 – Type pycharm download in the browser.
Step 2 - Click on first link
Step. 3 – download Community version.

Python Basic syntax –

There is no use of curly braces or semicolon in Python programming language. It is English-like language. But Python uses the indentation to define a block of code. Indentation is nothing but adding whitespace before the statement when it is needed.

Ex –

```
def XYZ:
        print("abc")
```

If we don't have any statement, then we can write pass keyword inside the function.

```
def XYZ:
        pass
```

In the above example, the statements that are same level to right belong to the function. Generally, we can use four whitespaces to define indentation, also we can use tab keyword.

Useful Tips– tab keyword is used to provide indentation space and shift+tab to provide back step.

Python First Program

Unlike the other programming languages, Python provides the facility to execute the code using few lines. **For Example -** Suppose we want to print the **"Hello World"** program in Java; it will take three lines to print it.

```
public class HelloWorld {
 public static void main(String[] args){
// Prints "Hello, World" to the terminal window.
  System.out.println("Hello World");
 }
 }
```

On the other hand, we can do this using one statement in Python.

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

Both programs will print the same result, but it takes only one statement without using a semicolon or curly braces in Python.

Quize -

Quize 1 – Which syntax is correct?

```
def ABC:
Statement1
…………………
…………………
Statement n

def ABC:
        Statement 1
        ……………………..
        ……………………..
        statement n
```

Example 1: Add Two Numbers

```
# This program adds two numbers

 num1 = 1.5
 num2 = 6.3

 # Add two numbers
 sum = num1 + num2

 # Display the sum
 print('The sum of {0} and {1} is {2}'.format(num1, num2, sum))
```

Example 2: Add Two Numbers With User Input

```
# Store input numbers
num1 = input('Enter first number: ‘)
num2 = input('Enter second number: ‘)

# Add two numbers
sum = float(num1) + float(num2)

# Display the sum
print('The sum of {0} and {1} is {2}'.format(num1, num2, sum))
```

Differences between Python 2.x and Python 3.x with examples

- ❖ Division operator
- ❖ print function
- ❖ Unicode
- ❖ Xrange
- ❖ Error Handling
- ❖ __future__ module

1. Division operator - If we are porting our code or executing python 3.x code in python 2.x, it can be dangerous if integer division changes go unnoticed (since it doesn't raise any error). It is preferred to use the floating value (like 7.0/5 or 7/5.0) to get the expected result when porting our code.

Ex -
```
print 7 / 5
print -7 / 5
```

Output in Python 2.x
```
1
-2
```

Output in Python 3.x :
```
1.4
-1.4
```

2. print function - This is the most well-known change. In this, the **print** keyword in Python 2.x is replaced by the **print()** function in Python 3.x. However, parentheses work in Python 2 if space is added after the **print** keyword because the interpreter evaluates it as an expression.

Ex-
```
print 'Hello, Geeks'      # Python 3.x doesn't support
print('Hope You like these facts’)
```

Note - if we use parentheses in python 2.x then there is no issue but if we don’t use parentheses in python 3.x, we get SyntaxError.

3. Unicode - In Python 2, an implicit str type is ASCII. But in Python 3.x implicit str type is Unicode.

Ex -
```
print(type('default string'))
print(type(b'string with b ‘))

'”Output in Python 2.x (Bytes is same as str)
  <type 'str’>
  <type 'str’>

Output in Python 3.x (Bytes and str are different)
 <class 'str'>
 <class 'bytes'>
 '''
```

Python 2.x also supports Unicode

Ex -
```
print(type('default string '))
print(type(u'string with b '))

'''
Output in Python 2.x (Unicode and str are different)

<type 'str'>

<type 'unicode'>

Output in Python 3.x (Unicode and str are same)

<class 'str'>

<class 'str'>

'''
```

4. Xrange - xrange() of Python 2.x doesn’t exist in Python 3.x. In Python 2.x, range returns a list i.e. range(3) returns [0, 1, 2] while xrange returns a xrange object i. e., xrange(3) returns iterator object which works similar to Java iterator and generates number when needed.
If we need to iterate over the same sequence multiple times, we prefer range() as range provides a static list. xrange() reconstructs the sequence every time. xrange() doesn’t support slices and other list methods. The advantage of xrange() is, it saves memory when the task is to iterate over a large range.
In Python 3.x, the range function now does what xrange does in Python 2.x, so to keep our code portable, we might want to stick to using a range instead. So Python 3.x’s range function *is* xrange from Python 2.x.

Ex -
```
for x in xrange(1, 5):
        print(x)
```

Ex -
```
for x in range(1, 5):
        print(x)
```

```
'’’
Output in Python 2.x
1 2 3 4
1 2 3 4

Output in Python 3.x
NameError: name 'xrange' is not defined
'''
```

Error Handling - There is a small change in error handling in both versions. In python 3.x, ‘as’ keyword is required.

EX-

```
try:
    trying_to_check_error
except NameError, err:
    print err, 'Error Caused'   # Would not work in Python 3.x

'''
Output in Python 2.x:

name 'trying_to_check_error' is not defined Error Caused

Output in Python 3.x :

File "a.py", line 3

    except NameError, err:
                    ^
SyntaxError: invalid syntax
'’’
```

Ex -

```
try:

    trying_to_check_error

except NameError as err: # 'as' is needed in Python 3.x

    print (err, 'Error Caused')


'''
```

Output in Python 2.x:

```
(NameError("name 'trying_to_check_error' is not defined",), 'Error Caused')
```

Output in Python 3.x :

```
name 'trying_to_check_error' is not defined Error Caused

'''
```

6. __future__ module

This is basically not a difference between the two versions, but a useful thing to mention here. The idea of the __future__ module is to help migrate to Python 3.x.
If we are planning to have Python 3.x support in our 2.x code, we can use **_future_** imports in our code.
For example, in the Python 2.x code below, we use Python 3.x's integer division behavior using the __future__ module.

```
# In below python 2.x code, division works
# same as Python 3.x because we use  __future__
```

Ex -

```
from __future__ import division

print 7 / 5
print -7 / 5
```

Output -

```
1.4
-1.4
```

Ex-2

```
from __future__ import print_function
print(”IAmLearningPython”)
```

Output –

```
IAmLearningPython
```

Python Variables

What is a Variable in Python?

A Python variable is a reserved memory location to store values. In other words, a variable in a python program gives data to the computer for processing.

Python Variable Types

Every value in Python has a datatype. Different data types in Python are Numbers, List, Tuple, Strings, Dictionary, etc. Vari ables in Python can be declared by any name or even alphabets like a, aa, abc, etc.

How to Declare and use a Variable

```
a=100.0
print (a)
```

Re-declare a Variable

You can re-declare Python variables even after you have declared once.
Here we have Python declare variable initialized to f=0.
Later, we re-assign the variable f to value “py99”

```
# Declare a variable and initialize it
f = 0
Print(f)
# re-declaring the variable works
f = 'py9999'
Print(f)
```

Example

```
# Declare a variable and initialize it
x = 0
y=1
print(x)
print(y)

# re-declaring the variable works
y = ‘python'
print(y)
```

Python String Concatenation and Variable

Let’s see whether you can concatenate different data types like string and number together. For example, we will concatenate “xyz” with the number “99”.
Unlike Java, which concatenates number with string without declaring number as string, while declaring variables in Python requires declaring the number as string otherwise it will show a TypeError

Example

```
a=”xyz”
b = 99
print(a+b)
```

Python Variable Types: Local & Global

here are two types of variables in Python, Global variable and Local variable. When you want to use the same variable for rest of your program or module you declare it as a global variable, while if you want to use the variable in a specific function or method, you use a local variable while Python variable declaration.

Let's understand this Python variable types with the difference between local and global variables in the below program.

1. Let us define variable in Python where the variable "f" is **global** in scope and is assigned value 101 which is printed in output.
2. Variable f is again declared in function and assumes **local** scope. It is assigned value "I am learning Python." which is printed out as an output. This Python declare variable is different from the global variable "f" defined earlier
3. Once the function call is over, the local variable f is destroyed. when we again, print the value of "f" is it displays the value of global variable f=101

```
# Declare a variable and initialize it
   f = 101
   Print(f)
   # Global vs. local variables in functions

   def someFunction():
   # global f
      f = 'I am learning Python'
      print(f)
   someFunction()
   Print(f)
```

```
# Declare a variable and initialize it
        f = 101
        print(f)
        # Global vs. local variables in functions
        def someFunction():
        # global f
           f = 'I am learning Python'
           print(f)
        someFunction()
        print(f)
```

While Python variable declaration using the keyword **global,** you can reference the global variable inside a function.
Variable “f” is **global** in scope and is assigned value 101 which is printed in output
Variable f is declared using the keyword **global**. This is **NOT** a **local variable**, but the same global variable declared earlier. Hence when we print its value, the out put is 101
We changed the value of “f” inside the function. Once the function call is over, the changed value of the variable “f” persists.when we again, print the value of “f” is it displays the value “changing global variable”

Example -

```
f = 101;
print f
# Global vs.local variables in functions
def someFunction():
  global f
  print f
  f = "changing global variable"
someFunction()
print f
```

Example -

```
f = 101;
print(f)
# Global vs.local variables in functions
def someFunction():
  global f
  print(f)
  f = "changing global variable"
someFunction()
print(f)
```

Delete a variable

You can also delete Python variables using the command **del** “variable name”.
In the below example of Python delete variable, we deleted variable f, and when we proceed to print it, we get error “**variable name is not defined**” which means you have deleted the variable.

Example -

```
f = 11;
print(f)
del(f)
print(f)
```

Summary:

- Variables are referred to “envelop” or “buckets” where information can be maintained and referenced. Like any other programming language Python also uses a variable to store the information.
- Variables can be declared by any name or even alphabets like a, aa, abc, etc.
- Variables can be re-declared even after you have declared them for once
- Python constants can be understood as types of variables that hold the value which can not be changed. Usually, Python constants are referenced from other files. Python define constant is declared in a new or separate file which contains functions, modules, etc.
- Types of variables in Python or Python variable types : Local & Global
- Declare local variable when you want to use it for current function
- Declare Global variable when you want to use the same variable for rest of the program
- To delete a variable, it uses keyword “del”.

Python Operators: Arithmetic, Logical, Comparison, Assignment, Bitwise & Precedence

What is operators?

The operator can be defined as a symbol which is responsible for a particular operation between two operands. Operators are the pillars of a program on which the logic is built in a specific programming language.

Types Of Operators

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

Arithmetic Operators

Arithmetic Operators perform various arithmetic calculations like addition, subtraction, multiplication, division, %modulus, exponent, etc. There are various methods for arithmetic calculation in Python like you can use the eval function, declare variable & calculate, or call functions.

Ex – Addition

```
x= 4
y= 5
Z =  x+y
print(x + y)
o/p = 9
```

Similarly, you can use other arithmetic operators like for multiplication(*), division (/), substraction (-), etc.

Ex – Division

```
print (7 / 5)
print(7 / 5)
o/p = 1.4
o/p = -1.4
```

```
a = 10
b=20

def addval(a,b):
        c= a+b
        print(c)

addval(a,b)
```

Comparison Operators

Comparison Operators In Python compares the values on either side of the operand and determines the relation between them. It is also referred to as relational operators. Various comparison operators in python are (==, != , <>, >,<=, etc.)

Ex - For comparison operators we will compare the value of x to the value of y and print the result in true or false.

```
x = 4
y = 5
print(('x > y  is’, x>y))
```

Likewise, you can try other comparison operators (x < y, x==y, x!=y, etc.)

Ex-

```
a= 20
b= 20
print(a <= b)
```

Assignment Operators

Assignment Operators in **Python** are used for assigning the value of the right operand to the left operand. Various assignment operators used in Python are (+=, – = , *=, /= , etc.).

Ex -

```
num1 = 4
num2 = 5
Num3 =0
Num3 += num1
Num3 = num3+num1
print(("Line  1 - Value of num1 : ", num1))
print(("Line  2 - Value of num2 : ", num2))
```

Example of compound assignment operator

We can also use a compound assignment operator, where you can add, subtract, multiply right operand to left and assign addition (or any other arithmetic function) to the left operand.

Step 1: Assign value to num1 and num2
Step 2: Add value of num1 and num2 (4+5=9)
Step 3: To this result add num1 to the output of Step 2 (9+4)
Step 4: It will print the final result as 13

Ex-

```
num1 = 4
num2 = 5
res = num1 + num2
```

Logical Operators or Bitwise Operators

Logical operators in Python are used for conditional statements are true or false. Logical operators in Python are AND, OR and NOT. For logical operators following condition are applied.

- For AND operator – It returns TRUE if both the operands (right side and left side) are true
- For OR operator- It returns TRUE if either of the operand (right side or left side) is true
- For NOT operator- returns TRUE if operand is false

Ex –

```
a = true
b = false
print(('a and b is’, a and b))
print(('a or b is’, a or b))
print(('not a is’, not a))
```

Membership Operators

These operators test for membership in a sequence such as lists, strings or tuples. There are two membership operators that are used in Python. (in, not in). It gives the result based on the variable present in specified sequence or string.

Ex -

```
x = 4
y = 8
list = [1, 2, 3, 4, 5 ]
if ( x in list ):
   print("Line 1 - x is available in the given list")
else:
   print("Line 1 - x is not available in the given list")
if ( y not in list ):
   print("Line 2 - y is not available in the given list")
else:
   print("Line 2 - y is available in the given list")
```

- Declare the value for x and y
- Declare the value of list
- Use the “in” operator in code with if statement to check the value of x existing in the list and print the result accordingly
- Use the “not in” operator in code with if statement to check the value of y exist in the list and print the result accordingly
- Run the code- When the code run it gives the desired output

Identity Operators

Identity Operators in Python are used to compare the memory location of two objects. The two identity operators used in Python are (is, is not).

Operator is: It returns true if two variables point the same object and false otherwise

Operator is not: It returns false if two variables point the same object and true otherwise

A = (20/10)+30*120-30**(40+60)

Operators (Decreasing order of precedence)	Meaning
**	Exponent
*, /, //, %	Multiplication, Division, Floor division, Modulus
+, –	Addition, Subtraction
<= < > >=	Comparison operators
= %= /= //= -= += *= **=	Assignment Operators
is is not	Identity operators
in not in	Membership operators
not or and	Logical operators

Ex-

```
x = 20
y = 20
if ( x is y ):
    print("x & y  SAME  identity")
y=30
if ( x is not y ):
    print("x & y have DIFFERENT identity")
```

Summary:

- Operators in a programming language are used to perform various operations on values and variables. In Python, you can use operators like
- There are various methods for arithmetic calculation in Python as you can use the eval function, declare variable & calculate, or call functions
- Comparison operators often referred as relational operators are used to compare the values on either side of them and determine the relation between them
- Python assignment operators are simply to assign the value to variable
- Python also allows you to use a compound assignment operator, in a complicated arithmetic calculation, where you can assign the result of one operand to the other.
- For AND operator – It returns TRUE if both the operands (right side and left side) are true
- For OR operator- It returns TRUE if either of the operand (right side or left side) is true
- For NOT operator- returns TRUE if operand is false
- There are two membership operators that are used in Python. (in, not in).
- It gives the result based on the variable present in specified sequence or string
- The two identify operators used in Python are (is, is not)
- It returns true if two variables point the same object and false otherwise
- Precedence operator can be useful when you have to set priority for which calculation need to be done first in a complex calculation.

Python Comments

Python Comment is an essential tool for the programmers. Comments are generally used to explain the code. We can easily understand the code if it has a proper explanation. A good programmer must use the comments because in the future anyone wants to modify the code as well as implement the new module; then, it can be done easily.

Types of comments.

1. Single line
2. Multi-line comments
3. Docstrings Comment

Single line -

To apply the comment in the code we use the hash(#) at the beginning of the statement or code.

Ex -

```
#This is the print statement
print("Hello Python")
```

Multi-line comments

We must use the hash(#) at the beginning of every line of code to apply the multiline Python comment.

Ex -

```
# First line of the comment
# Second line of the comment
# Third line of the comment
a,b = 5 ,10
print("The sum is:", a+b )
```

Docstrings Comment

The docstring comment is mostly used in the module, function, class or method. It is a documentation Python string.

Ex -

```
def intro():
  """
  This function prints  Hi python
   hello
   hi
  """
  print("Hi  python")

intro()
```

Output: Hi python

We can check a function's docstring by using the __doc__ attribute.

Generally, four whitespaces are used as the indentation. The amount of indentation depends on user, but it must be consistent throughout that block.

Ex -

```
def intro():
  """
  This function prints  Hi python
  """
  print("  Hi python  ")
intro.__doc__
```

Output: '\n This function prints Hello Joseph\n '

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.
Python enables us to check the type of the variable used in the program. Python provides us the **type()** function, which returns the type of the variable passed.

EX –

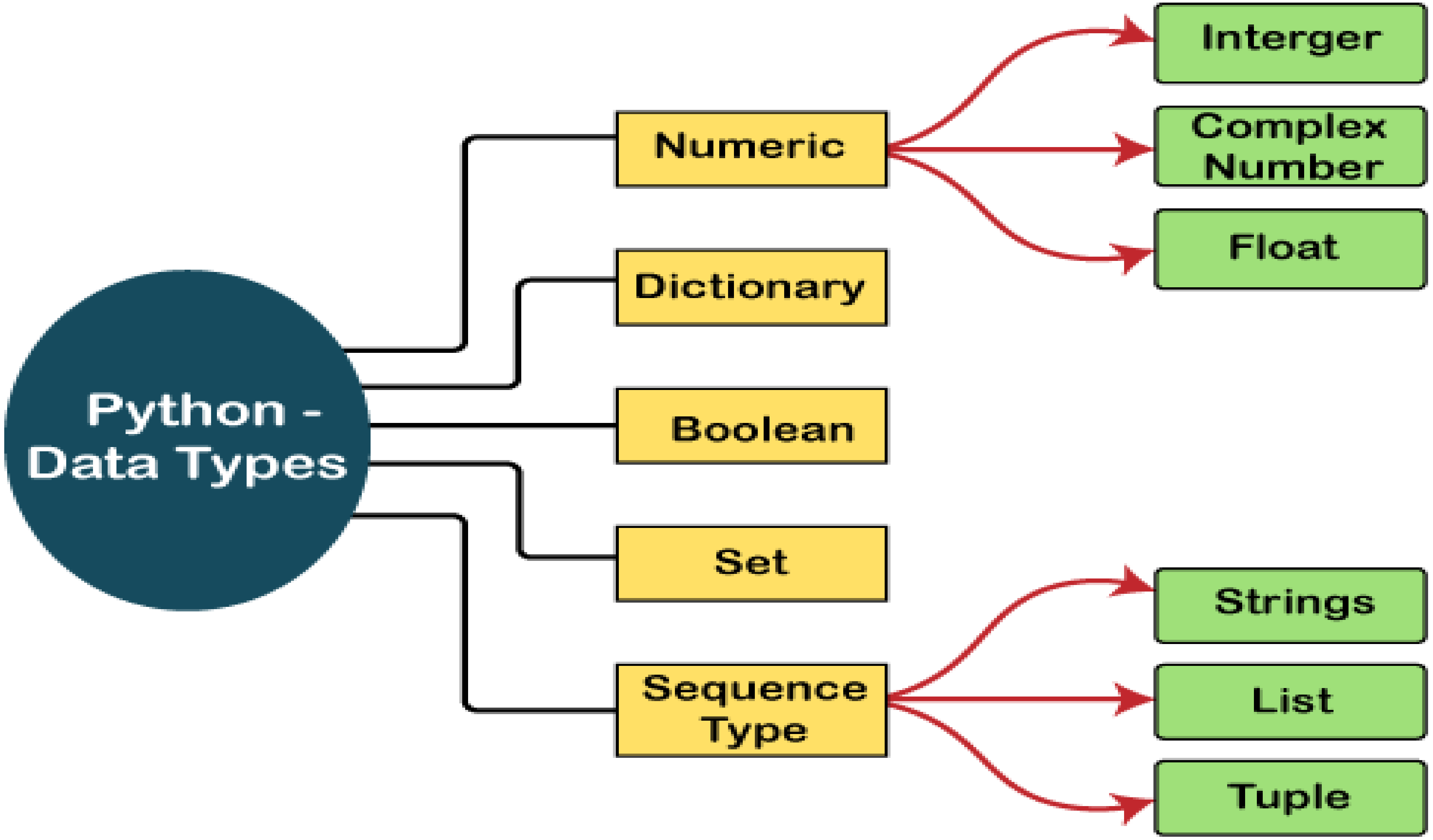
```
a=10
b="Hi Python"
c = 10.5
print(type(a))
print(type(b))
print(type(c))
```

Output-

```
<type 'int’> <type 'str’><type 'float'>
```

Python has the following data types built-in by default, in these categories:

Text Type:	str
Numeric Types:	int, float, complex
Sequence Types:	list, tuple, range
Mapping Type:	dict
Set Types:	set, frozenset
Boolean Type:	bool
Binary Types:	bytes, bytearray, memoryview


Python - Data Types
Numeric
Dictionary
Boolean
Set
Sequence Type
Interger
Complex Number
Float
Strings
List
Tuple

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;

```
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(1+3j,complex))
```

Output:

```
The type of a <class 'int’>
 The type of b <class 'float’>
The type of c <class 'complex’>
c is complex number: True
```

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

In Python, sequence is the ordered collection of similar or different data types. Sequences allows to store multiple values in an organized and efficient fashion. There are several sequence types in Python –

- String
- List
- Tuple

String

A string is a collection of one or more characters put in a single quote, double-quote or triple quote. In python there is no character data type, a character is a string of length one. It is represented by str class.

