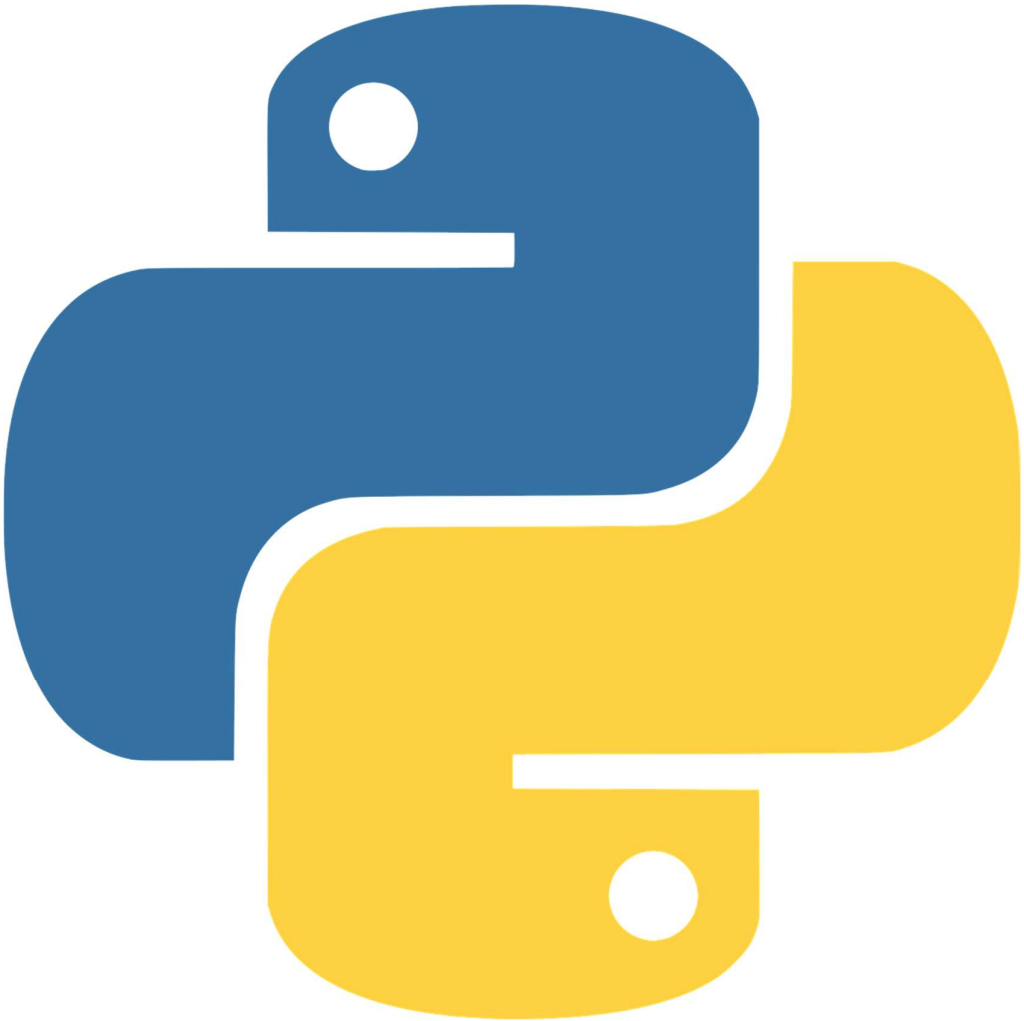
Python Programming



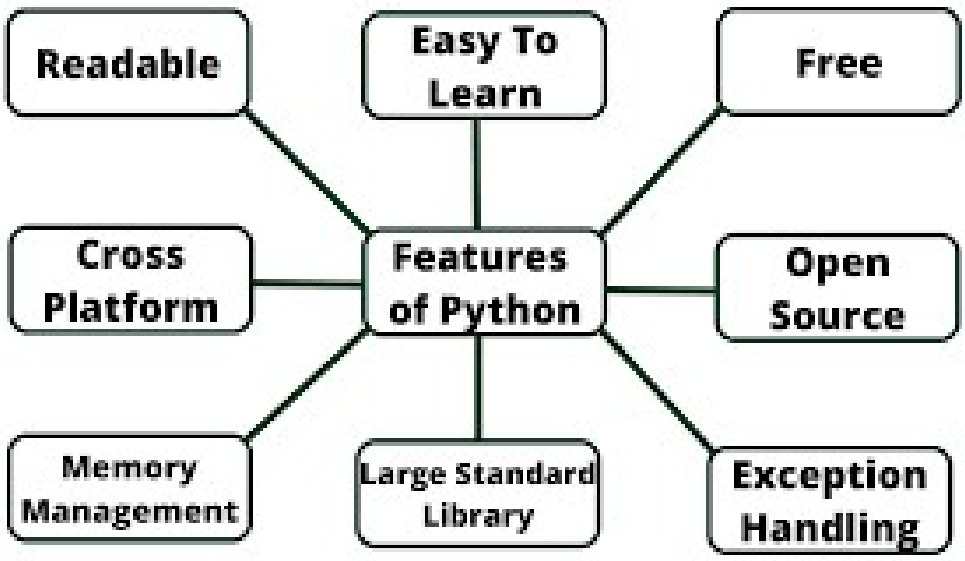
Introduction to Python

Python is an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming. Python's elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms.

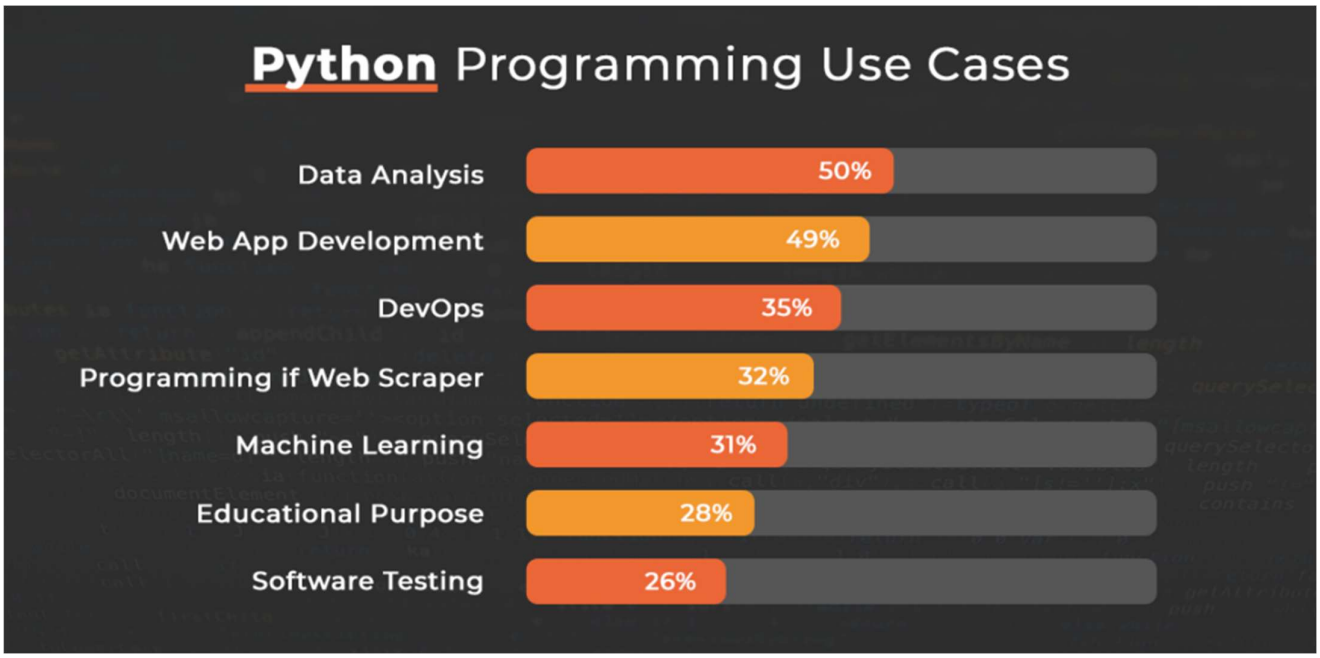
The Python interpreter is easily extended with new functions and data types implemented in C or C++ (or other languages callable from C). Python is also suitable as an extension language for customizable applications.