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'.

Ex-

```
# Creating a String with single Quotes
String1 = 'Welcome to the Python World'
print("String with the use of Single Quotes: ")
print(String1)
```

Output –

```
Welcome to the Python World
```

# Creating a String with double Quotes

```
String1 = “I'm a Python”
print("\nString with the use of Double Quotes: ")
print(String1)
print(type(String1))
```

Output –

```
I'm a Geek <class 'str’>
```

# Creating a String with triple Quotes

```
String1 = '''I'm a Python “and” I live in a world of " Program"''
print("\nString with the use of Triple Quotes: ")
print(String1)
print(type(String1))
```

Output –

```
I'm a Python and I live in a world of " Program”
<class 'str'>
```

```
# Creating String with triple Quotes allows multiple lines \

String1  = ”’Python
            For
            Life'''
print("\nCreating a multiline  String:  ")
print(String1)
```

Output –

```
Creating a multiline String:
’Python
For
Life
```

Ex-

```
str1 = 'hello  world’     #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
```

OutPut-

```
he
h
Hello world Hello world
Hello world how are you
```

LIST –

Lists are just like dynamically sized arrays, declared in other languages (vector in C++ and ArrayList in Java). Lists need not be homogeneous always which makes it the most powerful tool in Python.
A single list may contain DataTypes like Integers, Strings, as well as Objects.
Lists are mutable, and hence, they can be altered even after their creation.

```
L1 = [1,’a’,7.5,2,15,’b’,’jiya’,’a’,’a’]
        (0,1,2,3,4,5)
```

List in Python are ordered and have a definite count. The elements in a list are indexed according to a definite sequence and the indexing of a list is done with 0 being the first index. Each element in the list has its definite place in the list, which allows duplicating of elements in the list, with each element having its own distinct place and credibility.

Note- Lists are a useful tool for preserving a sequence of data and further iterating over it.

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.

Creating a List

Lists in Python can be created by just placing the sequence inside the square brackets[]. Unlike Sets, a list doesn't need a built-in function for the creation of a list.

```
L1 =[]
L2 = [11,23,'a','xyz']
L2 =list(11,23,'a','xyz')
```

Set Ex-

```
Set1= set('a',12,31)
```

Note – Unlike Sets, the list may contain mutable elements.

1.Creating a List

```
List1 = []
print("Blank List: ")
print(List)
```

2.Creating a List of numbers

```
list1 = [10,20,30]
print("\nList of numbers: ”,list1)
print(List)
```

3.Creating a List of strings and accessing

```
# using index
List1 = [”I", ”am", ”fine"]
print("\nList Items: ")
print(List1[0])
print(List1[2])
```

4.Creating a Multi-Dimensional List

```
# (By Nesting a list inside a List)
List1 = [[I’, am'], [‘fine']]
print("\nMulti-Dimensional List: ")
print(List1)
```

OutPut –

```
Blank List:
[]

List of numbers:
[10, 20, 14]

List Items
I
Fine

Multi-Dimensional
List: [[I’, am'], [fine']]
```

Tuple

Tuple is a collection of Python objects much like a list. The sequence of values stored in a tuple can be of any type, and they are indexed by integers. Tuple is an immutable.

Values of a tuple are syntactically separated by ‘commas’. Although it is not necessary, it is more common to define a tuple by closing the sequence of values in parentheses. This helps in understanding the Python tuples more easily.

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.

Ex –

```
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])
```

Ex -

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



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

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

Output –

```
 <class 'tuple’>
 ('hi', 'Python', 2)
 ('Python', 2)
 ('hi’,)
('hi', 'Python', 2, 'hi', 'Python', 2)
('hi', 'Python', 2, 'hi', 'Python', 2, 'hi', 'Python', 2)

 Traceback (most recent call last):
 File "main.py", line 14, in <module>
 t[2] = "hi";
 TypeError: 'tuple' object does not support item assignment
```

## Creating a Tuple with Mixed Datatypes.

Tuples can contain any number of elements and of any datatype (like strings, integers, list, etc.). Tuples can also be created with a single element, but it is a bit tricky. Having one element in the parentheses is not sufficient, there must be a trailing ‘comma’ to make it a tuple.

```
# Creating tuple with Mixed Datatype
Tuple1 = (5, 'Welcome', 7, 'Geeks')
print("\nTuple with Mixed Datatypes: ")
print(Tuple1)

# Creating a Tuple
# with nested tuples
Tuple1 = (0, 1, 2, 3)
Tuple2 = ('python', 'geek')
Tuple3 = (Tuple1, Tuple2)
print("\nTuple with nested tuples: ")
print(Tuple3)
```

```
# Creating a Tuple
# with repetition
Tuple1 = ('Geeks',) * 3
print("\nTuple with repetition: ")
print(Tuple1)

# Creating a Tuple
# with the use of loop
Tuple1 = ('Geeks')
n = 5
print("\nTuple with a loop")
for i in range(int(n)):
    Tuple1 = (Tuple1,)
    print(Tuple1)
```

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. Consider the following example.

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

Set is an unordered collection of data type that is iterable, mutable and has no duplicate elements. The order of elements in a set is undefined though it may consist of various elements.
The major advantage of using a set, as opposed to a list, is that it has a highly optimized method for checking whether a specific element is contained in the set.

Creating a Set

Sets can be created by using the built-in **set()** function with an iterable object or a sequence by placing the sequence inside curly braces, separated by ‘comma’.
Note – A set cannot have mutable elements like a list or dictionary, as it is mutable.

A set contains only unique elements but at the time of set creation, multiple duplicate values can also be passed. Order of elements in a set is undefined and is unchangeable. Type of elements in a set need not be the same, various mixed up data type values can also be passed to the set.

EX -

```
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)
```

```
# Creating a Set
set1 = set()
print("Initial blank Set: ")
print(set1)

# Creating a Set with the use of a String
set1 = set(”welcomeToPython")
print("\nSet with the use of String:  ")
print(set1)

# Creating a Set with the use of Constructor
# (Using object to Store String)
String = welcomeToPython '
set1 = set(String)
print("\nSet with the use of an Object: " )
print(set1)

# Creating a Set with the use of a List
set1 = set([”welcome", ”to", ”python"])
print("\nSet with the use of List: ")
print(set1)
```

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 {}.

Dictionary in Python is an unordered collection of data values, used to store data values like a map, which, unlike other Data Types that hold only a single value as an element, Dictionary holds **key:value** pair. Key-value is provided in the dictionary to make it more optimized.

Note – Keys in a dictionary don't allow Polymorphism.

Creating a Dictionary

In Python, a Dictionary can be created by placing a sequence of elements within curly **{}** braces, separated by 'comma'. Dictionary holds pairs of values, one being the Key and the other corresponding pair element being its **Key:value**. Values in a dictionary can be of any data type and can be duplicated, whereas keys can't be repeated and must be *immutable*.

Note – Dictionary keys are case sensitive, the same name but different cases of Key will be treated distinctly.

```
# Creating a Dictionary with Integer Keys
Dict = {1: 'Geeks’, 2: 'For', 3: 'Geeks'}
print("\nDictionary with the use of Integer Keys: ")
print(Dict)

# Creating a Dictionary with Mixed keys
Dict = {'Name': 'Geeks', 1: [1, 2, 3, 4]}
print("\nDictionary with the use of Mixed Keys: ")
print(Dict)
```

Dictionary can also be created by the built-in function dict(). An empty dictionary can be created by just placing to curly braces{}.

```
# Creating an empty Dictionary
Dict = {}
print("Empty Dictionary: ")
print(Dict)
```

```
# Creating a Dictionary with dict() method
Dict = dict({1: 'Geeks', 2: 'For', 3:'Geeks'})
print("\nDictionary with the use of dict(): ")
print(Dict)

# Creating a Dictionary with each item as a Pair
Dict = dict([(1, 'Geeks'), (2, 'For')])
print("\nDictionary with each item as a pair: ")
print(Dict)
```

Nested Dictionary

EX -

```
Dict = {1: 'Geeks', 2: 'For',
        3:{'A' : 'Welcome', 'B' : 'To', 'C' : 'Geeks'}}

print(Dict)
```

EX-

```
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'. Consider the following example.

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

Arrays

An array is defined as a collection of items that are stored at contiguous memory locations. It is a container which can hold a fixed number of items, and these items should be of the same type. An array is popular in most programming languages like C/C++, JavaScript, etc.
Array is an idea of storing multiple items of the same type together and it makes easier to calculate the position of each element by simply adding an offset to the base value. A combination of the arrays could save a lot of time by reducing the overall size of the code. It is used to store multiple values in single variable. If you have a list of items that are stored in their corresponding variables like this:

The array can be handled in Python by a module named **array**. It is useful when we have to manipulate only specific data values. Following are the terms to understand the concept of an array:
Element - Each item stored in an array is called an element.
Index - The location of an element in an array has a numerical index, which is used to identify the position of the element.

Array Representation

An array can be declared in various ways and different languages. The important points that should be considered are as follows:
Index starts with 0.
We can access each element via its index.
The length of the array defines the capacity to store the elements.

Creating a Array

Array in Python can be created by importing array module. **array(*data_type*, *value_list*)** is used to create an array with data type and value list specified in its arguments.

```
# Creation of Array
# importing "array" for array creations
import array as arr
```

```
# creating an array with integer type
a = arr.array('i', [1, 2, 3])

# printing original array
print ("The new created array is : ", end =" ")
for i in range (0, 3):
    print (a[i], end =" ")
print()

# creating an array with float type
b = arr.array('d', [2.5, 3.2, 3.3])

# printing original array
print ("The new created array is : ", end =" ")
for i in range (0, 3):
    print (b[i], end =" ")
```

Output -

```
The new created array is : 1 2 3
The new created array is : 2.5 3.2 3.3
```

Type Conversion in Python

Python defines type conversion functions to directly convert one data type to another which is useful in day-to-day and competitive programming. This article is aimed at providing information about certain conversion functions.
There are two types of Type Conversion in Python:
Implicit Type Conversion
Explicit Type Conversion
Let's discuss them in detail.

Implicit Type Conversion

In Implicit type conversion of data types in Python, the Python interpreter automatically converts one data type to another without any user involvement. To get a more clear view of the topic see the below examples.

Ex-

```
x = 10
print("x is of type:",type(x))

y = 10.6
print("y is of type:",type(y))

x = x + y
print(x)
print("x is of type:",type(x))
```

Output -

```
 x is of type: <class 'int’>
 y is of type: <class 'float’>
 20.6
 x is of type: <class 'float'>
```

Explicit Type Conversion

In Explicit Type Conversion in Python, the data type is manually changed by the user as per their requirement. Various forms of explicit type conversion are explained below:

1. **int(a, base)**: This function converts **any data type to integer**. ‘Base’ specifies the **base in which string is** if the data type is a string.
   2. float(): This function is used to convert **any data type to a** floating-point **number**

Ex-

```
# initializing string
s = "10010”

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

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

Output -

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

3. ord() : This function is used to convert a **character to integer.**
4. hex() : This function is to convert **integer to hexadecimal string**.
5. oct() : This function is to convert **integer to octal string**.

```
# using  ord(), hex(), oct()
# initializing  integer
s = '4’

# printing character converting to integer
c = ord(s)
print ("After converting character to integer : ",end="")
print (c)

# printing integer converting to hexadecimal string
c = hex(56)
print ("After converting 56 to hexadecimal string : ",end="")
print (c)
```

```
# printing integer converting to octal string
c = oct(56)
print ("After converting 56 to octal string : ",end="")
print (c)
```

Output –

```
After converting character to integer : 52
After converting 56 to hexadecimal string : 0x38
After converting 56 to octal string : 0o70
```

6. tuple() : This function is used to **convert to a tuple**.
7. set() : This function returns the **type after converting to set**.
8. list() : This function is used to convert **any data type to a list type**.

```
# using  tuple(), set(), list()
# initializing  string
s = ‘python'
```

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

# printing string converting to set
c = set(s)
print ("After converting string to set : ",end="")
print (c)

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

Output-

```
After converting string to tuple : ('g', 'e', 'e', 'k', 's’)
 After converting string to set : {'k', 'e', 's', 'g’}
 After converting string to list : ['g', 'e', 'e', 'k', 's']
```

9. dict() : This function is used to **convert a tuple of order (key,value) into a dictionary**.
10. str() : Used to **convert integer into a string.**
11. complex(real,imag) : This function **converts real numbers to complex(real,imag) number.**

```
# using  dict(), complex(), str()
 # initializing  integers
a = 1
b = 2

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

# printing integer converting to complex number
c = complex(1,2)
print ("After converting integer to complex number : ",end="")
print (c)

# printing integer converting to string
c = str(a)
print ("After converting integer to string : ",end="")
print (c)
```

```
# printing tuple converting to expression dictionary
c = dict(tup)
print ("After converting tuple to dictionary : ",end="")
print (c)
```

Output-

```
After converting integer to complex number : (1+2j)
After converting integer to string : 1
After converting tuple to dictionary : {'a': 1, 'f': 2, 'g': 3}
```

12. chr(number): This function **converts number to its corresponding ASCII character.**

```
# Convert ASCII value to characters
a = chr(76)
b = chr(77)
print(a)
print(b)
```

Output –

```
L
M
```

Input/Output

Taking input in Python

Developers often have a need to interact with users, either to get data or to provide some sort of result. Most programs today use a dialog box as a way of asking the user to provide some type of input. While Python provides us with two inbuilt functions to read the input from the keyboard.

1. input (prompt)
2. raw_input (prompt)

input( ) : This function first takes the input from the user and convert it into string. Type of the returned object always will be <type ‘str’>. It does not evaluate the expression it just return the complete statement as String. For

example –

```
# a use of input()

val = input("Enter your value: ")
print(val)
```

How the input function works in Python :

When input() function executes program flow will be stopped until the user has given an input.
The text or message display on the output screen to ask a user to enter input value is optional i.e. the prompt, will be printed on the screen is optional.
Whatever you enter as input, input function convert it into a string. if you enter an integer value still input() function convert it into a string. You need to explicitly convert it into an integer in your code using typecasting.

```
# Program to check input
# type in Python

num = input ("Enter number :")
print(num)
name1 = input("Enter name : ")
print(name1)

# Printing type of input value
print ("type of number", type(num))
print ("type of name", type(name1))
```

raw_input ( ) : This function works in older version (like Python 2.x). This function takes exactly what is typed from the keyboard, convert it to string and then return it to the variable in which we want to store.

For example –

```
g = raw_input("Enter your name : ")
print g
```

Output - Enter your name :

Here, ***g*** is a variable which will get the string value, typed by user during the execution of program. Typing of data for the raw_input() function is terminated by enter key. We can use raw_input() to enter numeric data also. In that case we use typecasting.For more details on typecasting refer this.

Taking input from console in Python

What is Console in Python? Console (also called Shell) is basically a command line interpreter that takes input from the user i.e one command at a time and interprets it. If it is error free then it runs the command and gives required output otherwise shows the error message.

Taking multiple inputs from user in Python

The developer often wants a user to enter multiple values or inputs in one line. In C++/C user can take multiple inputs in one line using scanf but in Python user can take multiple values or inputs in one line by two methods.

- Using split() method
- Using List comprehension

Using split() method :

This function helps in getting multiple inputs from users. It breaks the given input by the specified separator. If a separator is not provided then any white space is a separator. Generally, users use a split() method to split a Python string but one can use it in taking multiple inputs.

Syntax : input().split(separator, maxsplit)

```
# Python program showing how to multiple input using split

# taking two inputs at a time
x, y = input("Enter two values: ").split()
print("Number of boys: ", x)
print("Number of girls: ", y)
print()

# taking three inputs at a time
x, y, z = input("Enter three values: ").split()
print("Total number of students: ", x)
print("Number of boys is : ", y)
print("Number of girls is : ", z)
print()

# taking two inputs at a time
a, b = input("Enter two values: ").split()
print("First number is {} and second number is {}".format(a, b))
print()
```

```
# taking multiple inputs at a time and type casting using list() function
x = list(map(int, input("Enter multiple values: ").split()))
print("List of students: ", x)
```

- **<u>Using List comprehension</u> :**
  List comprehension is an elegant way to define and create list in Python. We can create lists just like mathematical statements in one line only. It is also used in getting multiple inputs from a user.

```
# Python program showing how to take multiple input using List comprehension

# taking two input at a time
x, y = [int(x) for x in input("Enter two values: ").split()]
print("First Number is: ", x)
print("Second Number is: ", y)
print()

# taking three input at a time
x, y, z = [int(x) for x in input("Enter three values: ").split()]
print("First Number is: ", x)
print("Second Number is: ", y)
print("Third Number is: ", z)
print()
```

```
# taking two inputs at a time
x, y = [int(x) for x in input("Enter two values: ").split()]
print("First number is {} and second number is {}".format(x, y))
print()

# taking multiple inputs at a time
x = [int(x) for x in input("Enter multiple values: ").split()]
print("Number of list is: ", x)
```

Note: The above examples take input separated by spaces. In case we wish to take input separated by comma (,), we can use the following:

```
# taking multiple inputs at a time separated by comma
x = [int(x) for x in input("Enter multiple value: ").split(",")]
print("Number of list is: ", x)
```

How to print without newline in Python?

Generally, people switching from C/C++ to Python wonder how to print two or more variables or statements without going into a new line in python. Since the python print() function by default ends with a newline. Python has a predefined format if you use print(a_variable) then it will **go to the next line automatically.**

Ex -

```
print(”python")
print(”hellopython")
```

But sometimes it may happen that we don’t want to go to the next line but want to print on the same line. So what we can do?

Ex -

Input : print(”python") print(”hellopython")
Output : python hellopython
Input : a = [1, 2, 3, 4]
Output : 1 2 3 4

The solution discussed here is totally dependent on the python version you are using.

Print without newline in Python 2.x

```
# Python 2 code for printing on the same line printing python and hellopython in the same line

print(”python"),
print(”hellopython")

# array
a = [1, 2, 3, 4]

# printing a element in same
# line
for i in range(4):
    print(a[i]),
```

Output –

```
python hellopython
1 2 3 4
```

Print without newline in Python 3.x

```
# Python 3 code for printing on the same line printing
# python and hellopython in the same line

print(”hello", end =" ")
print(”hellopython")

# array
a = [1, 2, 3, 4]

# printing a element in same
# line
for i in range(4):
    print(a[i], end =" ")
```

Output –

```
python hellopython
1 2 3 4
```

Print without newline in Python 3.x without using for loop

```
# Print without newline in Python 3.x without using for loop

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

# using * symbol prints the list
# elements in a single line
print(*l)

#This code is contributed by anuragsingh1022
```

```
Output - 1 2 3 4 5 6
```

Python | Output using print() function

Python print() function prints the message to the screen or any other standard output device.
Syntax: print(value(s), sep= ‘ ‘, end = ‘\n’, file=file, flush=flush)
Parameters:

- **value(s) :** Any value, and as many as you like. Will be converted to string before printed.
- **sep=’separator’ :** (Optional) Specify how to separate the objects, if there is more than one.Default :’ ‘
- **end=’end’:** (Optional) Specify what to print at the end.Default : ‘\n’
- **file :** (Optional) An object with a write method. Default :sys.stdout
- **flush :** (Optional) A Boolean, specifying if the output is flushed (True) or buffered (False). Default: False
- **Returns:** It returns output to the screen.

Though it is not necessary to pass arguments in the print() function, it requires an empty parenthesis at the end that tells python to execute the function rather calling it by name. Now, let’s explore the optional arguments that can be used with the print() function.

String Literals

String literals in python’s print statement are primarily used to format or design how a specific string appears when printed using the print() function.

\n : This string literal is used to add a new blank line while printing a statement.
“” : An empty quote (“”) is used to print an empty line.

Ex-

print(” Python \n Python is the best language for programming .")

Output:

Python

Python is the best language for programming.

- **end= ” ” statement**

The end keyword is used to specify the content that is to be printed at the end of the execution of the print() function. By default, it is set to “\n”, which leads to the change of line after the execution of print() statement.

Example: Python print() without new line.

Ex –

```
# This line will automatically add a new line before the next print statement
print (" Python is the best language for programming")

# This print() function ends with "**" as set in the end argument.
print (" Python is the best language for programming ", end= "**")
print("Welcome to python")
```

Output –

```
Python is the best language for programming
Python is the best language for programming **Welcome to python
```

▪ flush Argument

The I/Os in python are generally buffered, meaning they are used in chunks. This is where flush comes in as it helps users to decide if they need the written content to be buffered or not. By default, it is set to false. If it is set to true, the output will be written as a sequence of characters one after the other. This process is slow simply because it is easier to write in chunks rather than writing one character at a time. To understand the use case of the flush argument in the print() function, let’s take an example.

Example:

Imagine you are building a countdown timer, which appends the remaining time to the same line every second. It would look something like below:

```
3>>>2>>>1>>>Start
```

The initial code for this would look something like below;

Ex-

```
import time
count_seconds = 3
for i in reversed(range(count_seconds + 1)):
        if i > 0:
                print(i, end='>>>’)
                time.sleep(1)
        else:
                print('Start')
```

So, the above code adds text without a trailing newline and then sleeps for one second after each text addition. At the end of the countdown, it prints Start and terminates the line. If you run the code as it is, it waits for 3 seconds and abruptly prints the entire text at once.

▪ Separator

The print() function can accept any number of positional arguments. These arguments can be separated from each other using a **“,” separator**. These are primarily used for formatting multiple statements in a single print() function.

Example:

```
b = "for"
print(”Python", b , " Language ")
```

Output – Python for Language

▪ file Argument

Contrary to popular belief, the print() function doesn’t convert the messages into text on the screen. These are done by lower-level layers of code, that can read data(message) in bytes. The print() function is an interface over these layers, that delegates the actual printing to a stream or **file-like object**. By default, the print() function is bound to *sys.stdout* through the file argument.

Example: Python print() to file

Ex -

```
import io
# declare a dummy file
dummy_file = io.StringIO()

# add message to the dummy file
print('Hello Python!!', file=dummy_file)

# get the value from dummy file
dummy_file.getvalue()
```

Output - 'Hello Python!!\n’

Example : Using print() function in Python

```
# Python 3.x program showing how to print data on a screen

# One object is passed
print(” ")
HelloPython
x = 5
# Two objects are passed
print("x =", x)
```

```
# code for disabling the softspace feature
print('G', 'F', 'G', sep='')

# using end argument
print("Python", end='@')
print(" HelloPython ")
```

Python Conditions(if-else)

Decision making is the most important aspect of almost all the programming languages. As the name implies, decision making allows us to run a particular block of code for a particular decision. Here, the decisions are made on the validity of the particular conditions. Condition checking is the backbone of decision making.

In python, decision making is performed by the following statements.

Python supports the usual logical conditions from mathematics:

Equals: a == b
Not Equals: a != b
Less than: a < b
Less than or equal to: a <= b
Greater than: a > b
Greater than or equal to: a >= b
These conditions can be used in several ways, most commonly in "if statements" and loops.

Statement	Description
If Statement	The if statement is used to test a specific condition. If the condition is true, a block of code (if-block) will be executed.
If - else Statement	The if-else statement is similar to if statement except the fact that, it also provides the block of the code for the false case of the condition to be checked. If the condition provided in the if statement is false, then the else statement will be executed.
Nested if Statement	Nested if statements enable us to use if ? else statement inside an outer if statement.

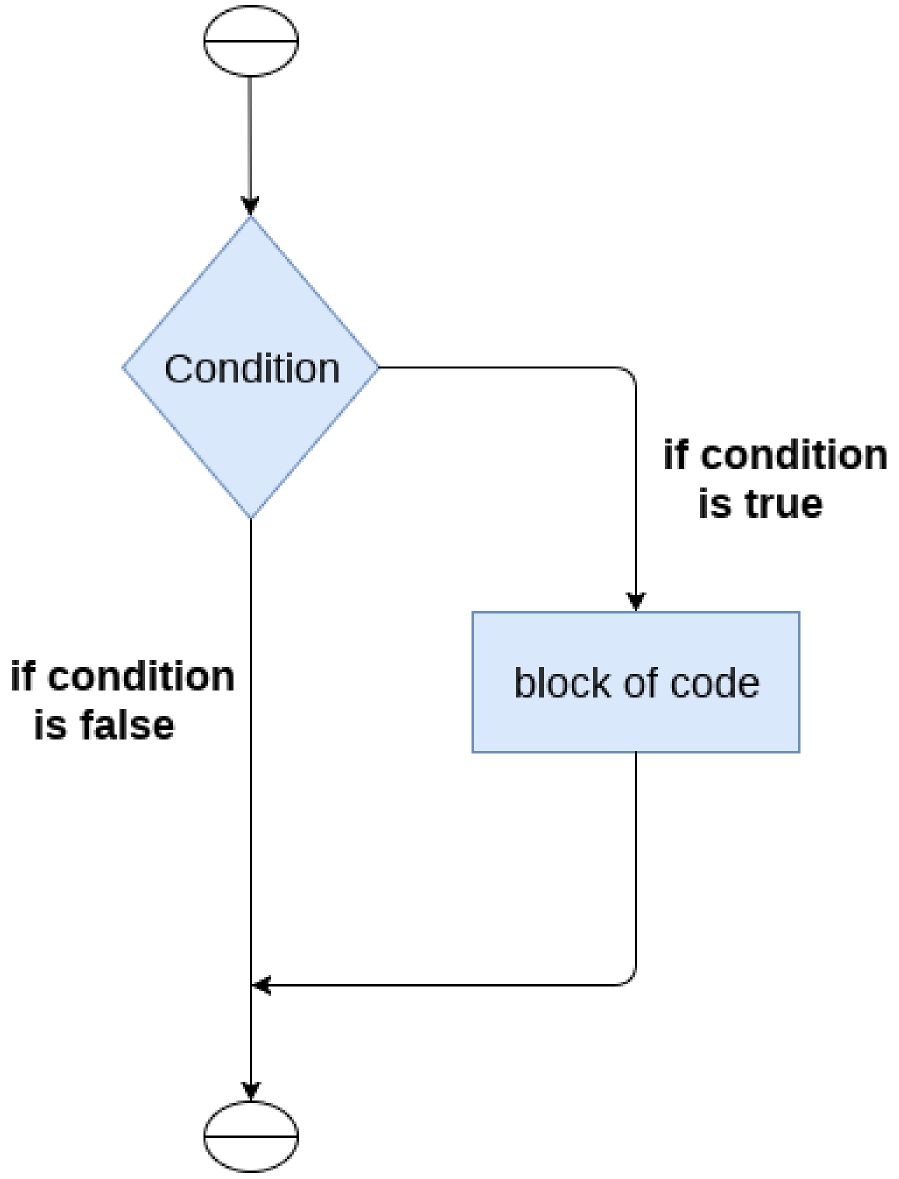
If statements

An "if statement" is written by using the if keyword.

The if statement is used to test a particular condition and if the condition is true, it executes a block of code known as if-block. The condition of if statement can be any valid logical expression which can be either evaluated to true or false.

The syntax of the if-statement is given below.

```
if expression:
    statement
```



Ex 1–

```
# If statement:
a = 33
b = 200
if b > a:
    print("b is greater than a")
```

In this example we use two variables, a and b, which are used as part of the if statement to test whether b is greater than a. As a is 33, and b is 200, we know that 200 is greater than 33, and so we print to screen that "b is greater than a".

Ex 2-

```
num = int(input("enter the number?"))
if num%2 == 0:
    print("Number is even")
```

Output:

```
enter the number?10
Number is even
```

Ex 3 : Program to 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");
```

Output –

```
Enter a? 100
Enter b? 120
Enter c? 130
c is largest
```

▪ Indentation

Python relies on indentation (whitespace at the beginning of a line) to define scope in the code. Other programming languages often use curly-brackets for this purpose.

Generally, four spaces are given to indent the statements which are a typical amount of indentation in python. Indentation is the most used part of the python language since it declares the block of code. All the statements of one block are intended at the same level indentation. We will see how the actual indentation takes place in decision making and other stuff in python.

Ex-

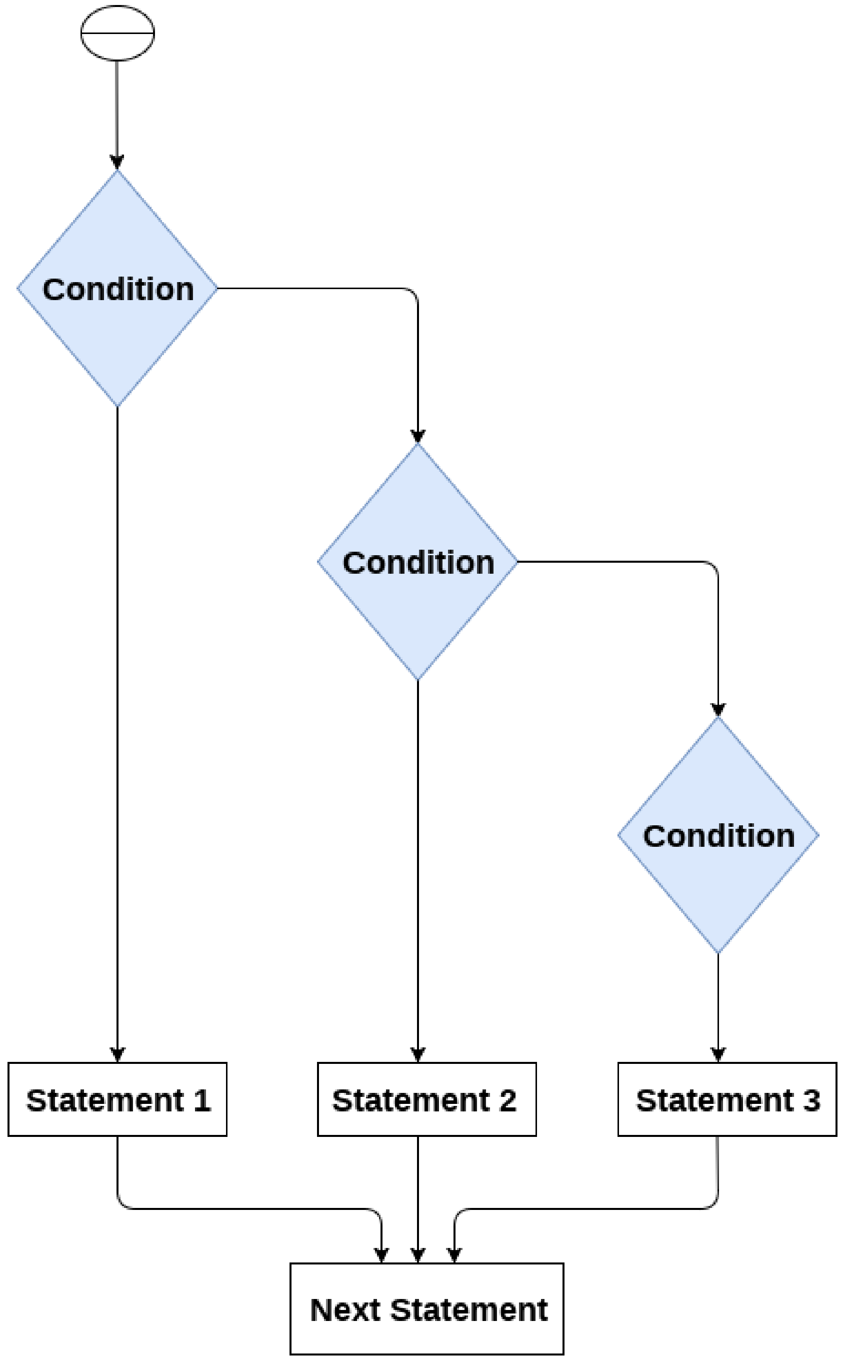
```
#If statement, without indentation (will raise an error):

a = 33
b = 200
if b > a:
print("b is greater than a") # you will get an error
```

The elif statement

The elif keyword is pythons' way of saying "if the previous conditions were not true, then try this condition".

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.



The syntax of the elif statement is given below.

```
if expression 1:
    # block of statements
elif expression 2:
    # block of statements
elif expression 3:
    # block of statements
else:
    # block of statements
```

Ex-

```
a = 33
b = 33
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
```

In this example a is equal to b, so the first condition is not true, but the elif condition is true, so we print to screen that "a and b are equal".

The if-else statement

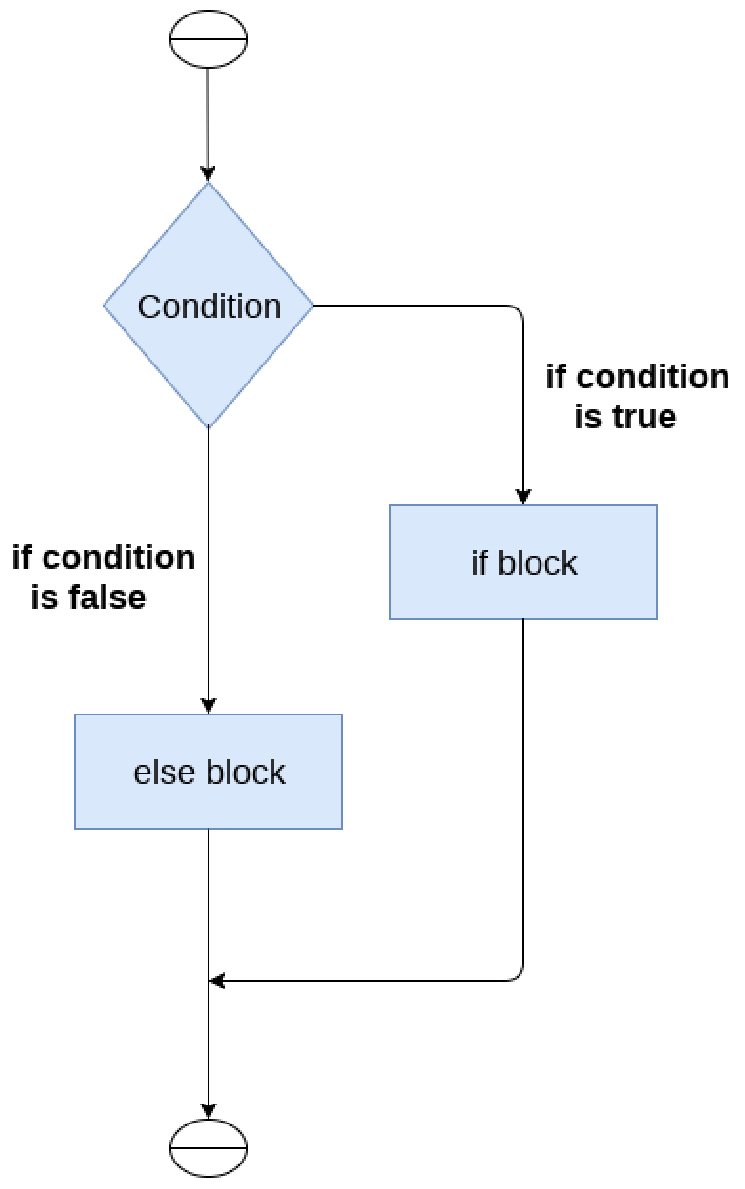
The else keyword catches anything which isn't caught by the preceding conditions.

The if-else statement provides an else block combined with the if statement which is executed in the false case of the condition.

If the condition is true, then the if-block is executed. Otherwise, the else-block is executed.

The syntax of the if-else statement is given below.

```
if condition:
    #block of statements
else:
    #another block of statements (else-block)
```



Ex – Program to check whether a number is even or not.

```
num = int(input("enter the number?"))
if num%2 == 0:
    print("Number is even...")
else:
    print("Number is odd...")
```

Output –

```
enter the number?10
Number is even
```

Ex- : 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 !!");
```

Output –

```
Enter your age? 90
You are eligible to vote !!
```

Ex -

```
a = 200
b = 33
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
else:
  print("a is greater than b")
```

In this example a is greater than b, so the first condition is not true, also the elif condition is not true, so we go to the else condition and print to screen that "a is greater than b".

You can also have an else without the elif.

Ex -

```
a = 200
b = 33
if b > a:
  print("b is greater than a")
else:
  print("b is not greater than a")
```

- Shorthand If

If you have only one statement to execute, you can put it on the same line as the if statement.

Ex- One line if statement:

```
if a > b: print("a is greater than b")
```

- Shorthand If ... Else

If you have only one statement to execute, one for if, and one for else, you can put it all on the same line:

Ex - One line if else statement:

```
a = 2
b = 330
print("A") if a > b else print("B")
```

This technique is known as **Ternary Operators**, or **Conditional Expressions**.

You can also have multiple else statements on the same line:

Ex - One line if else statement, with 3 conditions:

```
a,b = 330,330
print("A") if a > b else print("=") if a == b else print("B")
```

- And

The and keyword is a logical operator, and is used to combine conditional statements:

Ex- Test if a is greater than b, AND if c is greater than a:

```
a = 200
b = 33
c = 500
if a > b and c > a:
  print("Both conditions are True")
```

- Or

The or keyword is a logical operator, and is used to combine conditional statements:

Ex- Test if a is greater than b, OR if a is greater than c:

```
a = 200
b = 33
c = 500
if a > b or a > c:
  print("At least one of the conditions is True")
```

- Nested If

You can have if statements inside if statements, this is called *nested* if statements.

Ex-

```
x = 41
if x > 10:
  print("Above ten,")
  if x > 20:
    print("and also above 20!")
  else:
    print("but not above 20.")
```

- The pass Statement

if statements cannot be empty, but if you for some reason have an if statement with no content, put in the pass statement to avoid getting an error.

Ex -

```
a = 33
b = 200
if b > a:
  pass
```

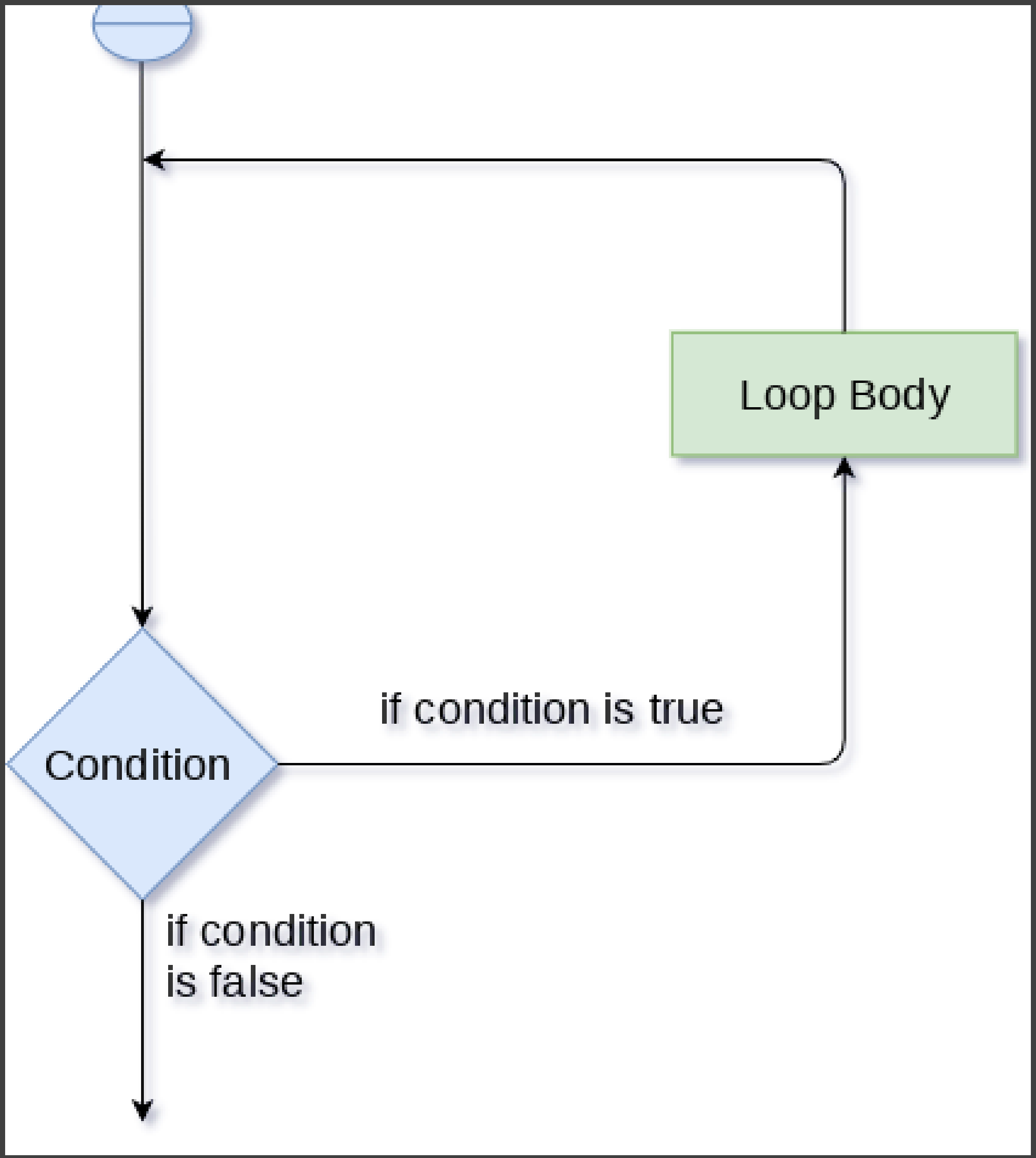
Control Flow

The flow of the programs written in any programming language is sequential by default. Sometimes we may need to alter the flow of the program. The execution of a specific code may need to be repeated several numbers of times.