Python is developed by **Guido van Rossum**. Guido van Rossum started implementing Python in 1989. Python is a very simple programming language so even if you are new to programming, you can learn python without facing any issues.

Features of Python



Use Cases of Python



Variables in Python

Data Types

Data types are the classification or categorization of data items. It represents the kind of value that tells what operations can be performed on a particular data. Since everything is an object in Python programming, data types are actually classes and variables are instance (object) of these classes.

Name	Type	Description
Integers	int	Whole numbers, such as: 3 300 200
Floating point	float	Numbers with a decimal point: 2.3 4.6 100.0
Strings	str	Ordered sequence of characters: "hello" 'Sammy' "2000" "楽しい"
Lists	list	Ordered sequence of objects: [10,"hello",200.3]
Dictionaries	dict	Unordered Key:Value pairs: {"mykey" : "value" , "name" : "Frankie"}
Tuples	tup	Ordered immutable sequence of objects: (10,"hello",200.3)
Sets	set	Unordered collection of unique objects: {"a","b"}
Booleans	bool	Logical value indicating **True** or **False**

Variables

Objects are Python's abstraction for data. In Python, data are represented by objects or by relations between objects. Use the type() function to get the class name of an object. For example, the following displays the class name of integer value.

```
>>> type(10)
<class 'int'>
```

The type of 10 is int. An object of int class contains a integer literal 10. The same thing for string value too.

```
>>> type('Hello World')
<class 'string'>
```

Variables in Python are names given to objects, so that it becomes easy to refer a value. In other words, a variable points to an object. A literal value is assigned to a variable using the = operator where the left side should be the name of a variable, and the right side should be a value. The following assigns a name to an integer value.

```
>>> num=10
```

```
>>> print(num) #display value
10
>>> print(num * 2) # multiply and display result
20
```

```
>>> greet='Hello World'
>>> print(greet)
Hello World
>>> type(greet)
<class 'string'>
```

```
>>> x=5
>>> y=5
>>> x+y
10
>>> x='Hello '
>>> y='World'
>>> x+y
'Hello World'
```

Each object in Python has an id. It is the object's address in memory represented by an integer value. The id() function returns the id of the specified object where it is stored, as shown below.

```
>>> x=100
>>> id(x)
8791062077568
>>> greet='Hello'
>>> id(greet)
4521652332
```

Naming Conventions : Any suitable identifier can be used as a name of a variable, based on the following rules:

1. The name of the variable should start with either an alphabet letter (lower or upper case) or an underscore (_), but it cannot start with a digit.
2. More than one alpha-numeric characters or underscores may follow.
3. The variable name can consist of alphabet letter(s), number(s) and underscore(s) only. For

4. example, myVar, MyVar, _myVar, MyVar123 are valid variable names, but m*var, my-var, 1myVar are invalid variable names.
5. Variable names in Python are case sensitive. So, NAME, name, nAME, and nAmE are treated as different variable names.
6. Variable names cannot be a reserved keywords in Python.

Operators in Python

Arithmetic Operators

Operator	Description	Example
+	Addition - Adds values on either side of the operator	a + b will give 30
-	Subtraction - Subtracts right hand operand from left hand operand	a - b will give -10
*	Multiplication - Multiplies values on either side of the operator	a * b will give 200
/	Division - Divides left hand operand by right hand operand	b / a will give 2
%	Modulus - Divides left hand operand by right hand operand and returns remainder	b % a will give 0
**	Exponent - Performs exponential (power) calculation on operators	a**b will give 10 to the power 20
//	Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed.	9//2 is equal to 4 and 9.0//2.0 is equal to 4.0

Logical Operators

Operator	Description	Example
and	Called Logical AND operator. If both the operands are true then then condition becomes true.	(a and b) is true.
or	Called Logical OR Operator. If any of the two operands are non zero then then condition becomes true.	(a or b) is true.
not	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	not(a and b) is false.

Comparison Operator

Operator	Description	Example
==	Checks if the value of two operands are equal or not, if yes then condition becomes true.	(a == b) is not true.
!=	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.	(a != b) is true.
<>	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.	(a <> b) is true. This is similar to != operator.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(a > b) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(a < b) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(a >= b) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(a <= b) is true.

Identity Operator

Operator	Description	Example
is	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.	x is y, here **is** results in 1 if id(x) equals id(y).
is not	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.	x is not y, here **is not** results in 1 if id(x) is not equal to id(y).

Assignment Operator

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	c = a + b will assigne value of a + b into c
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	c += a is equivalent to c = c + a
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	c -= a is equivalent to c = c - a
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	c *= a is equivalent to c = c * a
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	c /= a is equivalent to c = c / a
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	c %= a is equivalent to c = c % a

Membership Operator

Operator	Description	Example
in	Evaluates to true if it finds a variable in the specified sequence and false otherwise.	x in y, here **in** results in a 1 if x is a member of sequence y.
not in	Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.	x not in y, here **not in** results in a 1 if x is not a member of sequence y.

Bitwise Operator

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(a & b) will give 12 which is 0000 1100
|	Binary OR Operator copies a bit if it exists in eather operand.	(a | b) will give 61 which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(a ^ b) will give 49 which is 0011 0001
~	Binary Ones Complement Operator is unary and has the efect of 'flipping' bits.	(~a) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	a << 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	a >> 2 will give 15 which is 0000 1111

Data Types in Python

Python has five standard Data Types:

- Numbers
- String
- List
- Tuple
- Dictionary

Python sets the variable type based on the value that is assigned to it. Unlike more riggers languages, Python will change the variable type if the variable value is set to another value. For example:

```
var = 123 # This will create a number integer assignment
var = 'john' # the `var` variable is now a string type.
```

Numbers

Python numbers variables are created by the standard Python method:

```
var = 382
```

Most of the time using the standard Python number type is fine. Python will automatically convert a number from one type to another if it needs. But, under certain circumstances that a specific number type is needed (ie. complex, hexidecimal), the format can be forced into a format by using additional syntax in the table below:

Type	Format	Description
int	a = 10	Signed Integer
long	a = 345L	(L) Long integers, they can also be represented in octal and hexadecimal
float	a = 45.67	(.) Floating point real values
complex	a = 3.14J	(J) Contains integer in the range 0 to 255.

Most of the time Python will do variable conversion automatically. You can also use Python conversion functions (int(), long(), float(), complex()) to convert data from one type to another. In addition, the `type` function returns information about how your data is stored within a variable.

```
message = "Good morning"
num = 85
pi = 3.14159


print(type(message))  # This will return a string
print(type(n))  # This will return an integer
print(type(pi))  # This will return a float
```

String

Create string variables by enclosing characters in quotes. Python uses single quotes ' double quotes " and triple quotes """ to denote literal strings. Only the triple quoted strings """ also will automatically continue across the end of line statement.

```
firstName = 'john'
lastName = "smith"
message = """This is a string that will span across multiple lines. Using newline characters
and no spaces for the next lines. The end of lines within this string also count as a newline when printed"""
```

Strings can be accessed as a whole string, or a substring of the complete variable using brackets '[]'. Here are a couple examples:

```
var1 = 'Hello World!'
var2 = 'RhinoPython'

print var1[0] # this will print the first character in the string an `H`
print var2[1:5] # this will print the substring 'hinoP`
```

Python can use a special syntax to format multiple strings and numbers. The string formatter is quickly covered here because it is seen often and it is important to recognize the syntax.

```
print "The item {} is repeated {} times".format(element,count))
```

The `{}` are placeholders that are substituted by the variables `element` and `count` in the final string. This compact syntax is meant to keep the code more readable and compact.

Python is currently transitioning to the format syntax above, but python can use an older syntax, which is being phased out, but is still seen in some example code:

```
print "The item %i is repeated %i times"% (element,count)
```

List

Lists are a very useful variable type in Python. A list can contain a series of values. List variables are declared by using brackets `[ ]` following the variable name.

```
A = [ ] # This is a blank list variable
B = [1, 23, 45, 67] # this list creates an initial list of 4 numbers.
C = [2, 4, 'john'] # lists can contain different variable type
```

```
mylist = ['Rhino', 'Grasshopper', 'Flamingo', 'Bongo']
B = len(mylist) # This will return the length of the list which is 3. The index is 0, 1, 2, 3.
print mylist[1] # This will return the value at index 1, which is 'Grasshopper'
print mylist[0:2] # This will return the first 3 elements in the list.
```

You can assign data to a specific element of the list using an index into the list. The list index starts at zero. Data can be assigned to the elements of an array as follows:

```
mylist = [0, 1, 2, 3]
mylist[0] = 'Rhino'
mylist[1] = 'Grasshopper'
mylist[2] = 'Flamingo'
mylist[3] = 'Bongo'
print mylist[1]
```

Lists aren't limited to a single dimension. Although most people can't comprehend more than three or four dimensions. You can declare multiple dimensions by separating an with commas. In the following example, the MyTable variable is a two-dimensional array :

```
MyTable = [[], []]
```

In a two-dimensional array, the first number is always the number of rows; the second number is the number of columns.

Tuple

Tuples are a group of values like a list and are manipulated in similar ways. But, tuples are fixed in size once they are assigned. In Python the fixed size is considered immutable as compared to a list that is dynamic and mutable. Tuples are defined by parenthesis ().

```
myGroup = ('Rhino', 'Grasshopper', 'Flamingo', 'Bongo')
```

Here are some advantages of tuples over lists:

1. Elements to a tuple. Tuples have no append or extend method.
2. Elements cannot be removed from a tuple.
3. You can find elements in a tuple, since this doesn't change the tuple.
4. You can also use the in operator to check if an element exists in the tuple.
5. Tuples are faster than lists. If you're defining a constant set of values and all you're ever going to do with it is iterate through it, use a tuple instead of a list.
6. It makes your code safer if you "write-protect" data that does not need to be changed.

It seems tuples are very restrictive, so why are they useful? There are many datastructures in Rhino that require a fixed set of values. For instance a Rhino point is a list of 3 numbers `[34.5, 45.7, 0]`. If this is set as tuple, then you can be assured the original 3 number structure stays as a point `(34.5, 45.7, 0)`. There are other datastructures such as lines, vectors, domains and other data in Rhino that also require a certain set of values that do not change. Tuples are great for this.

Dictionary

Dictionaries in Python are lists of `Key`:`Value` pairs. This is a very powerful datatype to hold a lot of related information that can be associated through `keys`. The main operation of a dictionary is to extract a value based on the `key` name. Unlike lists, where index numbers are used, dictionaries allow the use of a `key` to access its

members. Dictionaries can also be used to sort, iterate and compare data.

Dictionaries are created by using braces ({}) with pairs separated by a comma (,) and the key values associated with a colon(:). In Dictionaries the `Key` must be unique. Here is a quick example on how dictionaries might be used:

```
room_num = {'john': 425, 'tom': 212}
room_num['john'] = 645  # set the value associated with the 'john' key to 645
print (room_num['tom']) # print the value of the 'tom' key.
room_num['isaac'] = 345 # Add a new key 'isaac' with the associated value
print (room_num.keys()) # print out a list of keys in the dictionary
print ('isaac' in room_num) # test to see if 'issac' is in the dictionary. This returns true.
```

Functions in Python

Python includes many built-in functions. These functions perform a predefined task and can be called upon in any program, as per requirement. However, if you don't find a suitable built-in function to serve your purpose, you can define one. We will now see how to define and use a function in a Python program.

Defining a Function

A function is a reusable block of programming statements designed to perform a certain task. To define a function, Python provides the def keyword. The following is the syntax of defining a function.

```
def function_name(parameters):
    """docstring"""
    statement1
    statement2
    ...
    ...
    return [expr]
```

The keyword def is followed by a suitable identifier as the name of the function and parentheses. One or more parameters may be optionally mentioned inside parentheses. The : symbol after parentheses starts an indented block.

The first statement in the function body can be a string, which is called the docstring. It explains the functionality of the function/class. The docstring is not mandatory.

The function body contains one or more statements that perform some actions. It can also use pass keyword.

Optionally, the last statement in the function block is the return statement. It sends an execution control back to calling the environment. If an expression is added in front of return, its value is also returned to the calling code.

The following example defines the greet() function.

Example: User-defined Function

```
def greet():
    """This function displays 'Hello World!'"""
    print('Hello World!')
```

Above, we have defined the greet() function. The first statement is a docstring that mentions what this function does. The second like is a print method that displays the specified string to the console. Note that it does not have the return statement.

To call a defined function, just use its name as a statement anywhere in the code. For example, the above function can be called using parenthesis, greet()

Example: Calling User-defined Function

```
greet()
```

Output

```
Hello World!
```

By default, all the functions return None if the return statement does not exist.

Example: Calling User-defined Function

```
val = greet()
print(val)
```

Output

None

The help() function displays the docstring, as shown below.