For this purpose, The programming languages provide various types of loops which are capable of repeating some specific code several numbers of times. Consider the following diagram to understand the working of a loop statement.

Why we use loops in python?

The looping simplifies the complex problems into the easy ones. It enables us to alter the flow of the program so that instead of writing the same code again and again, we can repeat the same code for a finite number of times. For example, if we need to print the first 10 natural numbers then, instead of using the print statement 10 times, we can print inside a loop which runs up to 10 iterations.



Advantages of loops

There are the following advantages of loops in Python.

- It provides code re-usability.
- Using loops, we do not need to write the same code again and again.
- Using loops, we can traverse over the elements of data structures (array or linked lists).

There are the following loop statements in Python.

Loop Statement	Description
for loop	The for loop is used in the case where we need to execute some part of the code until the given condition is satisfied. The for loop is also called as a per-tested loop. It is better to use for loop if the number of iteration is known in advance.
while loop	The while loop is to be used in the scenario where we don't know the number of iterations in advance. The block of statements is executed in the while loop until the condition specified in the while loop is satisfied. It is also called a pre-tested loop.
do-while loop	The do-while loop continues until a given condition satisfies. It is also called post tested loop. It is used when it is necessary to execute the loop at least once (mostly menu driven programs).

Python for loop

The for **loop in Python** is used to iterate the statements or a part of the program several times. It is frequently used to traverse the data structures like list, tuple, or dictionary.

The syntax of for loop in python is given below.

```
for iterating_var in sequence:
   statement(s)
```

For loop Using Sequence

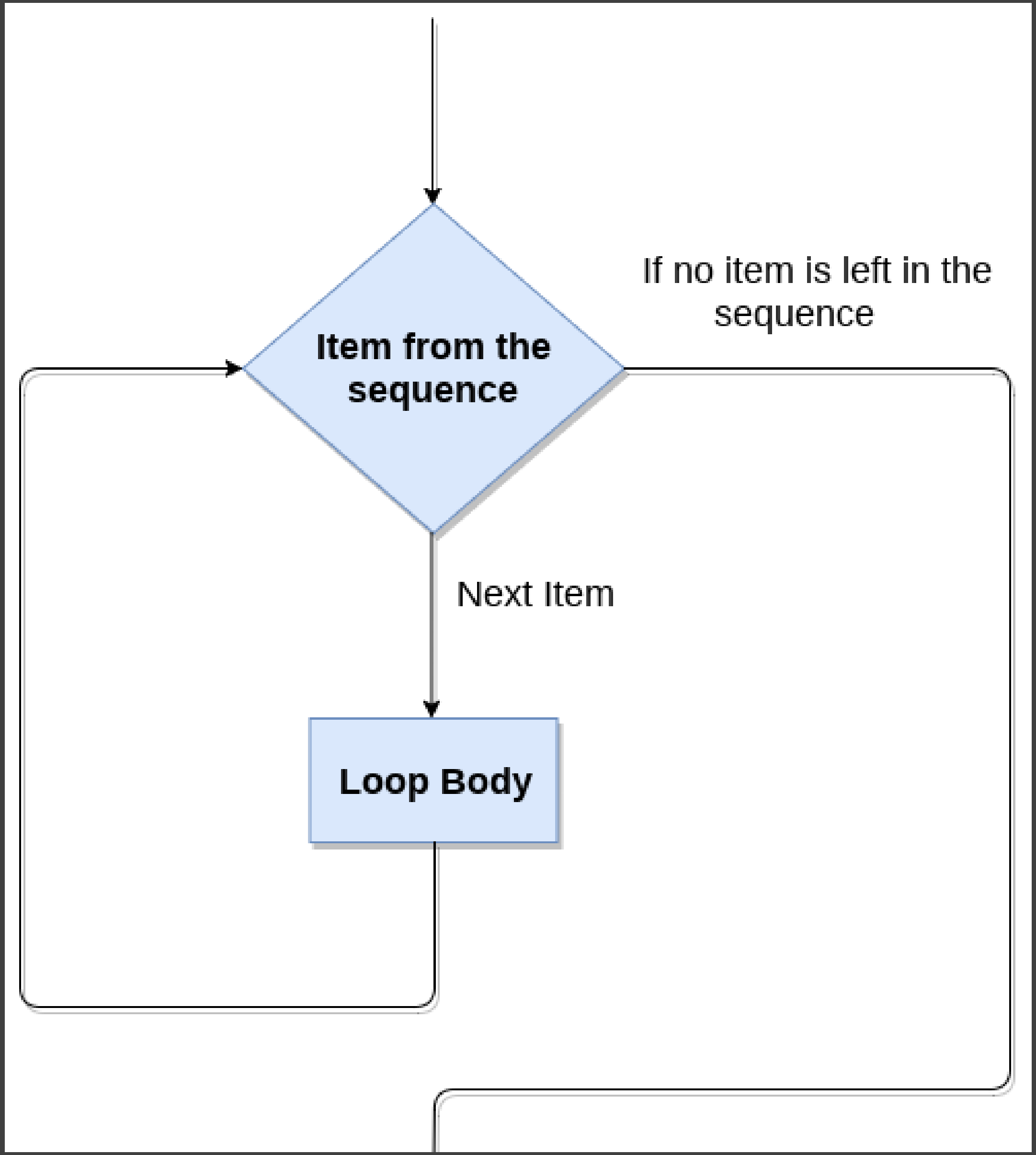
Ex1: Iterating string using for loop

```
str = "Python"
for i in str:
    print(i)
```

Output –

```
P
Y
T
H
O
N
```

Flow Diagram



Ex2: Program to print the table of the given number.

```
list = [1,2,3,4,5,6,7,8,9,10]
n = 5
for i in list:
    c = n*i
    print(c)
```

Output -

```
5
10
15
20
25
30
35
40
45
50
```

Ex3: Program to print the sum of the given list.

```
list = [10,30,23,43,65,12]
sum = 0
for i in list:
    sum = sum+i
print("The sum is:",sum)
```

Output –

```
The sum is: 183
```

For loop Using range() function

The range() function

The **range()** function is used to generate the sequence of the numbers. If we pass the range(10), it will generate the numbers from 0 to 9. The syntax of the range() function is given below.

Syntax: range(start,stop,step size)

- The start represents the beginning of the iteration.
- The stop represents that the loop will iterate till stop-1. The **range(1,5)** will generate numbers 1 to 4 iterations. It is optional.
- The step size is used to skip the specific numbers from the iteration. It is optional to use. By default, the step size is 1. It is optional.

Ex-1: Program to print numbers in sequence.

```
for i in range(10):
    print(i,end = ' ')
```

Output –

```
0 1 2 3 4 5 6 7 8 9
```

Ex - 2: Program to print table of given number.

```
n = int(input("Enter the number "))
for i in range(1,11):
    c = n*i
    print(n,"*",i,"=",c)
```

Output -

```
Enter the number
 10 10 * 1 = 10
 10 * 2 = 20
 10 * 3 = 30
 10 * 4 = 40
 10 * 5 = 50
 10 * 6 = 60
 10 * 7 = 70
 10 * 8 = 80
 10 * 9 = 90
 10 * 10 = 100
```

Ex - 3: Program to print even number using step size in range().

```
n = int(input("Enter the number "))
for i in range(2,n,2):
    print(i)
```

Output –

```
Enter the number 20
 2
 4
 6
 8
 10
 12
 14
 16
 18
```

We can also use the **range()** function with sequence of numbers. The **len()** function is combined with range() function which iterate through a sequence using indexing. Consider the following example.

```
list = ['Peter','Joseph','Ricky','Devansh']
for i in range(len(list)):
    print("Hello",list[i])
```

Output –

```
Hello Peter
Hello Joseph
Hello Ricky
Hello Devansh
```

Nested for loop in python

Python allows us to nest any number of for loops inside a **for** loop. The inner loop is executed n number of times for every iteration of the outer loop. The syntax is given below.

Syntax -

```
for iterating_var1 in sequence:  #outer loop
    for iterating_var2 in sequence:  #inner loop
        #block of statements
#Other statements
```

Ex-1: Nested for loop

```
# User input for number of rows
rows = int(input("Enter the rows:"))
# Outer loop will print number of rows
for i in range(0,rows+1):
# Inner loop will print number of Astrisk
    for j in range(i):
        print("*",end = '')
    print()
```

```
Enter the rows:3
*
**
***
```

Example-2: Program to number pyramid.

```
rows = int(input("Enter the rows"))
for i in range(0,rows+1):
    for j in range(i):
        print(i,end = '')
    print()
```

```
1
22
333
4444
55555
```

Using else statement with for loop

Unlike other languages like C, C++, or Java, Python allows us to use the else statement with the for loop which can be executed only when all the iterations are exhausted. Here, we must notice that if the loop contains any of the break statement, then the else statement will not be executed.

```
for i in range(0,5):
    print(i)
else:
    print("for loop completely exhausted, since there is no break.")
```

Output-

```
0
1
2
3
4
for loop completely exhausted, since there is no break.
```

Ex-2

```
for i in range(0,5):
    print(i)
    break;
else:print("for loop is exhausted");
print("The loop is broken due to break statement...came out of the loop")
```

In the above example, the loop is broken due to the break statement; therefore, the else statement will not be executed. The statement present immediate next to else block will be executed.

Output: 0

The loop is broken due to the break statement...came out of the loop. We will learn more about the break statement in next tutorial.

Python While loop

The Python while loop allows a part of the code to be executed until the given condition returns false. It is also known as a pre-tested loop.

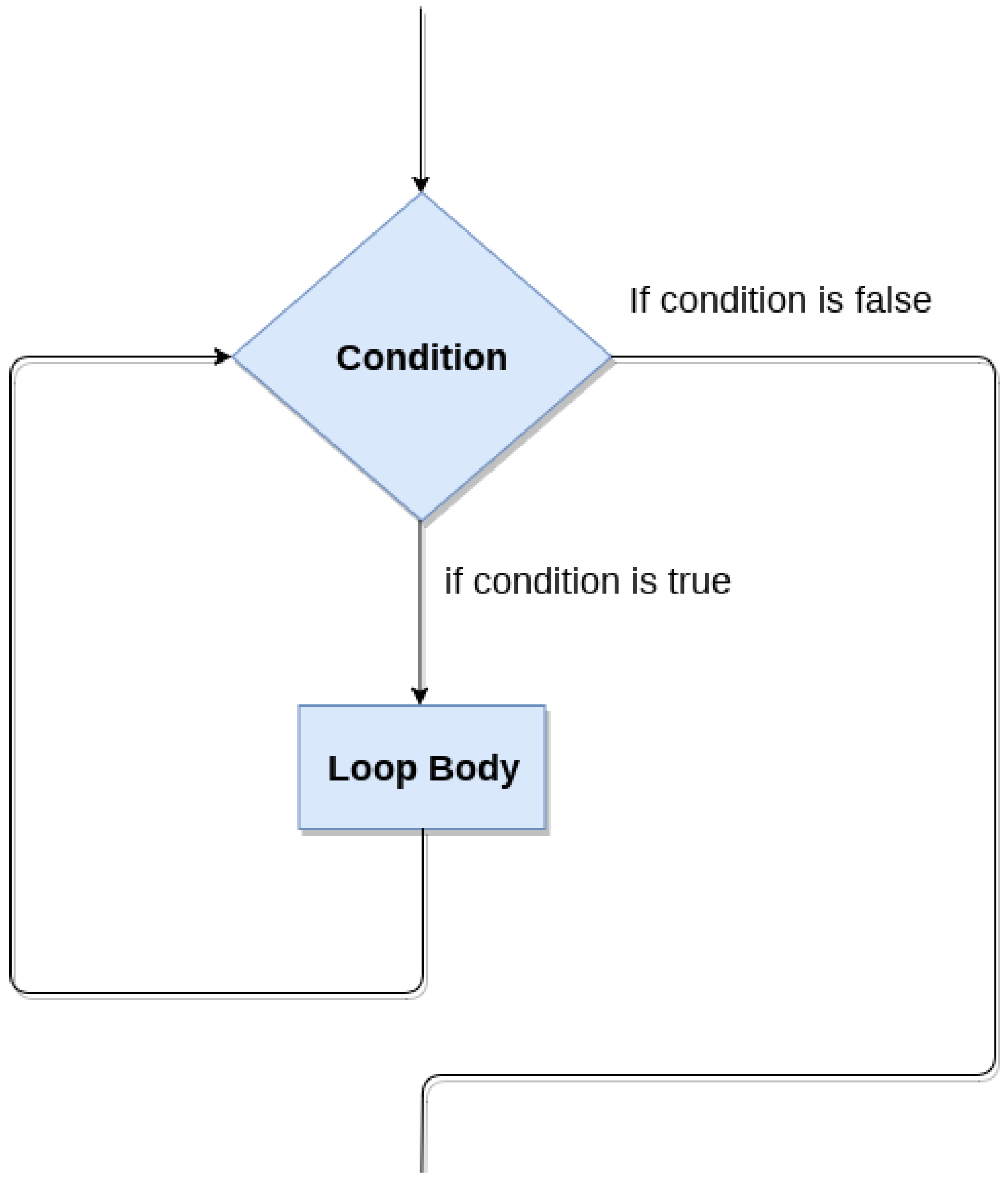
It can be viewed as a repeating if statement. When we don't know the number of iterations then the while loop is most effective to use.

The syntax is given below.

```
while expression:
    statements
```

Here, the statements can be a single statement or a group of statements. The expression should be any valid Python expression resulting in true or false. The true is any non-zero value and false is 0.

While loop Flowchart



Loop Control Statements

We can change the normal sequence of **while** loop's execution using the loop control statement. When the while loop's execution is completed, all automatic objects defined in that scope are demolished. Python offers the following control statement to use within the while loop.

1. Continue Statement - When the continue statement is encountered, the control transfer to the beginning of the loop. Let's understand the following example.

2. Break Statement - When the break statement is encountered, it brings control out of the loop

3. Pass Statement - The pass statement is used to declare the empty loop. It is also used to define empty class, function, and control statement. Let's understand the following example.

Ex-1: Continue Statement

```
# prints all letters except 'a' and 't'
i = 0
str1 = 'javatpoint'
while i < len(str1):
    if str1[i] == 'a' or str1[i] == 't':
        i += 1
        continue
    print('Current Letter :', a[i])
    i += 1
```

Output:

```
Current Letter : j
Current Letter : v
Current Letter : p
Current Letter : o
Current Letter : i
Current Letter : n
```

Ex-2: Break Statement

```
# The control transfer is transfered
# when break statement soon it sees t

i = 0
str1 = 'javatpoint'

while i < len(str1):
    if str1[i] == 't':
        i += 1
        break
    print('Current Letter :', str1[i])
    i += 1
```

Output:

```
Current Letter : j Current Letter : a Current Letter : v Current
Letter : a
```

Ex -3: Pass Statement

```
# An empty loop
str1 = 'javatpoint'
i = 0
while i < len(str1):
    i += 1
    pass
print('Value of i :', i)
```

Output:

```
Value of i : 10
```

Example-1: Program to print 1 to 10 using while loop

```
i=1
#The while loop will iterate until condition becomes false.
While(i<=10):
    print(i)
    i=i+1
```

Output –

```
1
2
3
4
5
6
7
8
9
10
```

Example -2: Program to print table of given numbers.

```
i=1
number=0
b=9
number = int(input("Enter the number:"))
while i<=10:
    print("%d X %d = %d \n"%(number,i,number*i))
    i = i+1
```

Output -

```
Enter the number:10
 10 X 1 = 10
 10 X 2 = 20
 10 X 3 = 30
 10 X 4 = 40
 10 X 5 = 50
 10 X 6 = 60
 10 X 7 = 70
 10 X 8 = 80
 10 X 9 = 90
 10 X 10 = 100
```

Infinite while loop

If the condition is given in the while loop never becomes false, then the while loop will never terminate, and it turns into the **infinite while loop.**

Any **non-zero** value in the while loop indicates an **always-true** condition, whereas zero indicates the always-false condition. This type of approach is useful if we want our program to run continuously in the loop without any disturbance.

Ex-1:

```
while (1):
    print("Hi! we are inside the infinite while loop")
```

Output:

```
Hi! we are inside the infinite while loop Hi! we are inside
the infinite while loop
```

Ex-2:

```
var = 1
while(var != 2):
    i = int(input("Enter the number:"))
    print("Entered value is %d"%(i))
```

Output -

```
Enter the number:10
Entered value is 10
Enter the number:10
Entered value is 10
Enter the number:10
Entered value is 10
Infinite time
```

Using else with while loop

Python allows us to use the else statement with the while loop also. The else block is executed when the condition given in the while statement becomes false. Like for loop, if the while loop is broken using break statement, then the else block will not be executed, and the statement present after else block will be executed. The else statement is optional to use with the while loop. Consider the following example.

Ex-1:

```
i=1
while(i<=5):
   print(i)
   i=i+1
else:
   print("The while loop exhausted")
```

Ex-2:

```
i=1
while(i<=5):
   print(i)
   i=i+1
   if(i==3):
      break
else:
   print("The while loop exhausted")
```

Output:

```
1 2
```

In the above code, when the break statement encountered, then while loop stopped its execution and skipped the else statement.

Ex-3: Program to print Fibonacci numbers to given limit.

```
terms = int(input("Enter the terms "))
# first two intial terms
a = 0
b = 1
count = 0

# check if the number of terms is Zero or negative
if (terms <= 0):
   print("Please enter a valid integer")
elif (terms == 1):
   print("Fibonacci sequence upto",limit,":")
   print(a)
else:
   print("Fibonacci sequence:")
   while (count < terms) :
       print(a, end = ' ')
       c = a + b
       # updateing values
       a = b
       b = c
   count += 1
```

Output:

```
Enter the terms 10
Fibonacci sequence:
0 1 1 2 3 5 8 13 21 34
```

Python break statement

The break is a keyword in python which is used to bring the program control out of the loop. The break statement breaks the loops one by one, i.e., in the case of nested loops, it breaks the inner loop first and then proceeds to outer loops. In other words, we can say that break is used to abort the current execution of the program and the control goes to the next line after the loop.
The break is commonly used in the cases where we need to break the loop for a given condition.
The syntax of the break is given below.

```
#loop statements
break;
```

Ex-1:

```
list =[1,2,3,4]
count = 1;
for i in list:
    if i == 4:
        print("item matched")
        count = count + 1;
        break
print("found at",count,"location");
```

Output –

```
item matched
found at 2 location
```

Ex-2:

```
str = "python"
for i in str:
    if i == 'o':
        break
    print(i);
```

Output:

```
p
y
t
h
```

Ex-3: break statement with while loop

```
i = 0;
while 1:
    print(i," ",end=""),
    i=i+1;
    if i == 10:
        break;
print("came out of while loop");
```

Output:

```
0 1 2 3 4 5 6 7 8 9
came out of while loop
```

Ex-3:

```
n=2
while 1:
    i=1;
    while i<=10:
        print("%d X %d = %d\n"%(n,i,n*i));
        i = i+1;
    choice = int(input("Do you want to continue printing the tabl
e, press 0 for no?"))
    if choice == 0:
        break;
    n=n+1
```

Output -

```
2 X 1 = 2 2 X 2 = 4 2 X 3 = 6 2 X 4 = 8 2 X 5 = 10 2 X 6 = 12 2 X 7 =
14 2 X 8 = 16 2 X 9 = 18 2 X 10 = 20

Do you want to continue printing the table, press 0 for no?1
3 X 1 = 3 3 X 2 = 6 3 X 3 = 9 3 X 4 = 12 3 X 5 = 15 3 X 6 = 18 3 X 7
= 21 3 X 8 = 24 3 X 9 = 27 3 X 10 = 30

Do you want to continue printing the table, press 0 for no?0
```

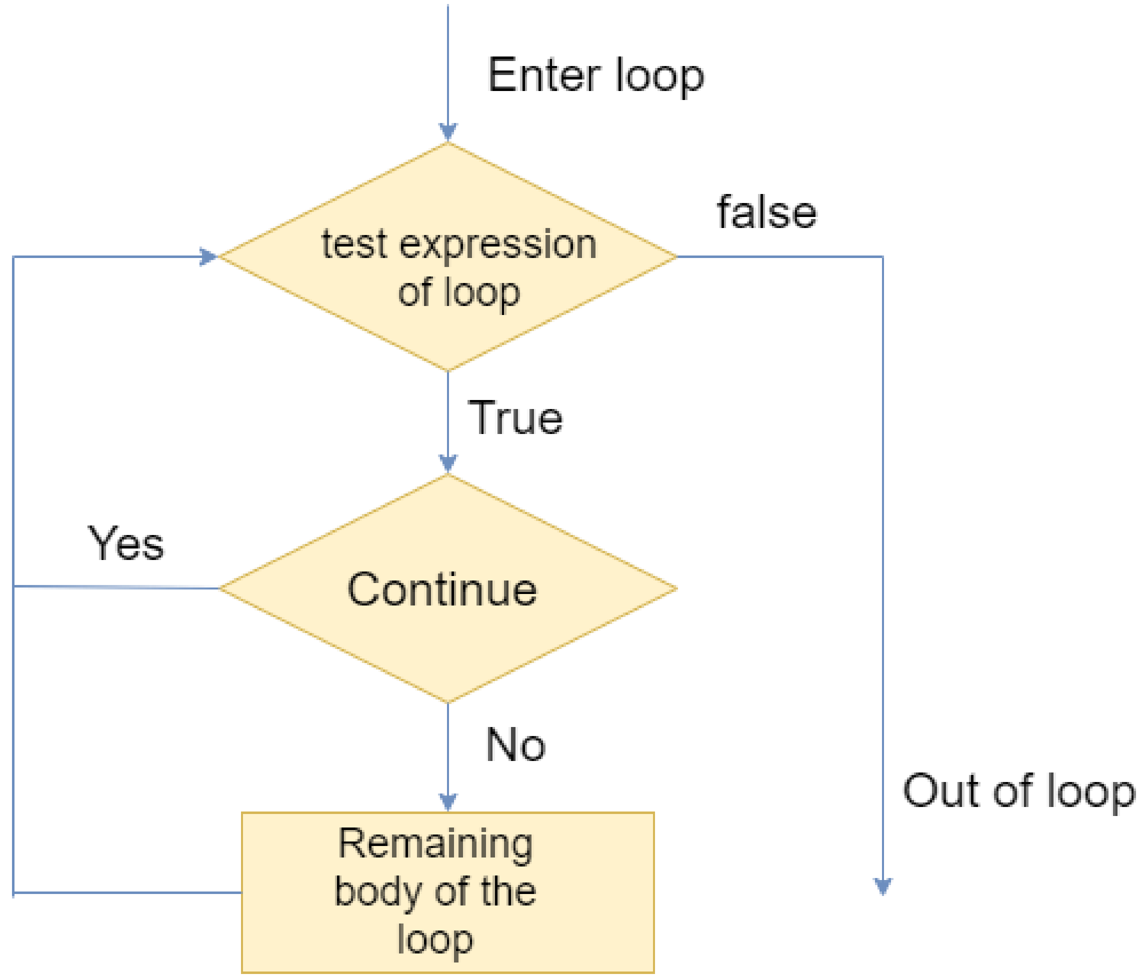
Python Continue Statement

The continue statement in Python is used to bring the program control to the beginning of the loop. The continue statement skips the remaining lines of code inside the loop and start with the next iteration. It is mainly used for a particular condition inside the loop so that we can skip some specific code for a particular condition. The continue statement in Python is used to bring the program control to the beginning of the loop. The continue statement skips the remaining lines of code inside the loop and start with the next iteration. It is mainly used for a particular condition inside the loop so that we can skip some specific code for a particular condition.

Syntax-

```
#loop statements
continue
#the code to be skipped
```

Flow Diagram



Consider the following examples.

Ex-1:

```
i = 0
while(i < 10):
   i = i+1
   if(i == 5):
      continue
   print(i)
```

Output:

```
1
2 3 4 6 7 8 9 10
```

Observe the output of above code, the value 5 is skipped because we have provided the **if condition** using with **continue statement** in while loop. When it matched with the given condition then control transferred to the beginning of the while loop and it skipped the value 5 from the code.

Ex-2:

```
str = "JavaTpoint"
for i in str:
    if(i == 'T'):
        continue
    print(i)
```

Output:

```
J
a
v
a
p
o
i
n
t
```

Pass Statement

The pass statement is a null operation since nothing happens when it is executed. It is used in the cases where a statement is syntactically needed but we don't want to use any executable statement at its place.
For example, it can be used while overriding a parent class method in the subclass but don't want to give its specific implementation in the subclass.

In Python, the pass keyword is used to execute nothing; it means, when we don't want to execute code, the pass can be used to execute empty. It is the same as the name refers to. It just makes the control to pass by without executing any code. If we want to bypass any code pass statement can be used.
It is beneficial when a statement is required syntactically, but we want we don't want to execute or execute it later. The difference between the comments and pass is that comments are entirely ignored by the Python interpreter, where the pass statement is not ignored.

Example - Pass statement

```
# pass is just a placeholder for
# we will adde functionality later.
values = {'P', 'y', 't', 'h','o','n'}
for val in values:
    pass
```

Pass is also used where the code will be written somewhere but not yet written in the program file. Consider the following example.

Ex –

```
list = [1,2,3,4,5]
flag = 0
for i in list:
    print("Current element:",i,end=" ");
    if i==3:
        pass
        print("\nWe are inside pass block\n");
        flag = 1
    if flag==1:
        print("\nCame out of pass\n");
        flag=0
```

Output-

```
Current element: 1
Current element: 2
Current element: 3
We are inside pass block Came out of pass
Current element: 4 Current element: 5
```

Ex - 2:

```
for i in [1,2,3,4,5]:
    if(i==4):
        pass
        print("This is pass block",i)
    print(i)
```

Output:

```
1
2
3
This is pass block 4
4
5
```

We can create empty class or function using the pass statement.

```
# Empty Function
def function_name(args):
    pass
#Empty Class
class Python:
    pass
```

Python Lists Methods

The list is a most versatile datatype available in Python which can be written as a list of comma-separated values (items) between square brackets. Important thing about a list is that items in a list need not be of the same type.

```
list1 = ['physics', 'chemistry', 1997, 2000]
```

Updating Lists

You can update single or multiple elements of lists by giving the slice on the left-hand side of the assignment operator, and you can add to elements in a list with the append() method.

Ex-

```
list = ['physics', 'chemistry', 1997, 2000];
print "Value available at index 2 : "
print list[2]
list[2] = 2001;
print "New value available at index 2 : "
print list[2]
```

Output -

```
Value available at index 2 :
1997
New value available at index 2 :
2001
```

Sr. No.	Function with Description
1	cmp(list1, list2)Compares elements of both lists.
2	len(list)Gives the total length of the list.
3	max(list)Returns item from the list with max value.
4	min(list)Returns item from the list with min value.
5	list(seq)Converts a tuple into list.

Click to add text

Python includes following list methods

Sr.No.	Methods with Description
1	list.append(obj)Appends object obj to list
2	list.count(obj)Returns count of how many times obj occurs in list
3	list.extend(seq)Appends the contents of seq to list
4	list.index(obj)Returns the lowest index in list that obj appears
5	list.insert(index, obj)Inserts object obj into list at offset index
6	list.pop(obj=list[-1])Removes and returns last object or obj from list
7	list.remove(obj)Removes object obj from list
8	list.reverse()Reverses objects of list in place
9	list.sort([func])Sorts objects of list, use compare func if given

Tuple Methods Concept

A tuple is a collection of objects which ordered and immutable. Tuples are sequences, just like lists. The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use parentheses, whereas lists use square brackets.
Creating a tuple is as simple as putting different comma-separated values. Optionally you can put these comma-separated values between parentheses also.

For example −

```
tup1 = ('physics', 'chemistry', 1997, 2000)
tup2 = (1, 2, 3, 4, 5 )
tup3 = "a", "b", "c", "d"
```

To write a tuple containing a single value you have to include a comma, even though there is only one value −

```
tup1 = (50,)
```

Updating Tuples

Tuples are immutable which means you cannot update or change the values of tuple elements. You are able to take portions of existing tuples to create new tuples as the following

Example demonstrates −

```
tup1 = (12, 34.56)
tup2 = ('abc', 'xyz')

# Following action is not valid for tuples
# tup1[0] = 100

# So let's create a new tuple as follows
tup3 = tup1 + tup2
print tup3;
```

When the above code is executed, it produces the following result −

```
(12, 34.56, 'abc', 'xyz')
```

Delete Tuple Elements

Removing individual tuple elements is not possible. There is, of course, nothing wrong with putting together another tuple with the undesired elements discarded.
To explicitly remove an entire tuple, just use the **del** statement.

For example −

```
tup = ('physics', 'chemistry', 1997, 2000)
print(tup)
del(tup)
print("After deleting tup : ")
print tup
```

This produces the following result.

Note an exception raised, this is because after **del tup** tuple does not exist any more −

```
('physics', 'chemistry', 1997, 2000)
After deleting tup :
Traceback (most recent call last):
File "test.py", line 9, in <module>

print tup;
NameError: name 'tup' is not defined
```

No Enclosing Delimiters

Any set of multiple objects, comma-separated, written without identifying symbols, i.e., brackets for lists, parentheses for tuples, etc., default to tuples, as indicated in these short examples −

Ex -

```
print('abc', -4.24e93, 18+6.6j, 'xyz')
x, y = 1, 2
print("Value of x , y : ", x,y)
```

When the above code is executed, it produces the following result −

```
abc -4.24e+93 (18+6.6j) xyz
Value of x , y : 1 2
```

Built-in Tuple Functions

Python includes the following tuple functions −

Sr.No.	Function with Description
1	cmp(tuple1, tuple2)Compares elements of both tuples.
2	len(tuple)Gives the total length of the tuple.
3	max(tuple)Returns item from the tuple with max value.
4	min(tuple)Returns item from the tuple with min value.
5	tuple(seq)Converts a list into tuple

Built-in Function	Description
all()	Returns true if all element are true or if tuple is empty
any()	return true if any element of the tuple is true. if tuple is empty, return false
enumerate()	Returns enumerate object of tuple
sum()	Sums up the numbers in the tuple
sorted()	input elements in the tuple and return a new sorted list
tuple()	Convert an iterable to a tuple.

String Methods Concept-

Strings are amongst the most popular types in Python. We can create them simply by enclosing characters in quotes. Python treats single quotes the same as double quotes. Creating strings is as simple as assigning a value to a variable.

Example −

```
var1 = 'Hello World!'
var2 = "Python Programming"
```

Accessing Values in Strings

Python does not support a character type; these are treated as strings of length one, thus also considered a substring. To access substrings, use the square brackets for slicing along with the index or indices to obtain your substring.

Example −

```
var1 = 'Hello World!'
var2 = "Python Programming"
print("var1[0]: ", var1[0])
print("var2[1:5]: ", var2[1:5])
```

When the above code is executed, it produces the following result −

```
var1[0]: H
var2[1:5]: ytho
```

Updating Strings

You can "update" an existing string by (re)assigning a variable to another string. The new value can be related to its previous value or to a completely different string altogether.

For example −

```
var1 = 'Hello World!'
Print("Updated String :- ", var1[:6] + 'Python')
```

When the above code is executed, it produces the following result −

```
Updated String :- Hello Python
```

Escape Characters

Following table is a list of escape or non-printable characters that can be represented with backslash notation.
An escape character gets interpreted; in a single quoted as well as double quoted strings.

Backslash notation	Hexadecimal character	Description
\a	0x07	Bell or alert
\b	0x08	Backspace
\cx		Control-x
\C-x		Control-x
\e	0x1b	Escape
\f	0x0c	Formfeed
\M-\C-x		Meta-Control-x
\n	0x0a	Newline
\nnn		Octal notation, where n is in the range 0.7
\r	0x0d	Carriage return
\s	0x20	Space
\t	0x09	Tab
\v	0x0b	Vertical tab
\x		Character x
\xnn		Hexadecimal notation, where n is in the range 0.9, a.f, or A.F

String Special Operators

Assume string variable **a** holds 'Hello' and variable **b** holds 'Python', then −

Operator	Description	Example
+	Concatenation - Adds values on either side of the operator	a + b will give HelloPython
*	Repetition - Creates new strings, concatenating multiple copies of the same string	a*2 will give -HelloHello
[]	Slice - Gives the character from the given index	a[1] will give e
[:]	Range Slice - Gives the characters from the given range	a[1:4] will give ell
in	Membership - Returns true if a character exists in the given string	H in a will give 1
not in	Membership - Returns true if a character does not exist in the given string	M not in a will give 1
r/R	Raw String - Suppresses actual meaning of Escape characters. The syntax for raw strings is exactly the same as for normal strings with the exception of the raw string operator, the letter "r," which precedes the quotation marks. The "r" can be lowercase (r) or uppercase (R) and must be placed immediately preceding the first quote mark.	print("r'\n' prints \n and print R'\n'prints \n")
%	Format - Performs String formatting	See at next section

String Formatting Operator

One of Python's coolest features is the string format operator %. This operator is unique to strings and makes up for the pack of having functions from C's printf() family.

Following is a simple example −

```
Print("My name is %s and weight is %d kg!" % ('Zara', 21))
```

When the above code is executed, it produces the following result −

```
My name is Zara and weight is 21 kg!
```

we can combine strings and numbers by using the format() method!
The format() method takes the passed arguments, formats them, and places them in the string where the placeholders {} are:

Example -
Use the format() method to insert numbers into strings:

```
age = 36
txt = "My name is John, and I am {}"
print(txt.format(age))
```

The format() method takes unlimited number of arguments, and are placed into the respective placeholders:

Example

```
quantity = 3
itemno = 567
price = 49.95
myorder = "I want {} pieces of item {} for {} dollars."
print(myorder.format(quantity, itemno, price))
```

You can use index numbers {0} to be sure the arguments are placed in the correct placeholders:

Example

```
quantity = 3
itemno = 567
price = 49.95
myorder = "I want to pay {2} dollars for {0} pieces of item {1}."
print(myorder.format(quantity, itemno, price))
```

Here is the list of complete set of symbols which can be used along with % −

Format Symbol	Conversion
%c	character
%s	string conversion via str() prior to formatting
%i	signed decimal integer
%d	signed decimal integer
%u	unsigned decimal integer
%o	octal integer
%x	hexadecimal integer (lowercase letters)
%X	hexadecimal integer (UPPERcase letters)
%e	exponential notation (with lowercase 'e')
%E	exponential notation (with UPPERcase 'E')
%f	floating point real number
%g	the shorter of %f and %e
%G	the shorter of %f and %E

Other supported symbols and functionality are listed in the following table −

Symbol	Functionality
*	argument specifies width or precision
-	left justification
+	display the sign
<sp>	leave a blank space before a positive number
#	add the octal leading zero ('0') or hexadecimal leading '0x' or '0X', depending on whether 'x' or 'X' were used.
0	pad from left with zeros (instead of spaces)
%	'%%' leaves you with a single literal '%'
(var)	mapping variable (dictionary arguments)
m.n.	m is the minimum total width and n is the number of digits to display after the decimal point (if appl.)

Unicode String

Normal strings in Python are stored internally as 8-bit ASCII, while Unicode strings are stored as 16-bit Unicode. This allows for a more varied set of characters, including special characters from most languages in the world.

Ex -

```
print(u'Hello, world!')
```

When the above code is executed, it produces the following result −

```
Hello, world!
```

String Methods

Python has a set of built-in methods that you can use on strings.
Note: All string methods returns new values. They do not change the original string.

Method	**Description**
capitalize()	Converts the first character to upper case Ex - txt = "36 is my age." x = txt.capitalize() print (x)
casefold()	The casefold() method returns a string where all the characters are lower case. This method is similar to the lower() method, but the casefold() method is stronger, more aggressive, meaning that it will convert more characters into lower case, and will find more matches when comparing two strings and both are converted using the casefold() method. Ex - txt = "Hello, And Welcome To My World!" x = txt.casefold() print(x)

<u>**CENTER()**</u>	**THE CENTER() METHOD WILL CENTER ALIGN THE STRING, USING A SPECIFIED CHARACTER (SPACE IS DEFAULT) AS THE FILL CHARACTER.** **TXT = "BANANA"** **X = TXT.CENTER(20)** **PRINT(X)**

<u>count()</u>	returns the number of times a specified value appears in the string.
	txt = "I love apples, apple are my favorite fruit"
	x = txt.count("apple", 10, 24)
	print(x)

<u>encode()</u>	The encode() method encodes the string, using the specified encoding. If no encoding is specified, UTF-8 will be used.
	Syntax - string.encode(encoding=encoding, errors=errors)
	txt = "My name is Ståle"
	x = txt.encode()
	print(x)

<u>endswith()</u>	Returns true if the string ends with the specified value
	txt = "Hello, welcome to my world."
	x = txt.endswith("my world.", 5, 11)
	print(x)

expandtabs()	**Sets the tab size to the specified number of whitespaces.**
	txt = "H\te\tl\tl\to"
	print(txt)
	print(txt.expandtabs())
	print(txt.expandtabs(2))

find()	Searches the string for a specified value and returns the position of where it was found
	string.find(value, start, end)
	txt = "Hello, welcome to my world."
	x = txt.find("e", 5, 10)
	print(x)

format()	Formats specified values in a string
	txt = "For only {price:.2f} dollars!"
	print(txt.format(price = 49))

format_map()	Formats specified values in a string
index()	The index() method finds the first occurrence of the specified value. The index() method raises an exception if the value is not found. The index() method is almost the same as the find() method, the only difference is that the find() method returns -1 if the value is not found. *string*.index(*value*, *start*, *end*) txt = "Hello, welcome to my world." x = txt.index("e", 5, 10) print(x)

isalnum()	The isalnum() method returns True if all the characters are alphanumeric, meaning alphabet letter (a-z) and numbers (0-9).
	Example of characters that are not alphanumeric: (space)!#%&? etc.
	string.isalnum()
	txt = "Company 12"
	x = txt.isalnum()
	print(x)

isalpha()	The isalpha() method returns True if all the characters are alphabet letters (a-z).
	Example of characters that are not alphabet letters: (space)!#%&? etc.
	txt = "Company10"
	x = txt.isalpha()
	print(x)

isdecimal()	The isdecimal() method returns True if all the characters are decimals (0-9).
	a = "\u0030" #unicode for 0
	b = "\u0047" #unicode for G
	print(a.isdecimal())
	print(b.isdecimal())

isdigit()	The isdigit() method returns True if all the characters are digits, otherwise False. Exponents, like ², are also considered to be a digit.
	a = "\u0030" #unicode for 0
	b = "\u00B2" #unicode for ²
	print(a.isdigit())
	print(b.isdigit())

isidentifier()	The isidentifier() method returns True if the string is a valid identifier, otherwise False.
	A string is considered a valid identifier if it only contains alphanumeric letters (a-z) and (0-9), or underscores (_). A valid identifier cannot start with a number, or contain any spaces.
	a = "MyFolder"
	b = "Demo002"
	c = "2bring"
	d = "my demo"
	print(a.isidentifier())
	print(b.isidentifier())
	print(c.isidentifier())
	print(d.isidentifier())

islower()	The islower() method returns True if all the characters are in lower case, otherwise False.
	Numbers, symbols and spaces are not checked, only alphabet characters.
	a = "Hello world!"
	b = "hello 123"
	c = "mynameisPeter"
	print(a.islower())
	print(b.islower())
	print(c.islower())

isnumeric()	**The isnumeric() method returns True if all the characters are numeric (0-9), otherwise False.** **Exponents, like ² and ¾ are also considered to be numeric values.** **"-1" and "1.5" are NOT considered numeric values, because all the characters in the string must be numeric, and the - and the . are not.** **a = "\u0030" #unicode for 0** **b = "\u00B2" #unicode for &sup2;** **c = "10km2"** **d = "-1"** **e = "1.5"** **print(a.isnumeric())** **print(b.isnumeric())** **print(c.isnumeric())** **print(d.isnumeric())** **print(e.isnumeric())**

isprintable()	The isprintable() method returns True if all the characters are printable, otherwise False.
	txt = "Hello!\nAre you #1?"
	x = txt.isprintable()
	print(x)

isspace()	**Returns True if all characters in the string are whitespaces.**
	txt = " s "
	x = txt.isspace()
	print(x)

istitle()	The istitle() method returns True if all words in a text start with a upper case letter, AND the rest of the word are lower case letters, otherwise False.
	a = "HELLO, AND WELCOME TO MY WORLD"
	b = "Hello"
	c = "22 Names"
	d = "This Is %'!?"
	print(a.istitle())
	print(b.istitle())
	print(c.istitle())
	print(d.istitle())

isupper()	The isupper() method returns True if all the characters are in upper case, otherwise False.
	a = "Hello World!"
	b = "hello 123"
	c = "MY NAME IS PETER"
	
	print(a.isupper())
	print(b.isupper())
	print(c.isupper())

join()	Joins the elements of an iterable to the end of the string
ljust()	Returns a left justified version of the string
lower()	Converts a string into lower case
lstrip()	Returns a left trim version of the string
maketrans()	Returns a translation table to be used in translations
partition()	Returns a tuple where the string is parted into three parts
replace()	Returns a string where a specified value is replaced with a specified value
rfind()	Searches the string for a specified value and returns the last position of where it was found
rindex()	Searches the string for a specified value and returns the last position of where it was found
rjust()	Returns a right justified version of the string
rpartition()	Returns a tuple where the string is parted into three parts
rsplit()	Splits the string at the specified separator, and returns a list

rstrip()	Returns a right trim version of the string
split()	Splits the string at the specified separator, and returns a list
splitlines()	Splits the string at line breaks and returns a list
startswith()	Returns true if the string starts with the specified value
strip()	Returns a trimmed version of the string
swapcase()	Swaps cases, lower case becomes upper case and vice versa
title()	Converts the first character of each word to upper case
translate()	Returns a translated string
upper()	Converts a string into upper case
zfill()	Fills the string with a specified number of 0 values at the beginning