Example: Calling User-defined Function

```
>>> help(greet)
Help on function greet in module __main__:

    greet()
        This function displays 'Hello World!'
```

Function Parameters

It is possible to define a function to receive one or more parameters (also called arguments) and use them for processing inside the function block. Parameters/arguments may be given suitable formal names. The greet() function is now defined to receive a string parameter called name. Inside the function, the print() statement is modified to display the greeting message addressed to the received parameter.

Example: Parameterized Function

```
def greet(name):
    print ('Hello ', name)

greet('Steve') # calling function with argument
greet(123)
```

Output

```
Hello Steve
Hello 123
```

The names of the arguments used in the definition of the function are called formal arguments/parameters. Objects actually used while calling the function are called actual arguments/parameters.

The function parameters can have an annotation to specify the type of the parameter using parameter:type syntax. For example, the following annotates the parameter type string.

Example: Parameterized Function

```
def greet(name:str):
    print ('Hello ', name)

greet('Steve') # calling function with string argument
greet(123) # raise an error for int argument
```

Multiple Parameters

A function can have multiple parameters. The following function takes three arguments.

Example: Parameterized Function

```
def greet(name1, name2, name3):
    print ('Hello ', name1, ' , ', name2 , ', and ', name3)

greet('Steve', 'Bill', 'Yash') # calling function with string argument
```

Output

Hello Steve, Bill, and Yash

Unknown Number of Arguments

A function in Python can have an unknown number of arguments by putting * before the parameter if you don't know the number of arguments the user is going to pass.

Example: Parameterized Function

```
def greet(*names):
    print ('Hello ', names[0], ', ', names[1], ', ', names[3])

greet('Steve', 'Bill', 'Yash')
```

Output

Hello Steve, Bill, and Yash

The following function works with any number of arguments.

Example: Parameterized Function

```
def greet(*names):
        i=0
        print('Hello ', end='')
        while len(names) > i:
                print(names[i], end=', ')
```

```
i+=1
greet('Steve', 'Bill', 'Yash')
greet('Steve', 'Bill', 'Yash', 'Kapil', 'John', 'Amir')
```

Output

```
Hello Steve, Bill, Yash,
Hello Steve, Bill, Yash, Kapil, John, Amir
```

Function with Keyword Arguments

In order to call a function with arguments, the same number of actual arguments must be provided. However, a function can be called by passing parameter values using the parameter names in any order. For example, the following passes values using the parameter names.

```
def greet(firstname, lastname):
    print ('Hello', firstname, lastname)

greet(lastname='Jobs', firstname='Steve') # passing parameters in any order using keyword argument
```

Output

```
Hello Steve Jobs
```

Keyword Argument **kwarg

The function can have a single parameter prefixed with **. This type of parameter initialized to a new ordered mapping receiving any excess keyword arguments, defaulting to a new empty mapping of the same type.

Example: Parameterized Function

```
def greet(**person):

print('Hello ', person['firstname'],  person['lastname'])

greet(firstname='Steve', lastname='Jobs')
greet(lastname='Jobs', firstname='Steve')
greet(firstname='Bill', lastname='Gates', age=55)
greet(firstname='Bill') # raises KeyError
```

Output

```
Hello Steve Jobs
Hello Steve Jobs
Hello Bill Gates
```

When using the ** parameter, the order of arguments does not matter. However, the name of the arguments must be the same. Access the value of keyword arguments using paramter_name['keyword_argument'].

If the function access the keyword argument but the calling code does not pass that keyword argument, then it will raise the KeyError exception, as shown below.

Example: Parameterized Function

Copy

```
def greet(**person):
        print('Hello ', person['firstname'],  person['lastname'])

greet(firstname='Bill') # raises KeyError, must provide 'lastname'
```

arguement

Output

```
Traceback (most recent call last):
  File "<pyshell#21>", line 1, in <module>
    greet(firstname='Bill')
  File "<pyshell#19>", line 2, in greet
    print('Hello ', person['firstname'],  person['lastname'])
KeyError: 'lastname'
```

Parameter with Default Value

While defining a function, its parameters may be assigned default values. This default value gets substituted if an appropriate actual argument is passed when the function is called. However, if the actual argument is not provided, the default value will be used inside the function.

The following `greet()` function is defined with the `name` parameter having the default value `'Guest'`. It will be replaced only if some actual argument is passed.

Example: Parameter with Default Value

```
def greet(name = 'Guest'):
    print ('Hello', name)

greet()
greet('Steve')
```

Output

```
Hello Guest
Hello Steve
```

Function with Return Value

Most of the time, we need the result of the function to be used in further processes. Hence, when a function returns, it should also return a value.

A user-defined function can also be made to return a value to the calling environment by putting an expression in front of the return statement. In this case, the returned value has to be assigned to some variable.

Example: Return Value

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

The above function can be called and provided the value, as shown below.

Example: Parameter with Default Value

```
total=sum(10, 20)
print(total)
total=sum(5, sum(10, 20))
print(total)
```

Output

```
30
35
```

Strings in Python

A string is a built-in sequence data type. A string object is an ordered collection of Unicode characters put between single, double or triple quotes.

```
'Hello Python'
Out[1]:
'Hello Python'
In [2]:
#using double quotes
"Hello Python"
Out[2]:
'Hello Python'
In [3]:
#using triple quotes
'''
Hello World
Welcome to Python
'''
Out[3]:
'\nHello World\nWelcome to Python\n'
```

Note that even though double or triple quotes are used, Python uses single quotes for internal representation. Triple quotes are helpful in forming a string of more than one lines. Triple quotes may be of triple single quotes ('''…''') or triple double quotes ("""…"""). For each physical line the string includes a newline character \n. When the string object is displayed using print() statement, the effect newline characters is visible.

```
In [4]:
#using triple quotes
aString='''
Hello World
Welcome to Python
'''
print (aString)
Out[4]:
Hello World
Welcome to Python
```

The newline character \n is one of the escape sequences identified by Python. Escape sequence invokes an alternative implementation character subsequence to \. In Python \ is used as escape character. Following table shows list of escape sequences.

Escape sequence	Description	Example	Result
\a	Bell or alert	"\a"	Bell sound
\b	Backspace	"ab\bc"	ac
\f	Formfeed	"hello\fworld"	hello world
\n	Newline	"hello\nworld"	hello world
\nnn	Octal notation, where n is in the range 0-7	'\101'	A
\t	Tab	'Hello\tPython'	HelloPython
\xnn	Hexadecimal notation, where n is in the range 0-9, a-f, or A-F	'\x41'	A

String is also built using Python's built-in str() function.

In [5]:

str(10)

Out[5]:

'10'

All strings are objects of built-in str class. The str class has a number of methods to perform various operations on string. Let us discuss string methods and their uses.

Following group of string methods deal with rearrangement of casing of alphabets in a string.

capitalize()	Changes first letter of string to capital
lower()	All uppercase letters in string are converted to lowercase.
swapcase()	Changes the case of letter from upper to lower and vice versa.
title()	First letter of all words changes to uppercase and the rest are lowercase.

upper()	All lowercase letters in string changed to uppercase.

```
In [6]:
word='hello'
word.capitalize()
Out[6]:
'Hello'
In [7]:
line='HELLO WORLD'
line.lower()
Out[7]:
'hello world'
In [8]:
line='Hello World'
line.swapcase()
Out[8]:
'hELLO wORLD'
In [9]:
line='python is beautiful'
```

```
In [10]:
line='hello world'
line.upper()
Out[10]:
'HELLO WORLD'
```

Following group of methods are Boolean in nature. They return either True or False.

endswith()	Determines if string ends with given substring.
startswith()	Determines if string starts with given substring.
isalnum()	Returns true if all characters are alphanumeric and false otherwise.
isalpha()	Returns true if all characters are alphabetic and false otherwise.
isdigit()	Returns true if string contains only digits.
islower()	Returns true if string has all cased characters in lowercase and false otherwise.

istitle()	Returns true if each word in string starts with uppercase character and others in lowercase.
isupper()	Returns true if all characters in string are in uppercase and false otherwise.

```
In [11]:
line="How are you?"
line.endswith("?")
Out[11]:
True
In [12]:
line.startswith("How")
Out[12]:
True
In [13]:
string="AMD64"
string.isalnum()
Out[13]:
True
```

```
In [14]:
string.isalpha()
Out[14]:
False
In [15]:
string="102.556" # . is not a digit
string.isdigit()
Out[15]:
False
In [16]:
line="How are you"
line.islower()
Out[16]:
False
In [17]:
line.istitle()
Out[17]:
False
In [18]:
line.isupper()
```

find()	Determine if a substring occurs in given string. Returns index if found and -1 otherwise
index()	Same as find(), but raises an exception if substring not found.
rfind()	Same as find(), but search string in reverse direction.
rindex()	Same as index(), but search string in reverse direction.
replace()	Replaces all occurrences of a given substring and replace with another.

```
In [19]:
string="Simple is Better Than Complex"
string.find("an")
Out[19]:
19
In [20]:
string.find("be")
Out[20]:
```

```
-1
In [21]:
string.index("is")
Out[21]:
7
In [22]:
string.index('Is')
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-22-2f723114bb0b> in <module>()
----> 1 string.index('Is')

ValueError: substring not found
In [23]:
string="Simple is Better Than Complex"
string.replace("is","was")
Out[23]:
'Simple was Better Than Complex'
```

center()	Returns a string padded with given character so that original string is centered to a specified width.
expandtabs()	replaces tab escape sequence in string with multiple spaces. default tab size is 8.
ljust()	string is padded with spaces on left and left-justified to a specified width.
lstrip()	Removes leading whitespaces in string.
rjust()	string is padded with spaces on right and right-justified to a specified width.
rstrip()	Removes trailing whitespaces of string.
split()	Splits given string according to delimiter character or substring (default is space) and returns list of substrings.
strip()	Performs both lstrip() and rstrip() on string.

```
In [24]:
string="Computer"
string.center(30, "*")
Out[24]:
'**********Computer**********'
In [25]:
string.ljust(30,"*")
Out[25]:
'Computer**********************'
In [26]:
string.rjust(30,"*")
Out[26]:
'**********************Computer'
In [27]:
string = "Hello\tWorld"
string.expandtabs()
Out[27]:
'Hello   World'
In [28]:
string='    computer    '
```

```
string.lstrip()
Out[28]:
'computer    '
In [29]:
string.rstrip()
Out[29]:
'    computer'
In [30]:
string='192.168.001.001'
string.split('.')
Out[30]:
['192', '168', '001', '001']
```

String formatting

Python uses C style format specification symbols (%d, %f, %s etc) to construct a string by substituting these symbols by Python objects.

In the example below, the symbols %s and %d in the string are substituted by values of objects in tuple outside the string, prefixed by % symbol

```
In [31]:
name="Kiran"
marks=95
"Hi my name is %s and I have secured %d percent marks in my B.E exam" %(name, marks)
Out[31]:
'Hi my name is Kiran and I have secured 95 percent marks in my B.E exam'
```

Symbol	**Purpose**
%c	character
%s	string
%i	signed decimal integer
%d	signed decimal integer

%u	unsigned decimal integer
%o	octal integer
%x / %X	hexadecimal integer
%e / %E	exponential notation
%f	floating point real number

In numeric formatting symbols, width before and/or after decimal point can be specified.

```
In [32]:
x=10
y=1001.21
"x=%5d y=%10.3f" %(x,y)
Out[32]:
'x=   10 y= 1001.210'
```

By default string is aligned to left. To make it right aligned prefix – symbol to width in %s symbol.

```
In [33]:
string='computer'
"%30s"%(string,)
Out[33]:
'                      computer'
In [34]:
"%-30s"%(string,)
Out[34]:
'computer
```

Python 3.x has a format() method which is more efficient and elegant as far as formatting with variable substitution is concerned.

Instead of C like % formatting operator, {} symbols are used as place holders.

```
In [35]:
name="Kiran"
marks=95
"Hi my name is {} and I have secured {} percent marks in my B.E
```

```
exam".format(name, marks)
Out[35]:
'Hi my name is Kiran and I have secured 95 percent marks in my B.E exam'
```

Format specification characters are used with : instead of %. It means to insert a string in place holder use {:s} and for integer use {:d}. Width of numeric and string variables is specified as before.

```
In [36]:
x=10
y=1001.21
"x={:5d} y={:10.3f}".format(x,y)
Out[36]:
'x=   10 y= 1001.210'
```

Alignment of strings is formatted by <, > and ^ symbols in place holder. They make the substituted string left aligned, right aligned or center aligned respectively. Default is < for left alignment.

```
string='computer'
"{:<30s}".format(string,)
Out[37]:
```

```
'computer                    '
In [38]:
"{:30s}".format(string,) #default is < - left alignment
Out[38]:
'computer                    '
In [39]:
"{:^30s}".format(string,)
Out[39]:
'           computer         '
```

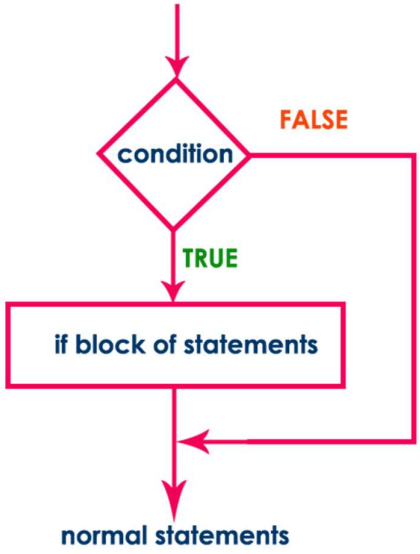
Decision Statements in Python

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

- if statement
- if-else statement
- if-elif statement

if statement in Python

In Python, we use the if statement to test a condition and decide the execution of a block of statements based on that condition result. The if statement checks, the given condition then decides the execution of a block of statements. If it is True, then the block of statements is executed and if it is False, then the block of statements is ignored. The execution flow of if statement is as follows.