Set

Sets are used to store multiple items in a single variable.
A set is a collection which is *unordered*, *unchangeable**, *unindexed*, mutable and has no duplicate elements.

```
# Creating a Set
set1 = set()
print("Initial blank Set: ")
print(set1)

# Creating a Set with the use of a String
set1 = set("Python")
print("\nSet with the use of String: ")
print(set1)

# (Using object to Store String)
String = 'Python'
set1 = set(String)
print("\nSet with the use of an Object: " )
print(set1)
```

Adding Elements to the set

The set add() method adds a given element to a set if the element is not present in the set.

Syntax -

```
set.add(elem)
```

The add() method doesn't add an element to the set if it's already present in it otherwise it will get added to the set.

Parameters:

add() takes single parameter **(elem)** which needs to be added in the set.

Returns:

The add() method doesn't return any value.

```
# set of letters
PEEK = {'P', 'e', 'k'}

# adding 's'
PEEK.add('s')
print('Letters are:', PEEK)
```

Ex -

```
PEEK = {6, 0, 4}
PEEK.add(1)
print('Letters are:', PEEK)

PEEK.add(0)
print('Letters are:', PEEK)
```

Output -

```
('Letters are:', set([0, 1, 4, 6]))
('Letters are:', set([0, 1, 4, 6]))
```

Ex -

```
s = {'p', 'e', 'e', 'k', 's'}
t = ('d', 'o')
# adding tuple t to set s.
s.add(t)
print(s)
```

Output -

```
{'k', 's', 'e', 'g', ('f', 'o')}
```

Using update() method

For the addition of two or more elements Update() method is used. The update() method accepts lists, strings, tuples as well as other sets as its arguments. In all of these cases, duplicate elements are avoided.

Syntax : set1.update(set2)
Here set1 is the set in which set2 will be added.
Parameters : Update() method takes only a single argument. The single argument can be a set, list, tuples or a dictionary. It automatically converts into a set and adds to the set.
Return value : This method adds set2 to set1 and returns nothing.

```
# using Update function
set1 = set([4, 5, (6, 7)])
set1.update([10, 11])
print("\nSet after Addition of elements using Update: ")
print(set1)
```

Output -

```
Set after Addition of elements using Update:
{4, 5, (6, 7), 10, 11}
```

Example 1: Working with Python set update list

```
list1 = [1, 2, 3]
list2 = [5, 6, 7]
list3 = [10, 11, 12]

# Lists converted to sets
set1 = set(list2)
set2 = set(list1)

# Update method
set1.update(set2)

# Print the updated set
print(set1)

# List is passed as an parameter which gets automatically converted to a set
set1.update(list3)
print(set1)
```

Output -

```
{1, 2, 3, 5, 6, 7}
{1, 2, 3, 5, 6, 7, 10, 11, 12}
```

Removing Elements to the set -

Using remove() method or discard() method:
Elements can be removed from the Set by using the built-in remove() function but a KeyError arises if the element doesn’t exist in the set. To remove elements from a set without KeyError, use discard(), if the element doesn’t exist in the set, it remains unchanged.

```
# Python program to demonstrate Deletion of elements in a Set

# Creating a Set
set1 = set([1, 2, 3, 4, 5, 6,
            7, 8, 9, 10, 11, 12])
print("Initial Set: ")
print(set1)

# Removing elements from Set using Remove() method
set1.remove(5)
set1.remove(6)
print("\nSet after Removal of two elements: ")
print(set1)
```

```
# Removing elements from Set using Discard() method
set1.discard(8)
set1.discard(9)
print("\nSet after Discarding two elements: ")
print(set1)

# Removing elements from Set using iterator.
for i in range(1, 5):
    set1.remove(i)
print("\nSet after Removing a range of elements: ")
print(set1)
```

Output -

```
Initial Set:
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}
Set after Removal of two elements:
{1, 2, 3, 4, 7, 8, 9, 10, 11, 12}
Set after Discarding two elements:
{1, 2, 3, 4, 7, 10, 11, 12}
Set after Removing a range of elements:
{7, 10, 11, 12}
```

Using pop() method:

Pop() function can also be used to remove and return an element from the set, but it removes only the last element of the set.

Note: *If the set is unordered then there's no such way to determine which element is popped by using the pop() function.*

Ex -

```
set1 = set([1, 2, 3, 4, 5, 6,
            7, 8, 9, 10, 11, 12])
print("Initial Set: ")
print(set1)

# Removing element from the Set using the pop() method
set1.pop()
print("\nSet after popping an element: ")
print(set1)
```

Output -

```
Initial Set:
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}
Set after popping an element:
{2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}
```

Using clear() method

To remove all the elements from the set, clear() function is used.

```
#Creating a set
set1 = set([1,2,3,4,5])
print("\n Initial set: ")
print(set1)

# Removing all the elements from Set using clear() method
set1.clear()
print("\nSet after clearing all the elements: ")
print(set1)
```

Output -

```
Initial set:
{1, 2, 3, 4, 5}

Set after clearing all the elements:
set()
```

Frozen sets in Python are immutable objects that only support methods and operators that produce a result without affecting the frozen set or sets to which they are applied. While elements of a set can be modified at any time, elements of the frozen set remain the same after creation. If no parameters are passed, it returns an empty frozenset.

```
# Creating a Set
String = ('G', 'e', 'e', 'k', 's', 'F', 'o', 'r')
Fset1 = frozenset(String)
print("The FrozenSet is: ")
print(Fset1)

# To print Empty Frozen Set No parameter is passed
print("\nEmpty FrozenSet: ")
print(frozenset())
```

Output -

```
The FrozenSet is:
frozenset({'o', 'G', 'e', 's', 'r', 'F', 'k'})

Empty FrozenSet:
frozenset()
```

Function	Description
add()	Adds an element to a set
remove()	Removes an element from a set. If the element is not present in the set, raise a KeyError
clear()	Removes all elements form a set
copy()	Returns a shallow copy of a set
pop()	Removes and returns an arbitrary set element. Raise KeyError if the set is empty
update()	Updates a set with the union of itself and others
union()	Returns the union of sets in a new set
difference()	Returns the difference of two or more sets as a new set

difference_update()	Removes all elements of another set from this set
discard()	Removes an element from set if it is a member. (Do nothing if the element is not in set)
intersection()	Returns the intersection of two sets as a new set
intersection_update()	Updates the set with the intersection of itself and another
isdisjoint()	Returns True if two sets have a null intersection
issubset()	Returns True if another set contains this set
issuperset()	Returns True if this set contains another set
symmetric_difference()	Returns the symmetric difference of two sets as a new set
symmetric_difference_update()	Updates a set with the symmetric difference of itself and another

Dictionary -

Dictionary in Python is an unordered collection of data values, used to store data values like a map, which, unlike other data types which hold only a single value as an element.
Example of Dictionary in Python
Dictionary holds **key:value** pair. Key-Value is provided in the dictionary to make it more optimized.

Ex -
```
Dict = {1: Python, 2: 'For', 3: learn}
print(Dict)
```

Output -
```
{1: Python, 2: 'For', 3: learn}
```

Creating Dictionary
In python a dictionary can be created by placing a sequence of elements within curly **{}** braces, separated by 'comma'.
Dictionary holds pairs of values, one being the Key and the other corresponding pair element being its **Key:value**.
Values in a dictionary can be of any data type and can be duplicated, whereas keys can't be repeated and must be *immutable*.
Note – Dictionary keys are case sensitive, the same name but different cases of Key will be treated distinctly.

```
# Creating a Dictionary with Integer Keys
Dict = {1: python, 2: 'For', 3: learn}
print("\nDictionary with the use of Integer Keys: ")
print(Dict)

# Creating a Dictionary with Mixed keys
Dict = {'Name': python, 1: [1, 2, 3, 4]}
print("\nDictionary with the use of Mixed Keys: ")
print(Dict)
```

Output -

```
Dictionary with the use of Integer Keys:
{1: python, 2: 'For', 3: learn}
Dictionary with the use of Mixed Keys:
{'Name': python, 1: [1, 2, 3, 4]}
```

Dictionary can also be created by the built-in function dict(). An empty dictionary can be created by just placing to curly braces{}.

Adding Elements to the Dictionary

Addition of elements can be done in multiple ways. One value at a time can be added to a Dictionary by defining value along with the key e.g. Dict[Key] = ‘Value’. Updating an existing value in a Dictionary can be done by using the built-in **update()** method. Nested key values can also be added to an existing Dictionary.

Note- While adding a value, if the key-value already exists, the value gets updated otherwise a new Key with the value is added to the Dictionary.

```
# Creating an empty Dictionary
Dict = {}
print("Empty Dictionary: ")
print(Dict)
# Adding elements one at a time
Dict = {1: python, 2: 'For', 3: learn}
Dict[0] = Python
Dict[2] = 'For'
Dict[3] = 1
print("\nDictionary after adding 3 elements: ")
print(Dict)

# Adding set of values to a single Key
Dict['Value_set'] = 2, 3, 4
print("\nDictionary after adding 3 elements: ")
print(Dict)
```

```
# Updating existing Key's Value
Dict[2] = 'Welcome'
print("\nUpdated key value: ")
print(Dict)

# Adding Nested Key value to Dictionary
Dict[5] = {'Nested': {'1': 'Life', '2': Python}}
print("\nAdding a Nested Key: ")
print(Dict)
```

Output -

```
Empty Dictionary:
{}
Dictionary after adding 3 elements:
{0: Python, 2: 'For', 3: 1}
Dictionary after adding 3 elements:
{0: Python, 2: 'For', 3: 1, 'Value_set': (2, 3, 4)}
Updated key value:
{0: Python, 2: 'Welcome', 3: 1, 'Value_set': (2, 3, 4)}
Adding a Nested Key:
{0: Python, 2: 'Welcome', 3: 1, 'Value_set': (2, 3, 4), 5:
{'Nested': {'1': 'Life', '2': Python}}}
```

get()

This method accepts key as argument and returns the value.

```
# Creating a Dictionary
Dict = {1: 'Python', 'name': 'For', 3: 'Python'}

# accessing a element using get() method
print("Accessing a element using get:")
print(Dict.get(3))
```

Output -

```
Accessing a element using get:
Python
```

Accessing a element of a nested dictionary

In order to access the value of any key in the nested dictionary, use indexing [] syntax.

```
# Creating a Dictionary
Dict = {'Dict1': {1: Python},
        'Dict2': {'Name': 'For'}}

# Accessing element using key
print(Dict['Dict1'])
print(Dict['Dict1'][1])
print(Dict['Dict2']['Name'])
```

Output -

```
{1: Python}
Python
For
```

clear()– Remove all the elements from the dictionary
copy()– Returns a copy of the dictionary
get()– Returns the value of specified key
items() – Returns a list containing a tuple for each key value pair
keys() – Returns a list containing dictionary’s keys
pop() – Remove the element with specified key
popitem**()** – Removes the last inserted key-value pair
update()– Updates dictionary with specified key-value pairs
values() – Returns a list of all the values of dictionary

```
# demo for all dictionary methods
dict1 = {1: "Python", 2: "Java", 3: "Ruby", 4: "Scala"}

# copy() method
dict2 = dict1.copy()
print(dict2)

# clear() method
dict1.clear()
print(dict1)

# get() method
print(dict2.get(1))
```

```
# items() method
print(dict2.items())

# keys() method
print(dict2.keys())

# pop() method
dict2.pop(4)
print(dict2)

# popitem() method
dict2.popitem()
print(dict2)

# update() method
dict2.update({3: "Scala"})
print(dict2)

# values() method
print(dict2.values())
```

Output -

```
{1: 'Python', 2: 'Java', 3: 'Ruby', 4: 'Scala'}
{}
Python
dict_items([(1, 'Python'), (2, 'Java'), (3, 'Ruby'), (4, 'Scala')])
dict_keys([1, 2, 3, 4])
{1: 'Python', 2: 'Java', 3: 'Ruby'}
{1: 'Python', 2: 'Java'}
{1: 'Python', 2: 'Java', 3: 'Scala'}
dict_values(['Python', 'Java', 'Scala'])
```

Python Function

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.
As you already know, Python gives you many built-in functions like print(), etc. but you can also create your own functions. These functions are called *user-defined function*

Defining a Function

You can define functions to provide the required functionality. Here are simple rules to define a function in Python.
1. Function blocks begin with the keyword **def** followed by the function name and parentheses (( )).
2. Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
3. The first statement of a function can be an optional statement - the documentation string of the function or *docstring.*
4. The code block within every function starts with a colon (:) and is indented.

- 5. The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

Syntax -

```
def functionname( parameters ):
"function_docstring"
function_suite
return [expression]
```

The following elements make up define a function, as seen above.

- The beginning of a function header is indicated by a keyword called def.
- name_of_function is the function's name that we can use to separate it from others. We will use this name to call the function later in the program. The same criteria apply to naming functions as to naming variables in Python.
- We pass arguments to the defined function using parameters. They are optional, though.
- The function header is terminated by a colon (:).
- We can use a documentation string called docstring in the short form to explain the purpose of the function.
- The body of the function is made up of several valid Python statements. The indentation depth of the whole code block must be the same (usually 4 spaces).
- We can use a return expression to return a value from a defined function.

Example of a User-Defined Function

We will define a function that when called will return the square of the number passed to it as an argument.

```
def square( num ):
    """
    This function computes the square of the number.
    """
    return num**2
object_ = square(9)
print( "The square of the number is: ", object_ )
```

Output - The square of the number is: 81

Calling a Function

A function is defined by using the def keyword and giving it a name, specifying the arguments that must be passed to the function, and structuring the code block.
After a function's fundamental framework is complete, we can call it from anywhere in the program. The following is an example of how to use the a_function function.

```
# Defining a function
def a_function( string ):
    "This prints the value of length of string"
    return len(string)

# Calling the function we defined
print( "Length of the string Functions is: ", a_function( "Functions" ) )
print( "Length of the string Python is: ", a_function( "Python" ) )
```

Output -

```
Length of the string Functions is:  9
Length of the string Python is:  6
```

Pass by Reference vs. Value

In the Python programming language, all arguments are supplied by reference. It implies that if we modify the value of an argument within a function, the change is also reflected in the calling function.

For instance,

```
# defining the function
def square( my_list ):
  """This function will find the square of items in list"""
   squares = []
   for l in my_list:
      squares.append( l**2 )
   return squares

# calling the defined function
list_ = [45, 52, 13];
result = square( list_ )
print( "Squares of the list is: ", result )
```

Output -

```
Squares of the list is:  [2025, 2704, 169]
```

Function Arguments

The following are the types of arguments that we can use to call a function:

- Default arguments
- Keyword arguments
- Required arguments
- Variable-length arguments

Default Arguments

A default argument is a kind of parameter that takes as input a default value if no value is supplied for the argument when the function is called. Default arguments are demonstrated in the following instance.

```
# Python code to demonstrate the use of default arguments  defining a function
def function( num1, num2 = 40 ):
    print("num1 is: ", num1)
    print("num2 is: ", num2)

# Calling the function and passing only one argument
print( "Passing one argument" )
function(10)

# Now giving two arguments to the function
print( "Passing two arguments" )
function(10,30)
```

```
Passing one argument
num1 is:  10
num2 is:  40
Passing two arguments
num1 is:  10
num2 is:  30
```

Keyword Arguments

The arguments in a function called are connected to keyword arguments. If we provide keyword arguments while calling a function, the user uses the parameter label to identify which parameters value it is.Since the Python interpreter will connect the keywords given to link the values with its parameters, we can omit some arguments or arrange them out of order. The function() method can also be called with keywords in the following manner:

```
# Defining a function
def function( num1, num2 ):
   print("num1 is: ", num1)
   print("num2 is: ", num2)
# Calling function and passing arguments without using keyword
print( "Without using keyword" )
function( 50, 30)
# Calling function and passing arguments using keyword
print( "With using keyword" )
function( num2 = 50, num1 = 30)
```

Output -

```
Without using keyword
num1 is:  50
num2 is:  30
With using keyword
num1 is:  30
num2 is:  50
```

Required Arguments

The arguments given to a function while calling in a pre-defined positional sequence are required arguments. The count of required arguments in the method call must be equal to the count of arguments provided while defining the function. We must send two arguments to the function function() in the correct order, or it will return a syntax error, as seen below.

```
# Python code to demonstrate the use of default arguments
# Defining a function
def function( num1, num2 ):
    print("num1 is: ", num1)
    print("num2 is: ", num2)

# Calling function and passing two arguments out of order, we need num1 to be 20 and num2 to be 30
print( "Passing out of order arguments" )
function( 30, 20 )
# Calling function and passing only one argument
print( "Passing only one argument" )
try:
    function( 30 )
except:
    print( "Function needs two positional arguments" )
```

Output -

```
Passing out of order arguments
num1 is:  30
num2 is:  20
Passing only one argument
Function needs two positional arguments
```

Variable-Length Arguments

We can use special characters in Python functions to pass as many arguments as we want in a function. There are two types of characters that we can use for this purpose:

***args** -These are Non-Keyword Arguments
****kwargs** - These are Keyword Arguments.

```
# Defining a function
def function( *args_list ):
    ans = []
    for l in args_list:
        ans.append( l.upper() )
    return ans
# Passing args arguments
object = function('Python', 'Functions', 'tutorial')
print( object )
# defining a function
def function( **kargs_list ):
    ans = []
    for key, value in kargs_list.items():
        ans.append([key, value])
    return ans
# Paasing kwargs arguments
object = function(First = "Python", Second = "Functions", Third = "Tutorial")
print(object)
```

Output -

```
['PYTHON', 'FUNCTIONS', 'TUTORIAL']
[['First', 'Python'], ['Second', 'Functions'], ['Third', 'Tutorial']]
```

return Statement

We write a return statement in a function to leave a function and give the calculated value when a defined function is called.