The general syntax of if statement in Python is as follows.

```
if condition:
    Statement_1
    Statement_2
    Statement_3
    ...
```

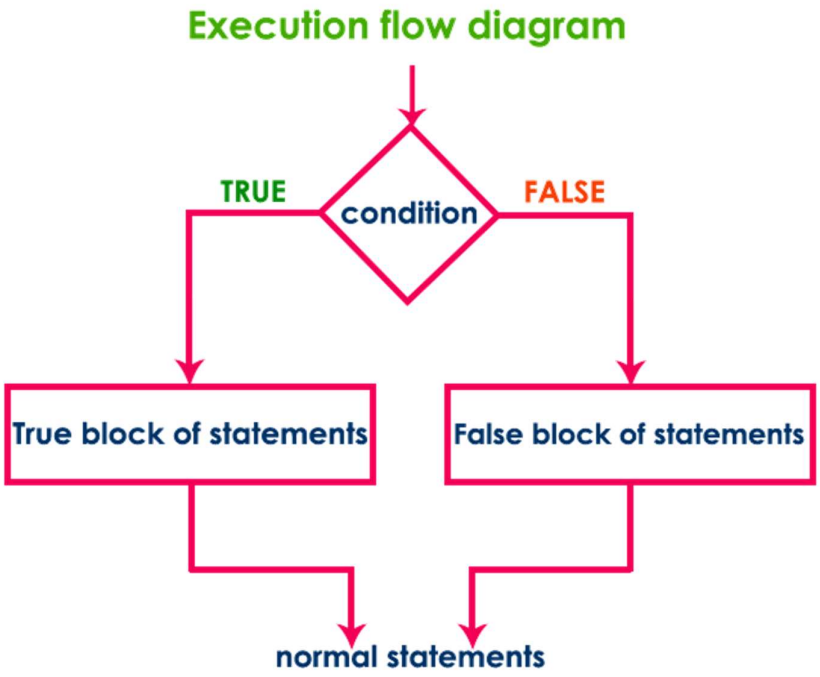
When we define an if statement, the block of statements must be specified using indentation only. The indentation is a series of white-spaces. Here, the number of white-spaces is variable, but all statements must use the identical number of white-spaces. Let's look at the following example Python code.

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

if-else statement in Python

In Python, we use the if-else statement to test a condition and pick the execution of a block of statements out of two blocks based on that condition result. The if-else statement checks the given condition then decides which block of statements to be executed based on the condition result. If the condition is True, then the true block of statements is

executed and if it is False, then the false block of statements is executed. The execution flow of if-else statement is as follows.



The general syntax of if-else statement in Python is as follows.

```
if condition:
        Statement_1
        Statement_2
        Statement_3
        ...
else:
        Statement_4
        Statement_5
        ...
```

In the above syntax, whenever the condition is True, the statements 1 2 and 3 are gets executed. And if the condition is False then the statements 4 and 5 are gets executed. Let's look at the following example of Python code.

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

elif statement in Python

In Python, When we want to test multiple conditions we use **elif** statement.

The general syntax of if-elif-else statement in Python is as follows.

```
if condition_1:
        Statement_1
        Statement_2
        Statement_3
        ...
elif condition_2:
        Statement_4
        Statement_5
```

```
Statement_6
    ...
else:
    Statement_7
    Statement_8
    ...
```

In the above syntax, whenever the condition_1 is True, the statements 1 2 and 3 are gets executed. If the condition_1 is False and condition_2 is True then the statements 4, 5, and 6 are gets executed. And if condition_1 nad Condition_2 both are False then the statements 7 and 8 are executed. Let's look at the following example of Python code.

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

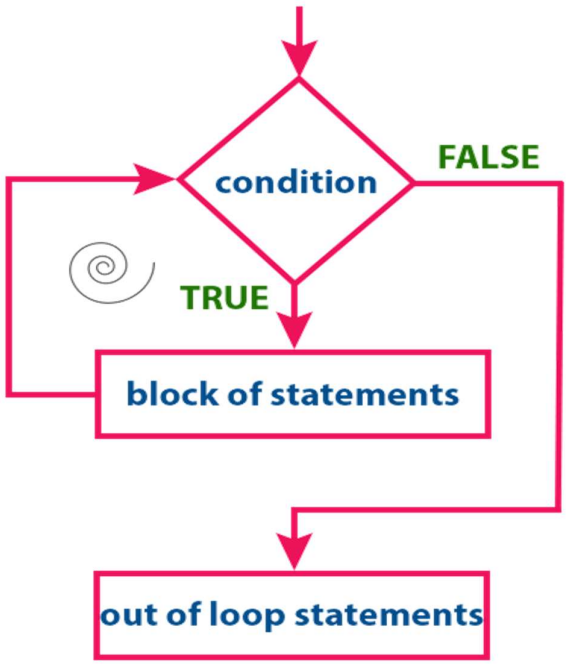
Loop Statement in Python

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

- while statement
- for statement

while statement

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



The general syntax of while statement in Python is as follows.

```
while condition:
    Statement_1
    Statement_2
    Statement_3
    ...
```

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

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

while statement with 'else' clause in Python

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

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

for statement in Python

In Python, the for statement is used to iterate through a sequence like a list, a tuple, a set, a dictionary, or a string. The for statement is used to repeat the execution of a set of statements for every element of a sequence.

The general syntax of for statement in Python is as follows.

for <variable> in <sequence>:

    Statement_1

    Statement_2

    Statement_3

In the above syntax, the **variable** is stored with each element from the **sequence** for every iteration.

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

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

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

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

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

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

for statement with 'else' clause in Python

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

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

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

Modules in Python

Any text file with the .py extension containing Python code is basically a module. Different Python objects such as functions, classes, variables, constants, etc., defined in one module can be made available to an interpreter session or another Python script by using the import statement. Functions defined in built-in modules need to be imported before use. On similar lines, a custom module may have one or more user-defined Python objects in it. These objects can be imported in the interpreter session or another script.

If the programming algorithm requires defining a lot of functions and classes, they are logically organized in modules. One module stores classes, functions and other resources of similar relevance. Such a modular structure of the code makes it easy to understand, use and maintain.

Creating a Module

Shown below is a Python script containing the definition of sum() function. It is saved as calc.py.

calc.py

```
def sum(x, y):
    return x + y
```

Importing a Module

We can now import this module and execute the sum() function in the Python shell.

Example: Importing a Module

```
>>> import calc
>>> calc.sum(5, 5)
10
```

In the same way, to use the above calc module in another Python script, use the import statement.

Every module, either built-in or custom made, is an object of a module class. Verify the type of different modules using the built-in type() function, as shown below.

Example: Module Type

```
>>> import math
>>> type(math)
<class 'module'>
>>> import calc
>>> type(calc)
<class 'module'>
```

Renaming the Imported Module

Use the as keyword to rename the imported module as shown below.

Example:

```
>>> import math as cal
>>> cal.log(4)
1.3862943611198906
```

from .. import statement

The above import statement will load all the resources of the module in the current working environment (also called namespace). It is possible to import specific objects from a module by using this syntax. For example, the following module calc.py has three functions in it.

calc.py
```
def sum(x,y):
    return x + y
def average(x, y):
    return (x + y)/2
def power(x, y):
    return x**y
```

Now, we can import one or more functions using the from...import statement. For example, the following code imports only two functions in the test.py.

Example: Importing Module's Functions
```
>>> from functions import sum, average
>>> sum(10, 20)
30
>>> average(10, 20)
15
>>> power(2, 4)
```

The following example imports only one function - sum.

Example: Importing Module's Function

```
>>> from functions import sum
>>> sum(10, 20)
30
>>> average(10, 20)
```

You can also import all of its functions using the `from...import *` syntax.

Example: Import Everythin from Module
Copy

```
>>> from functions import *
>>> sum(10, 20)
30
>>> average(10, 20)
15
>>> power(2, 2)
4
```

Object Oriented Programming

OOPs concepts in Python:

- Python Classes and Objects
- Inheritance
- Overloading
- Overriding
- Data hiding

Classes and Objects

- Python is an object-oriented programming language where programming stresses more on objects.
- Almost everything in Python is objects.

Classes
Class in Python is a collection of objects, we can think of a class as a blueprint or sketch or prototype. It contains all the details of an object.

In the real-world example, Animal is a class, because we have different kinds of Animals in the world and all of these are belongs to a class called Animal.

Defining a class

In Python, we should define a class using the keyword 'class'.

Syntax:
class classname:

#Collection of statements or functions or classes

Example:

```
class MyClass:
a = 10
b = 20
def add():
sum = a+b
print(sum)
```

In the above example, we have declared the class called 'Myclass' and we have declared and defined some variables and functions respectively.

To access those functions or variables present inside the class, we can use the class name by creating an object of it.

First, let's see how to access those using class name.

Example:

```
class MyClass:
        a = 10
        b = 20
 #Accessing variable present inside MyClass
print(MyClass.a)
```

Output

```
10
```

Objects

An object is usually an instance of a class. It is used to access everything present inside the class.

Creating an Object
Syntax:
variablename = classname

Example:

```
class MyClass:
    a = 10
    b = 20
    def add(self):
        sum = self.a + self.b
        print(sum)

#Creating an object of class MyClass
ob = MyClass()

#Accessing function and variables present inside MyClass using the obj
print(ob.a)
print(ob.b)
ob.add()
```

Output:

```
10
20
30
```

Constructor in Python

Constructor in Python is a special method which is used to initialize the members of a class during run-time when an object is created.

In Python, we have some special built-in class methods which start with a double underscore (__) and they have a special meaning in Python.

The name of the constructor will always be __init__().

Every class must have a constructor, even if you don't create a constructor explicitly it will create a default constructor by itself.

Example:

```
class MyClass:
        sum = 0

def __init__ (self, a, b):
      self.sum = a+b

def printSum(self):
       print("Sum of a and b is: ", self.sum)

#Creating an object of class MyClass
ob = MyClass(12, 15)
ob.printSum()
```

Output:

```
Sum of a and b is: 27
```

If we observe in the above example, we are not calling the __init__() method, because it will be called automatically when we create an object to that class and initialize the data members if any.

Always remember that a constructor will never return any values, hence it does not contain any return statements.

Inheritance

Inheritance is one of the most powerful concepts of OOPs.A class which inherits the properties of another class is called Inheritance.

The class which inherits the properties is called child class/subclass and the class from which properties are inherited is called parent class/base class.

Python provides three types of Inheritance:

- Single Inheritance
- Multilevel Inheritance
- Multiple Inheritance

Single Inheritance

In Single inheritance, one class will inherit the properties of one class only.

Example:

```
class Operations:
    a = 10
    b = 20
    def add(self):
        sum = self.a + self.b
        print("Sum of a and b is: ", sum)

class MyClass(Operations):
    c = 50
    d = 10
    def sub(self):
        sub = self.c – self.d
        print("Subtraction of c and d is: ", sub)
```

```
ob = MyClass()
ob.add()
ob.sub()
```

Output:

```
Sum of a and b is: 30
Subtraction of c and d is: 40
```

In the above example, we are inheriting the properties of the 'Operations' class into the class 'MyClass'.

Hence, we can access all the methods or statements present in the 'Operations' class by using the MyClass objects.

Multilevel Inheritance

In multilevel Inheritance, one or more class act as a base class.

Which means the second class will inherit the properties of the first class and the third class will inherit the properties of the second class. So the second class will act as both the Parent class as well as Child class.

```
class Addition:
    a = 10
    b = 20
    def add(self):
        sum = self.a + self.b
        print("Sum of a and b is: ", sum)
```

```
class Subtraction(Addition):
      def sub(self):
            sub = self.b-self.a
            print("Subtraction of a and b is: ", sub)

class Multiplication(Subtraction):
     def mul(self):
           multi = self.a * self.b
           print("Multiplication of a and b is: ", multi)

ob = Multiplication ()
ob.add()
ob.sub()
ob.mul()
```

Output:

```
Sum of a and b is: 30
Subtraction of a and b is: 10
Multiplication of a and b is: 200
```

In the above example, class 'Subtraction' inherits the properties of class 'Addition' and class 'Multiplication' will inherit the properties of class 'Subtraction'. So class 'Subtraction' will act as both Base class and derived class.

Multiple Inheritance

The class which inherits the properties of multiple classes is called Multiple Inheritance.

Example:

```
class Addition:
    a = 10
    b = 20
    def add(self):
        sum = self. a+ self.b
        print("Sum of a and b is: ", sum)

class Subtraction():
    c = 50
    d = 10
    def sub(self):
        sub = self.c-self.d
        print("Subtraction of c and d is: ", sub)

class Multiplication(Addition,Subtraction):
    def mul(self):
        multi = self.a * self.c
        print("Multiplication of a and c is: ", multi)

ob = Multiplication ()
ob.add()
ob.sub()
ob.mul()
```

Output:

```
Sum of a and b is: 30
Subtraction of c and d is: 10
Multiplication of a and c is: 500
```

Method Overloading in Python

Multiple methods with the same name but with a different type of parameter or a different number of parameters is called Method overloading

Example:

```
def product(a, b):
    p = a*b
    print(p)


def product(a, b, c):
    p = a*b*c
    print(p)


#Gives you an error saying one more argument is missing as it updated second function
#product(2, 3)
product(2, 3, 5)
```

Output:

```
30
```

Method overloading is not supported in Python, because if we see in the above example we have defined two functions with the same name 'product' but with a different number of parameters.

But in Python, the latest defined will get updated, hence the function product(a,b) will become useless.

Method Overriding in Python

If a subclass method has the same name which is declared in the superclass method then it is called Method overriding

To achieve method overriding we must use inheritance.

Example:

```
class A:

    def sayHi():

        print("I am in A")

class B(A):

    def sayHi():

        print("I am in B")

ob = B()

ob.sayHi()
```

Output:

I am in B

Data Hiding in Python

Data hiding means making the data private so that it will not be accessible to the other class members. It can be accessed only in the class where it is declared.

In python, if we want to hide the variable, then we need to write double underscore (__) before the variable name.

Example:

```
 Class MyClass:
        __num = 10
        def add(self, a):
              sum = self.__num + a
              print(sum)
ob = MyClass()
ob.add(20)
print(ob.__num)

#The above statement gives an error because we are trying to access
private variable outside the class
```

Output:

```
30
Traceback (most recent call last):
File “DataHiding.py”, line 10, in
print (ob.__num)
AttributeError: MyClass instance has
no attribute ‘__num
```

Methods in Python

Function

A function is a block of code to carry out a specific task, will contain its own scope and is called by name. All functions may contain zero(no) arguments or more than one arguments. On exit, a function can or can not return one or more values.