Syntax:

return < expression to be returned as output >

An argument, a statement, or a value can be used in the return statement, which is given as output when a specific task or function is completed. If we do not write a return statement, then None object is returned by a defined function.

```
# Defining a function with return statement
def square( num ):
    return num**2
# Calling function and passing arguments.
print( "With return statement" )
print( square( 39 ) )

# Defining a function without return statement
def square( num ):
     num**2
# Calling function and passing arguments.
print( "Without return statement" )
print( square( 39 ) )
```

Output -

```
With return statement
1521
Without return statement
None
```

Lambda Functions

Lambda Functions in Python are anonymous functions, implying they don't have a name. The def keyword is needed to create a typical function in Python, as we already know. We can also use the lambda keyword in Python to define an unnamed function.

Syntax of Python Lambda Function

```
lambda arguments: expression
```

This function accepts any count of inputs but only evaluates and returns one expression.
Lambda functions can be used whenever function arguments are necessary. In addition to other forms of formulations in functions, it has a variety of applications in certain coding domains. It's important to remember that according to syntax, lambda functions are limited to a single statement.

Example of Lambda Function in Python

An example of a lambda function that adds 4 to the input number is shown below.

```
# Code to demonstrate how we can use a lambda function
add = lambda num: num + 4
print( add(6) )
```

Output -

```
10
```

Using Lambda Function with filter()

The filter() method accepts two arguments in Python: a function and an iterable such as a list.
The function is called for every item of the list, and a new iterable or list is returned that holds just those elements that returned True when supplied to the function.
Here's a simple illustration of using the filter() method to return only odd numbers from a list.

```
# Code to filter odd numbers from a given list
list_ = [34, 12, 64, 55, 75, 13, 63]
odd_list = list(filter( lambda num: (num % 2 != 0) , list_ ))
print(odd_list)
```

Output -

```
[55, 75, 13, 63]
```

Using Lambda Function with map()

A method and a list are passed to Python's map() function.
The function is executed for all of the elements within the list, and a new list is produced with elements generated by the given function for every item.

Ex -

```
numbers_list = [2, 4, 5, 1, 3, 7, 8, 9, 10]
squared_list = list(map( lambda num: num ** 2 , numbers_list ))
print( squared_list )
```

Output - [4, 16, 25, 1, 9, 49, 64, 81, 100]

Using Lambda Function with List Comprehension

We'll apply the lambda function combined with list comprehension and lambda keyword with a for loop in this instance. We'll attempt to print the square of numbers in the range 0 to 11.

Ex -

```
squares = [lambda num = num: num ** 2 for num in range(0, 11)]
for square in squares:
    print( square(), end = " ")
```

Output -

```
0 1 4 9 16 25 36 49 64 81 100
```

Using Lambda Function with if-else

We will use the lambda function with the if-else block.

Ex -

```
Minimum = lambda x, y : x if (x < y) else y
print(Minimum( 35, 74 ))
```

Output -

```
35
```

Scope and Lifetime of Variables

The scope of a variable refers to the domain of a program wherever it is declared. A function's arguments and variables are not accessible outside the defined function. As a result, they only have a local domain.
The period of a variable's existence in RAM is referred to as its lifetime. Variables within a function have the same lifespan as the function itself.
When we get out of the function, they are removed. As a result, a function does not retain a variable's value from earlier executions.

```
#defining a function to print a number.
def number( ):
    num = 30
    print( "Value of num inside the function: ", num)

num = 20
number()
print( "Value of num outside the function:", num)
```

Output -

```
Value of num inside the function:  30
Value of num outside the function: 2
```

Python Function within Another Function

Functions are considered first-class objects in Python. In a programming language, first-class objects are treated the same wherever they are used. They can be used in conditional expressions, as arguments, and saved in built-in data structures. If a programming language handles functions as first-class entities, it is said to implement first-class functions. Python supports the notion of First Class functions.

Inner or nested function refers to a function defined within another defined function. Inner functions can access the parameters of the outer scope. Inner functions are constructed to cover them from the changes that happen outside the function. Many developers regard this process as encapsulation.

Ex -

```
def function1():
    string = 'Python functions tutorial'

    def function2():
        print( string )

    function2()
function1()
```

Output -

```
Python functions tutorial
```

Python Modules

Modular programming is the practice of segmenting a single, complicated coding task into multiple, simpler, easier-to-manage sub-tasks. We call these subtasks modules. Therefore, we can build a bigger program by assembling different modules that act like building blocks.

Modularizing our code in a big application has a lot of benefits.

Simplification: A module often concentrates on one comparatively small area of the overall problem instead of the full task. We will have a more manageable design problem to think about if we are only concentrating on one module. Program development is now simpler and much less vulnerable to mistakes.

Flexibility: Modules are frequently used to establish conceptual separations between various problem areas. It is less likely that changes to one module would influence other portions of the program if modules are constructed in a fashion that reduces interconnectedness. (We might even be capable of editing a module despite being familiar with the program beyond it.) It increases the likelihood that a group of numerous developers will be able to collaborate on a big project.

Reusability: Functions created in a particular module may be readily accessed by different sections of the assignment (through a suitably established api). As a result, duplicate code is no longer necessary.

Scope: Modules often declare a distinct namespace to prevent identifier clashes in various parts of a program.
In Python, modularization of the code is encouraged through the use of functions, modules, and packages.

What are Modules in Python?

A document with definitions of functions and various statements written in Python is called a Python module.

In Python, we can define a module in one of 3 ways:

- Python itself allows for the creation of modules.
- Similar to the re (regular expression) module, a module can be primarily written in C programming language and then dynamically inserted at run-time.
- A built-in module, such as the itertools module, is inherently included in the interpreter.

A module is a file containing Python code, definitions of functions, statements, or classes. An example_module.py file is a module we will create and whose name is example_module.

We employ modules to divide complicated programs into smaller, more understandable pieces. Modules also allow for the reuse of code.

Rather than duplicating their definitions into several applications, we may define our most frequently used functions in a separate module and then import the complete module.

Ex -

```
# Python program to show how to create a module. defining a function in the module to reuse it
def square( number ):
"""This function will square the number passed to it"""
    result = number ** 2
    return result
```

Here, a module called example_module contains the definition of the function square(). The function returns the square of a given number.

How to Import Modules in Python?

In Python, we may import functions from one module into our program, or as we say into, another module.
For this, we make use of the import Python keyword. In the Python window, we add the next to import keyword, the name of the module we need to import. We will import the module we defined earlier example_module.

import example_module
The functions that we defined in the example_module are not immediately imported into the present program. Only the name of the module, i.e., example_ module, is imported here.
We may use the dot operator to use the functions using the module name. For instance:

Ex-

- result = example_module.square(4)
- **print**("By using the module square of number is: ", result)

O/P - By using the module square of number is: 16

There are several standard modules for Python. The complete list of Python standard modules is available. The list can be seen using the help command.
Similar to how we imported our module, a user-defined module, we can use an import statement to import other standard modules.
Importing a module can be done in a variety of ways. Below is a list of them.

Python import Statement

Using the import Python keyword and the dot operator, we may import a standard module and can access the defined functions within it. Here's an illustration.

# Python program to show how to import a standard module We will import the math module which is a standard modul
e

Ex-

```
import math
print( "The value of euler's number is", math.e )
```

O/P - The value of euler's number is 2.718281828459045

Importing and also Renaming

While importing a module, we can change its name too. Here is an example to show.

# Python program to show how to import a module and rename it.We will import the math module and give a different n
ame to it

Ex-

```
import math as mt
print( "The value of euler's number is", mt.e )
```

O/P- The value of euler's number is 2.718281828459045

The math module is now named mt in this program. In some circumstances, it might help us type faster in case of modules having long names.
Please take note that now the scope of our program does not include the term math. Thus, mt.pi is the proper implementation of the module, whereas math.pi is invalid.

Python from...import Statement

We can import specific names from a module without importing the module as a whole. Here is an example.

# Python program to show how to import specific objects from a module.We will import euler's number from the math module using the from keyword

Ex-

```
from math import e
print( "The value of euler's number is", e )
```

O/P - The value of euler's number is 2.718281828459045

Only the e constant from the math module was imported in this case.
We avoid using the dot (.) operator in these scenarios. As follows, we may import many attributes at the same time:

```
# Python program to show how to import multiple objects from a module

from math import e, tau
print( "The value of tau constant is: ", tau )
print( "The value of the euler's number is: ", e )
```

O/P-
```
The value of tau constant is:  6.283185307179586
The value of the euler's number is:  2.718281828459045
```

Import all Names - From import * Statement

To import all the objects from a module within the present namespace, use the * symbol and the from and import keyword.

Syntax-

```
from name_of_module import *
```

There are benefits and drawbacks to using the symbol *. It is not advised to use * unless we are certain of our particular requirements from the module; otherwise, do so.

Here is an example of the same

```
# importing the complete math module using *
from math import *

# accessing functions of math module without using the dot operator
print( "Calculating square root: ", sqrt(25) )
print( "Calculating tangent of an angle: ", tan(pi/6) ) # here pi is also imported from the math module
```

O/P -
```
Calculating square root:  5.0
Calculating tangent of an angle:  0.5773502691896257
```

Locating Path of Modules

The interpreter searches numerous places when importing a module in the Python program. Several directories are searched if the built-in module is not present. The list of directories can be accessed using sys.path. The Python interpreter looks for the module in the way described below:
The module is initially looked for in the current working directory. Python then explores every directory in the shell parameter PYTHONPATH if the module cannot be located in the current directory. A list of folders makes up the environment variable known as PYTHONPATH. Python examines the installation-dependent set of folders set up when Python is downloaded if that also fails.

```
# We will import the sys module
import sys
# we will import sys.path
print(sys.path)
```

O/P - ['/home/pyodide', '/home/pyodide/lib/Python310.zip', '/lib/Python3.10', '/lib/Python3.10/lib-dynload', '', '/lib/Python3.10/site-packages']

The dir() Built-in Function

We may use the dir() method to identify names declared within a module.

For instance, we have the following names in the standard module str. To print the names, we will use the dir() method in the following way:

Ex-

```
# Python program to print the directory of a module
print( "List of functions:\n ", dir( str ), end=", " )
```

O/P-

```
List of functions:
['__add__', '__class__', '__contains__', '__delattr__',…...........................................]
```

Namespaces and Scoping

Objects are represented by names or identifiers called variables. A namespace is a dictionary containing the names of variables (keys) and the objects that go with them (values).

Both local and global namespace variables can be accessed by a Python statement. When two variables with the same name are local and global, the local variable takes the role of the global variable. There is a separate local namespace for every function. The scoping rule for class methods is the same as for regular functions. Python determines if parameters are local or global based on reasonable predictions. Any variable that is allocated a value in a method is regarded as being local.

Therefore, we must use the global statement before we may provide a value to a global variable inside of a function. Python is informed that Var_Name is a global variable by the line global Var_Name. Python stops looking for the variable inside the local namespace.

We declare the variable Number, for instance, within the global namespace. Since we provide a Number a value inside the function, Python considers a Number to be a local variable. UnboundLocalError will be the outcome if we try to access the value of the local variable without or before declaring it global.

Ex -

```
Number = 204
def AddNumber():
    # accessing the global namespace
    global Number
    Number = Number + 200
print( Number )
AddNumber()
print( Number )        O/P - 204, 404
```

File Handling in Python

Python too supports file handling and allows users to handle files i.e., to read and write files, along with many other file handling options, to operate on files. The concept of file handling has stretched over various other languages, but the implementation is either complicated or lengthy, but like other concepts of Python, this concept here is also easy and short. Python treats files differently as text or binary and this is important. Each line of code includes a sequence of characters and they form a text file. Each line of a file is terminated with a special character, called the EOL or End of Line characters like comma {,} or newline character. It ends the current line and tells the interpreter a new one has begun. Let’s start with the reading and writing files.

Working of open() function

Before performing any operation on the file like reading or writing, first, we have to open that file. For this, we should use Python’s inbuilt function open() but at the time of opening, we have to specify the mode, which represents the purpose of the opening file.

```
f = open(filename, mode)
```

Where the following mode is supported:

- **r:** open an existing file for a read operation.
- **w:** open an existing file for a write operation. If the file already contains some data then it will be overridden.
- **a:** open an existing file for append operation. It won’t override existing data.
- **r+:** To read and write data into the file. The previous data in the file will be overridden.
- **w+:** To write and read data. It will override existing data.
- **a+:** To append and read data from the file. It won’t override existing data.

EX-

```
# a file named "python", will be opened with the reading mode.
file = open(python.txt', 'r')
# This will print every line one by one in the file
for each in file:
    print (each)
```

The open command will open the file in the read mode and the for loop will print each line present in the file.

Working of read() mode

There is more than one way to read a file in Python. If you need to extract a string that contains all characters in the file then we can use **file.read()**. The full code would work like this:

```
# Python code to illustrate read() mode
file = open("file.txt", "r")
print (file.read())
```

Another way to read a file is to call a certain number of characters like in the following code the interpreter will read the first five characters of stored data and return it as a string:

```
# Python code to illustrate read() mode character wise
file = open("file.txt", "r")
print (file.read(5))
```

Creating a file using write() mode

Let’s see how to create a file and how to write mode works, so in order to manipulate the file, write the following in your Python environment:

```
# Python code to create a file
file = open(python.txt','w')
file.write("This is the write command")
file.write("It allows us to write in a particular file")
file.close()
```

The close() command terminates all the resources in use and frees the system of this particular program.

Working of append() mode

Let us see how the append mode works:

```
# Python code to illustrate append() mode
file = open(python.txt','a')
file.write("This will add this line")
file.close()
```

There are also various other commands in file handling that is used to handle various tasks like
rstrip(): This function strips each line of a file off spaces from the right-hand side.
lstrip(): This function strips each line of a file off spaces from the left-hand side.

It is designed to provide much cleaner syntax and exception handling when you are working with code. That explains why it's good practice to use them with a statement where applicable. This is helpful because using this method any files opened will be closed automatically after one is done, so auto-cleanup.

EX -

```
# Python code to illustrate with()
with open("file.txt") as file:
    data = file.read()
# do something with data
```

Using write along with the with() function

We can also use the write function along with the with() function:

Ex-

```
# Python code to illustrate with() alongwith write()
with open("file.txt", "w") as f:
    f.write("Hello World!!!")
```

split() using file handling

We can also split lines using file handling in Python. This splits the variable when space is encountered. You can also split using any characters as we wish. Here is the code:

```
# Python code to illustrate split() function
with open("file.text", "r") as file:
    data = file.readlines()
    for line in data:
        word = line.split()
        print (word)
```

Open a File in Python

Python provides inbuilt functions for creating, writing, and reading files. There are two types of files that can be handled in Python, normal text files and binary files (written in binary language, 0s, and 1s).

- **Text files:** In this type of file, each line of text is terminated with a special character called **EOL (End of Line)**, which is the new line character (‘\n’) in Python by default. In the case of CSV(Comma Separated Files, the EOF is a comma by default.
- **Binary files:** In this type of file, there is no terminator for a line, and the data is stored after converting it into machine-understandable binary language, i.e., 0 and 1 format.

Opening a file

Opening a file refers to getting the file ready either for reading or for writing. This can be done using the open() function. This function returns a file object and takes two arguments, one that accepts the file name and another that accepts the mode(Access Mode). Now, the question arises what is the access mode? Access modes govern the type of operations possible in the opened file.

It refers to how the file will be used once it’s opened. These modes also define the location of the **File Handle** in the file. **File handle** is like a cursor, which defines from where the data has to be read or written in the file. There are 6 access modes in python.

- **Read Only (‘r’):** Open text file for reading. The handle is positioned at the beginning of the file. If the file does not exist, raises an I/O error. This is also the default mode in which the file is opened.
- **Read and Write (‘r+’):** Open the file for reading and writing. The handle is positioned at the beginning of the file. Raises I/O error if the file does not exist.
- **Write Only (‘w’):** Open the file for writing. For the existing files, the data is truncated and over-written. The handle is positioned at the beginning of the file. Creates the file if the file does not exist.
- **Write and Read (‘w+’):** Open the file for reading and writing. For existing files, data is truncated and over-written. The handle is positioned at the beginning of the file.
- **Append Only (‘a’):** Open the file for writing. The file is created if it does not exist. The handle is positioned at the end of the file. The data being written will be inserted at the end, after the existing data.
- **Append and Read (‘a+’):** Open the file for reading and writing. The file is created if it does not exist. The handle is positioned at the end of the file. The data being written will be inserted at the end, after the existing data.
- **Read Only in Binary format(‘rb’):** It lets the user open the file for reading in binary format.
- **Read and Write in Binary Format(‘rb+’):** It lets the user open the file for reading and writing in binary format.
- **Write Only in Binary Format(‘wb’):** It lets the user open the file for writing in binary format. When a file gets opened in this mode, there are two things that can happen mostly. A new file gets created if the file does not exist. The content within the file will get overwritten if the file exists and has some data stored in it.

- **Write and Read in Binary Format('wb+'):** It lets the user open the file for reading as well as writing in binary format. When a file gets opened in this mode, there are two things that can mostly happen. A new file gets created for writing and reading if the file does not exist. The content within the file will get overwritten if the file exists and has some data stored in it.
- **Append only in Binary Format('ab'):** It lets the user open the file for appending in binary format. A new file gets created if there is no file. The data will be inserted at the end if the file exists and has some data stored in it.
- **Append and Read in Binary Format('ab+'):** It lets the user open the file for appending and reading in binary format. A new file will be created for reading and appending if the file does not exist. We can read and append if the file exists and has some data stored in it.

Syntax: File_object = open(r"File_Name", "Access_Mode")

Note: *The file should exist in the same directory as the Python script, otherwise full address of the file should be written. If the file is not exist, then an error is generated, that the file does not exist.*

# Python program to demonstrate opening a file.Open function to open the file "myfile.txt".(same directory) in read mode and store.it's reference in the variable file1

```
file1 = open(& quot
        myfile.txt & quot
        )
#Reading from file
print(file1.read())
file1.close()
```

O/P - Welcome to GeeksForGeeks!!

Note: In the above example, we haven’t provided the access mode. By default, the open() function will open the file in read mode, if no parameter is provided.

Example #2: Adding data to the existing file in Python
If you want to add more data to an already created file, then the access mode should be ‘a’ which is append mode, if we select ‘w’ mode then the existing text will be overwritten by the new data.

```
# Python program to demonstrate opening a file
# Open function to open the file "myfile.txt"
# (same directory) in append mode and store
# it's reference in the variable file1
file1 = open(" myfile.txt" , & quot; a" )

# Writing to file
file1.write(" \nWriting to file:)" )

# Closing file
file1.close()
```

How to read from a file in Python

Python provides inbuilt functions for creating, writing and reading files. There are two types of files that can be handled in python, normal text files and binary files (written in binary language, 0s and 1s).

- **Text files:** In this type of file, Each line of text is terminated with a special character called EOL (End of Line), which is the new line character ('\n') in python by default.
- **Binary files:** In this type of file, there is no terminator for a line and the data is stored after converting it into machine-understandable binary language.

Access mode

Access modes govern the type of operations possible in the opened file. It refers to how the file will be used once it's opened. These modes also define the location of the File Handle in the file. File handle is like a cursor, which defines from where the data has to be read or written in the file. Different access modes for reading a file are –

- **Read Only ('r') :** Open text file for reading. The handle is positioned at the beginning of the file. If the file does not exists, raises I/O error. This is also the default mode in which file is opened.
- **Read and Write ('r+') :** Open the file for reading and writing. The handle is positioned at the beginning of the file. Raises I/O error if the file does not exists.
- **Append and Read ('a+') :** Open the file for reading and writing. The file is created if it does not exist. The handle is positioned at the end of the file. The data being written will be inserted at the end, after the existing data.

Opening a File

It is done using the open() function. No module is required to be imported for this function.

Syntax:

File_object = open(r"File_Name", "Access_Mode")

The file should exist in the same directory as the python program file else, full address of the file should be written on place of filename.

Note: The r is placed before filename to prevent the characters in filename string to be treated as special character. For example, if there is \temp in the file address, then \t is treated as the tab character and error is raised of invalid address. The r makes the string raw, that is, it tells that the string is without any special characters. The r can be ignored if the file is in same directory and address is not being placed.

```
# Open function to open the file "MyFile1.txt"
# (same directory) in read mode and
file1 = open("MyFile.txt", "r")

# store its reference in the variable file1
# and "MyFile2.txt" in D:\Text in file2
file2 = open(r"D:\Text\MyFile2.txt", "r+")
```

Here, file1 is created as object for MyFile1 and file2 as object for MyFile2.

Closing a file

close() function closes the file and frees the memory space acquired by that file. It is used at the time when the file is no longer needed or if it is to be opened in a different file mode.

Syntax:

File_object.close()

```
# Opening and Closing a file "MyFile.txt"
# for object name file1.
file1 = open("MyFile.txt", "r")
file1.close()
```

Reading from a file

There are three ways to read data from a text file.

- **read() :** Returns the read bytes in form of a string. Reads n bytes, if no n specified, reads the entire file.File_object.read([n])
-
- **readline() :** Reads a line of the file and returns in form of a string.For specified n, reads at most n bytes. However, does not reads more than one line, even if n exceeds the length of the line.File_object.readline([n])

- **readlines() :** Reads all the lines and return them as each line a string element in a list.File_object.readlines()

Note: '\n' is treated as a special character of two bytes.

```
# Program to show various ways to read data from a file.
# Creating a file
file1 = open("myfile.txt", "w")
L = ["This is Delhi \n", "This is Paris \n", "This is London \n"]

# Writing data to a file
file1.write("Hello \n")
file1.writelines(L)
file1.close() # to change file access modes

file1 = open("myfile.txt", "r+")

print("Output of Read function is ")
print(file1.read())
print()

# seek(n) takes the file handle to the nth bite from the beginning.
file1.seek(0)

print("Output of Readline function is ")
print(file1.readline())
print()
file1.seek(0)
```

```
# To show difference between read and readline
print("Output of Read(9) function is ")
print(file1.read(9))
print()

file1.seek(0)

print("Output of Readline(9) function is ")
print(file1.readline(9))
print()

file1.seek(0)

# readlines function
print("Output of Readlines function is ")
print(file1.readlines())
print()
file1.close()
```

```
Output of Read function is
Hello
This is Delhi
This is Paris
This is London


Output of Readline function is
Hello


Output of Read(9) function is
Hello
Th

Output of Readline(9) function is
Hello


Output of Readlines function is
['Hello \n', 'This is Delhi \n', 'This is Paris \n', 'This is London \n']
```

With statement

with statement in Python is used in exception handling to make the code cleaner and much more readable. It simplifies the management of common resources like file streams. Unlike the above implementations, there is no need to call file.close() when using with statement. The with statement itself ensures proper acquisition and release of resources.

Syntax:

with open filename as file:

```
# Program to show various ways to read data from a file.

L = ["This is Delhi \n", "This is Paris \n", "This is London \n"]

# Creating a file
with open("myfile.txt", "w") as file1:
    # Writing data to a file
    file1.write("Hello \n")
    file1.writelines(L)
    file1.close() # to change file access modes
with open("myfile.txt", "r+") as file1:
    # Reading form a file
    print(file1.read())
```

Output – Hello , This is Delhi ,This is Paris , This is London

Writing to file in Python

Python provides inbuilt functions for creating, writing and reading files. There are two types of files that can be handled in python, normal text files and binary files (written in binary language, 0s and 1s).

- **Text files:** In this type of file, Each line of text is terminated with a special character called EOL (End of Line), which is the new line character ('\n') in python by default.
- **Binary files:** In this type of file, there is no terminator for a line and the data is stored after converting it into machine-understandable binary language.

Access mode

Access modes govern the type of operations possible in the opened file. It refers to how the file will be used once it's opened. These modes also define the location of the File Handle in the file. File handle is like a cursor, which defines from where the data has to be read or written in the file. Different access modes for reading a file are –

- **Write Only ('w') :** Open the file for writing. For an existing file, the data is truncated and over-written. The handle is positioned at the beginning of the file. Creates the file if the file does not exist.
- **Write and Read ('w+') :** Open the file for reading and writing. For an existing file, data is truncated and over-written. The handle is positioned at the beginning of the file.
- **Append Only ('a') :** Open the file for writing. The file is created if it does not exist. The handle is positioned at the end of the file. The data being written will be inserted at the end, after the existing data.

Opening a File

It is done using the open() function. No module is required to be imported for this function.

Syntax:

File_object = open(r"File_Name", "Access_Mode")

The file should exist in the same directory as the python program file else, full address of the file should be written on place of filename.

Note: The r is placed before filename to prevent the characters in filename string to be treated as special character. For example, if there is \temp in the file address, then \t is treated as the tab character and error is raised of invalid address. The r makes the string raw, that is, it tells that the string is without any special characters. The r can be ignored if the file is in same directory and address is not being placed.

```
# Open function to open the file "MyFile1.txt" (same directory) in read mode and
file1 = open("MyFile.txt", "w")

# store its reference in the variable file1
# and "MyFile2.txt" in D:\Text in file2
file2 = open(r"D:\Text\MyFile2.txt", "w+")
```

Here, file1 is created as object for MyFile1 and file2 as object for MyFile2.

Closing a file

close() function closes the file and frees the memory space acquired by that file. It is used at the time when the file is no longer needed or if it is to be opened in a different file mode.

Syntax:

```
File_object.close()
```

```
# Opening and Closing a file "MyFile.txt"
# for object name file1.
file1 = open("MyFile.txt", "w")
file1.close()
```

Writing to file

There are two ways to write in a file.

- **write() :** Inserts the string str1 in a single line in the text file.File_object.write(str1)
-
- **writelines() :** For a list of string elements, each string is inserted in the text file. Used to insert multiple strings at a single time.File_object.writelines(L) for L = [str1, str2, str3]
-

Note: ‘\n’ is treated as a special character of two bytes.

```python
# Python program to demonstrate writing to file
# Opening a file
file1 = open('myfile.txt', 'w')
L = ["This is Delhi \n", "This is Paris \n", "This is London \n"]
s = "Hello\n"

# Writing a string to file
file1.write(s)
# Writing multiple strings at a time
file1.writelines(L)
# Closing file
file1.close()
# Checking if the data is written to file or not
file1 = open('myfile.txt', 'r')
print(file1.read())
file1.close()
```

Output:

```
Hello
This is Delhi
This is Paris
This is London
```

Appending to a file

When the file is opened in append mode, the handle is positioned at the end of the file. The data being written will be inserted at the end, after the existing data. Let’s see the below example to clarify the difference between write mode and append mode.

```
# Python program to illustrate Append vs write mode
file1 = open("myfile.txt", "w")
L = ["This is Delhi \n", "This is Paris \n", "This is London \n"]
file1.writelines(L)
file1.close()

# Append-adds at last
file1 = open("myfile.txt", "a") # append mode
file1.write("Today \n")
file1.close()

file1 = open("myfile.txt", "r")
print("Output of Readlines after appending")
print(file1.read())
print()
file1.close()
```

```
# Write-Overwrites
file1 = open("myfile.txt", "w") # write mode
file1.write("Tomorrow \n")
file1.close()

file1 = open("myfile.txt", "r")
print("Output of Readlines after writing")
print(file1.read())
print()
file1.close()
```

Output:

```
Output of Readlines after appending
This is Delhi
This is Paris
This is London
Today


Output of Readlines after writing
Tomorrow
```

With statement

with statement in Python is used in exception handling to make the code cleaner and much more readable. It simplifies the management of common resources like file streams. Unlike the above implementations, there is no need to call file.close() when using with statement. The with statement itself ensures proper acquisition and release of resources.

Syntax:

```
with open filename as file:
```

```
# Program to show various ways to write data to a file using with statement
L = ["This is Delhi \n", "This is Paris \n", "This is London \n"]
# Writing to file
with open("myfile.txt", "w") as file1:
    # Writing data to a file
    file1.write("Hello \n")
    file1.writelines(L)
# Reading from file
with open("myfile.txt", "r+") as file1:
    # Reading form a file
    print(file1.read())
```

O/P -
```
Hello
This is Delhi
This is Paris
This is London
```

Python append to a file

While reading or writing to a file, access mode governs the type of operations possible in the opened file. It refers to how the file will be used once it’s opened. These modes also define the location of the File Handle in the file. The file handle is like a cursor, which defines from where the data has to be read or written in the file. In order to append a new line to the existing file, open the file in append mode, by using either ‘a’ or ‘a+’ as the access mode. The definition of these access modes is as follows:

- **Append Only (‘a’):** Open the file for writing. The file is created if it does not exist. The handle is positioned at the end of the file. The data being written will be inserted at the end, after the existing data.
- **Append and Read (‘a+’):** Open the file for reading and writing. The file is created if it does not exist. The handle is positioned at the end of the file. The data being written will be inserted at the end, after the existing data.

When the file is opened in append mode, the handle is positioned at the end of the file. The data being written will be inserted at the end, after the existing data. Let’s see the below example to clarify the difference between write mode and append mode.

Example 1: Python program to illustrate Append vs write mode.

```
file1 = open("myfile.txt", "w")
L = ["This is Delhi \n", "This is Paris \n", "This is London"]
file1.writelines(L)
file1.close()
```

```
# Append-adds at last
file1 = open("myfile.txt", "a") # append mode
file1.write("Today \n")
file1.close()

file1 = open("myfile.txt", "r")
print("Output of Readlines after appending")
print(file1.read())
print()
file1.close()

# Write-Overwrites
file1 = open("myfile.txt", "w") # write mode
file1.write("Tomorrow \n")
file1.close()

file1 = open("myfile.txt", "r")
print("Output of Readlines after writing")
print(file1.read())
print()
file1.close()
```

Output:

```
Output of Readlines after appending
This is Delhi
This is Paris
This is LondonToday

Output of Readlines after writing
Tomorrow
```

Example 2: Append data from a new line

In the above example, it can be seen that the data is not appended from the new line. This can be done by writing the newline ‘\n’ character to the file.

Note: ‘\n’ is treated as a special character of two bytes.

```
# Python program to illustrate append from new line

file1 = open("myfile.txt", "w")
L = ["This is Delhi \n", "This is Paris \n", "This is London"]
file1.writelines(L)
file1.close()
```

```
# Append-adds at last append mode

file1 = open("myfile.txt", "a")

# writing newline character

file1.write("\n")
file1.write("Today")

# without newline character

file1.write("Tomorrow")
file1 = open("myfile.txt", "r")
print("Output of Readlines after appending")
print(file1.read())
print()
file1.close()
```

Example 3: Using With statement in Python

with statement is used in exception handling to make the code cleaner and much more readable. It simplifies the management of common resources like file streams. Unlike the above implementations, there is no need to call file.close() when using with statement. The with statement itself ensures proper acquisition and release of resources.

```
# Program to show various ways to append data to a file using with statement

L = ["This is Delhi \n", "This is Paris \n", "This is London \n"]

# Writing to file
with open("myfile.txt", "w") as file1:
    # Writing data to a file
    file1.write("Hello \n")
    file1.writelines(L)

# Appending to file
with open("myfile.txt", 'a') as file1:
    file1.write("Today")

# Reading from file
with open("myfile.txt", "r+") as file1:
    # Reading form a file
    print(file1.read())
```

O/P -

```
Hello
This is Delhi
This is Paris
This is London
Today
```

Python Exceptions

When a Python program meets an error, it stops the execution of the rest of the program. An error in Python might be either an error in the syntax of an expression or a Python exception. We will see what an exception is. Also, we will see the difference between a syntax error and an exception in this tutorial. Following that, we will learn about trying and except blocks and how to raise exceptions and make assertions. After that, we will see the Python exceptions list.

What is an Exception?

An exception in Python is an incident that happens while executing a program that causes the regular course of the program's commands to be disrupted. When a Python code comes across a condition it can't handle, it raises an exception. An object in Python that describes an error is called an exception.

When a Python code throws an exception, it has two options: handle the exception immediately or stop and quit.

Exceptions versus Syntax Errors

When the interpreter identifies a statement that has an error, syntax errors occur. Consider the following scenario:

```
#Python code after removing the syntax error
string = "Python Exceptions"
for s in string:
    if (s != o:
        print( s )
```

O/P -
```
if (s != o:
       ^
SyntaxError: invalid syntax
```

The arrow in the output shows where the interpreter encountered a syntactic error. There was one unclosed bracket in this case. Close it and rerun the program

```
#Python code after removing the syntax error
string = "Python Exceptions"

for s in string:
    if (s != o):
        print( s )
```

Output:

```
     2 string = "Python Exceptions"
      4 for s in string:
----> 5     if (s != o):
      6         print( s )

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

We encountered an exception error after executing this code. When syntactically valid Python code produces an error, this is the kind of error that arises. The output's last line specified the name of the exception error code encountered. Instead of displaying just "exception error", Python displays information about the sort of exception error that occurred. It was a NameError in this situation. Python includes several built-in exceptions. However, Python offers the facility to construct custom exceptions.

Try and Except Statement - Catching Exceptions

In Python, we catch exceptions and handle them using try and except code blocks. The try clause contains the code that can raise an exception, while the except clause contains the code lines that handle the exception. Let's see if we can access the index from the array, which is more than the array's length, and handle the resulting exception.

```
# Python code to catch an exception and handle it using try and except code blocks

a = ["Python", "Exceptions", "try and except"]
try:
    #looping through the elements of the array a, choosing a range that goes beyond the length of the array
     for i in range( 4 ):
        print( "The index and element from the array is", i, a[i] )
#if an error occurs in the try block, then except block will be executed by the Python interpreter
except:
    print ("Index out of range")
```

Output:

```
The index and element from the array is 0 Python
The index and element from the array is 1 Exceptions
The index and element from the array is 2 try and except
Index out of range
```

The code blocks that potentially produce an error are inserted inside the try clause in the preceding example. The value of i greater than 2 attempts to access the list's item beyond its length, which is not present, resulting in an exception. The except clause then catches this exception and executes code without stopping it.

How to Raise an Exception

If a condition does not meet our criteria but is correct according to the Python interpreter, we can intentionally raise an exception using the raise keyword. We can use a customized exception in conjunction with the statement.

If we wish to use raise to generate an exception when a given condition happens, we may do so as follows:

```
#Python code to show how to raise an exception in Python
num = [3, 4, 5, 7]
if len(num) > 3:
    raise Exception( f"Length of the given list must be less than or equal to 3 but is {len(num)}" )
```

Output:

```
     1 num = [3, 4, 5, 7]
      2 if len(num) > 3:
----> 3     raise Exception( f"Length of the given list must be less than or equal to 3 but is {len(num)}" )

Exception: Length of the given list must be less than or equal to 3 but is 4
```

The implementation stops and shows our exception in the output, providing indications as to what went incorrect.

Assertions in Python

When we're finished verifying the program, an assertion is a consistency test that we can switch on or off.
The simplest way to understand an assertion is to compare it with an if-then condition. An exception is thrown if the outcome is false when an expression is evaluated.
Assertions are made via the assert statement, which was added in Python 1.5 as the latest keyword.
Assertions are commonly used at the beginning of a function to inspect for valid input and at the end of calling the function to inspect for valid output.
The assert Statement
Python examines the adjacent expression, preferably true when it finds an assert statement. Python throws an AssertionError exception if the result of the expression is false.

The syntax for the assert clause is −

assert Expressions[, Argument]
Python uses ArgumentException, if the assertion fails, as the argument for the AssertionError. We can use the try-except clause to catch and handle AssertionError exceptions, but if they aren't, the program will stop, and the Python interpreter will generate a traceback.

Output:

```
     7 #Calling function and passing the values
----> 8 print( square_root( 36 ) )
      9 print( square_root( -36 ) )

Input In [23], in square_root(Number)
      3 def square_root( Number ):
----> 4     assert ( Number < 0), "Give a positive integer"
      5     return Number**(1/2)

AssertionError: Give a positive integer
```

Try with Else Clause

Python also supports the else clause, which should come after every except clause, in the try, and except blocks. Only when the try clause fails to throw an exception the Python interpreter goes on to the else block.

```
# Python program to show how to use else clause with try and except clauses

# Defining a function which returns reciprocal of a number
def reciprocal( num1 ):
    try:
        reci = 1 / num1
    except ZeroDivisionError:
        print( "We cannot divide by zero" )
    else:
        print ( reci )
# Calling the function and passing values
reciprocal( 4 )
reciprocal( 0 )
```

Output:

```
0.25
We cannot divide by zero
```

Finally Keyword in Python

The finally keyword is available in Python, and it is always used after the try-except block. The finally code block is always executed after the try block has terminated normally or after the try block has terminated for some other reason. Here is an example of finally keyword with try-except clauses:

```
# Python code to show the use of finally clause.Raising an exception in try block

try:
    div = 4 // 0
    print( div )
# this block will handle the exception raised
except ZeroDivisionError:
    print( "Atepting to divide by zero" )
# this will always be executed no matter exception is raised or not
finally:
    print( 'This is code of finally clause' )
```

Output:

```
Atepting to divide by zero
This is code of finally clause
```

User-Defined Exceptions

By inheriting classes from the typical built-in exceptions, Python also lets us design our customized exceptions.
Here is an illustration of a RuntimeError. In this case, a class that derives from RuntimeError is produced. Once an exception is detected, we can use this to display additional detailed information.
We raise a user-defined exception in the try block and then handle the exception in the except block.
An example of the class EmptyError is created using the variable var.

```
class EmptyError( RuntimeError ):
   def __init__(self, argument):
      self.arguments = argument
```

Once the preceding **class** has been created, the following **is** how to **raise** an exception:

Code

```
var = " "
try:
    raise EmptyError( "The variable is empty" )
except (EmptyError, var):
    print( var.arguments )
```

Output:

```
     2 try:
----> 3     raise EmptyError( "The variable is empty" )
      4 except (EmptyError, var):

EmptyError: The variable is empty
```

Python Exceptions Listc.Here is the complete list of Python in-built exceptions.

Sr.No.	Name of the Exception	Description of the Exception
1	Exception	All exceptions of Python have a base class.
2	StopIteration	If the next() method returns null for an iterator, this exception is raised.
3	SystemExit	The sys.exit() procedure raises this value.
4	StandardError	Excluding the StopIteration and SystemExit, this is the base class for all Python built-in exceptions.
5	ArithmeticError	All mathematical computation errors belong to this base class.
6	OverflowError	This exception is raised when a computation surpasses the numeric data type's maximum limit.
7	FloatingPointError	If a floating-point operation fails, this exception is raised.
8	**ZeroDivisionError**	**For all numeric data types, its value is raised whenever a number is attempted to be divided by zero.**
9	AssertionError	If the Assert statement fails, this exception is raised.
10	AttributeError	This exception is raised if a variable reference or assigning a value fails.

11	**EOFError**	**When the endpoint of the file is approached, and the interpreter didn't get any input value by raw_input() or input() functions, this exception is raised.**
12	ImportError	This exception is raised if using the import keyword to import a module fails.
13	KeyboardInterrupt	If the user interrupts the execution of a program, generally by hitting Ctrl+C, this exception is raised.
14	**LookupError**	**LookupErrorBase is the base class for all search errors.**
15	IndexError	This exception is raised when the index attempted to be accessed is not found.
16	KeyError	When the given key is not found in the dictionary to be found in, this exception is raised.
17	NameError	This exception is raised when a variable isn't located in either local or global namespace.
18	UnboundLocalError	This exception is raised when we try to access a local variable inside a function, and the variable has not been assigned any value.
19	EnvironmentError	All exceptions that arise beyond the Python environment have this base class.
20	IOError	If an input or output action fails, like when using the print command or the open() function to access a file that does not exist, this exception is raised.

22	SyntaxError	This exception is raised whenever a syntax error occurs in our program.
23	IndentationError	This exception was raised when we made an improper indentation.
24	SystemExit	This exception is raised when the sys.exit() method is used to terminate the Python interpreter. The parser exits if the situation is not addressed within the code.
25	TypeError	This exception is raised whenever a data type-incompatible action or function is tried to be executed.
26	ValueError	This exception is raised if the parameters for a built-in method for a particular data type are of the correct type but have been given the wrong values.
27	RuntimeError	This exception is raised when an error that occurred during the program's execution cannot be classified.
28	NotImplementedError	If an abstract function that the user must define in an inherited class is not defined, this exception is raised.

Summary

We learned about different methods to raise, catch, and handle Python exceptions after learning the distinction between syntax errors and exceptions. We learned about these clauses in this tutorial:

- We can throw an exception at any line of code using the raise keyword.
- Using the assert keyword, we may check to see if a specific condition is fulfilled and raise an exception if it is not.
- All statements are carried out in the try clause until an exception is found.
- The try clause's exception(s) are detected and handled using the except function.
- If no exceptions are thrown in the try code block, we can write code to be executed in the else code block.

Here is the syntax of try, except, else, and finally clauses.

Syntax:

```
try:
    # Code block
    # These statements are those which can probably have some error
except:
    # This block is optional.
    # If the try block encounters an exception, this block will handle it.
else:
    # If there is no exception, this code block will be executed by the Python interpreter
finally:
    # Python interpreter will always execute this code.
```