Basic function syntax

```
def functionName( arg1, arg2,….):
  …….
  # Function_body
  ……..
```

Let's create our own (user), a very simple function called sum(user can give any name he wants)". Function "sum" is having two arguments called num1 and num2 and will return the sum of the arguments passed to the function(sum). When we call the function (sum) with values(arguments) 5 and 6, it returns 11.

```
def sum(num1, num2):
  return (num1 + num2)
```

Output

```
>>> sum(5,6)
11
```

So from above, we see the 'return' statement returns a value from python function.

The function allows us to implement code reusability. There are three kinds of functions −

- Built-in functions (As the name suggests, these functions come with the Python language, for example, help() to ask for any help, max()- to get maximum value, type()- to return the type of an object and many more.)
- User-defined functions (These are the functions that users create to help them, like the "sum" function we have created above).
- Anonymous Functions (also called lambda functions and unlike normal function which is defined using ***def*** keyword are defined using ***lambda*** keyword).

Method

A method in python is somewhat similar to a function, except it is associated with object/classes. Methods in python are very similar to functions except for two major differences.

- The method is implicitly used for an object for which it is called.
- The method is accessible to data that is contained within the class.

General Method Syntax

```
class ClassName:
  def method_name():
    ……..
    # Method_body
    ……
```

Let's understand the method through one simple code −

```
class Pet(object):


 def my_method(self):
      print("I am a Cat")
cat = Pet()
cat.my_method()
```

Output

```
I am a Cat
```

In the above code, we first defined class “Pet”. Then we created the object “cat” from this blueprint. Next, we call our custom method called my_method with the object(.i.e. cat).

Key differences between method and function in python

As we get the basic understanding of the function and method both, let's highlight the key differences between them −

- Unlike a function, methods are called on an object. Like in our example above we call our method .i.e. “my_method” on the object “cat” whereas the function “sum” is called without any object. Also, because the method is called on an object, it can access that data within it.
- Unlike method which can alter the object’s state, python function doesn’t do this and normally operates on i

Exception Handling in Python

The cause of an exception is often external to the program itself. For example, an incorrect input, a malfunctioning IO device etc. Because the program abruptly terminates on encountering an exception, it may cause damage to system resources, such as files. Hence, the exceptions should be properly handled so that an abrupt termination of the program is prevented.

Python uses try and except keywords to handle exceptions. Both keywords are followed by indented blocks.

Syntax:

```
try :
    #statements in try block
except :
    #executed when error in try block
```

The try: block contains one or more statements which are likely to encounter an exception. If the statements in this block are executed without an exception, the subsequent except: block is skipped.

If the exception does occur, the program flow is transferred to the except: block. The statements in the except: block are meant to handle the cause of the exception appropriately. For example, returning an appropriate error message.

You can specify the type of exception after the except keyword. The subsequent block will be executed only if the specified exception occurs. There may be multiple except clauses with different exception types in a single try block. If the type of exception doesn't match any

of the except blocks, it will remain unhandled and the program will terminate.

The rest of the statements after the except block will continue to be executed, regardless if the exception is encountered or not.

The following example will throw an exception when we try to divide an integer by a string.

Example: try...except blocks

```
try:
    a=5
    b='0'
    print(a/b)
except:
    print('Some error occurred.')
print("Out of try except blocks.")
```

Output

```
Some error occurred.
Out of try except blocks.
```

You can mention a specific type of exception in front of the except keyword. The subsequent block will be executed only if the specified exception occurs. There may be multiple except clauses with different exception types in a single try block. If the type of exception doesn't match any of the except blocks, it will remain unhandled and the program will terminate.

Example: Catch Specific Error Type

```
try:
    a=5
    b='0'
    print (a+b)
except TypeError:
    print('Unsupported operation')
print ("Out of try except blocks")
```

Output

```
Unsupported operation
Out of try except blocks
```

As mentioned above, a single try block may have multiple except blocks. The following example uses two except blocks to process two different exception types:

Example: Multiple except Blocks

```
try:
    a=5
    b=0
    print (a/b)
except TypeError:
    print('Unsupported operation')
except ZeroDivisionError:
    print ('Division by zero not allowed')
print ('Out of try except blocks')
```

Output

```
Division by zero not allowed
Out of try except blocks
```

else and finally

In Python, keywords else and finally can also be used along with the try and except clauses. While the except block is executed if the exception occurs inside the try block, the else block gets processed if the try block is found to be exception free.

Syntax:

```
try:
    #statements in try block
except:
    #executed when error in try block
else:
    #executed if try block is error-free
finally:
    #executed irrespective of exception occured or not
```

The finally block consists of statements which should be processed regardless of an exception occurring in the try block or not. As a consequence, the error-free try block skips the except clause and enters the finally block before going on to execute the rest of the code. If, however, there's an exception in the try block, the appropriate except block will be processed, and the statements in the finally block will be processed before proceeding to the rest of the code.

The example below accepts two numbers from the user and performs their division. It demonstrates the uses of else and finally blocks.

Example: try, except, else, finally blocks

```
try:
    print('try block')
    x=int(input('Enter a number: '))
    y=int(input('Enter another number: '))
    z=x/y
except ZeroDivisionError:
    print("except ZeroDivisionError block")
    print("Division by 0 not accepted")
else:
    print("else block")
    print("Division = ", z)
finally:
    print("finally block")
    x=0
    y=0
print ("Out of try, except, else and finally blocks." )
```

The first run is a normal case. The out of the else and finally blocks is displayed because the try block is error-free.

Output

```
try block
Enter a number: 10
Enter another number: 2
else block
Division =  5.0
finally block
Out of try, except, else and finally blocks.
```

The second run is a case of division by zero, hence, the except block and the finally block are executed, but the else block is not executed.

Output
try block
Enter a number: 10
Enter another number: 0
except ZeroDivisionError block
Division by 0 not accepted
finally block
Out of try, except, else and finally blocks.

In the third run case, an uncaught exception occurs. The finally block is still executed but the program terminates and does not execute the program after the finally block.

Output
try block
Enter a number: 10
Enter another number: xyz
finally block
Traceback (most recent call last):
  File "C:\python36\codes\test.py", line 3, in <module>
    y=int(input('Enter another number: '))
ValueError: invalid literal for int() with base 10: 'xyz'

Raise an Exception

Python also provides the raise keyword to be used in the context of exception handling. It causes an exception to be generated explicitly. Built-in errors are raised implicitly. However, a built-in or custom exception can be forced during execution.

The following code accepts a number from the user. The try block raises a Value Error exception if the number is outside the allowed range.

Example: Raise an Exception

```
try:
    x=int(input('Enter a number upto 100: '))
    if x > 100:
        raise ValueError(x)
except ValueError:
    print(x, "is out of allowed range")
else:
    print(x, "is within the allowed range")
```

Output

```
Enter a number upto 100: 200
200 is out of allowed range
Enter a number upto 100: 50
50 is within the allowed range
```

File Handling in Python

File handling is basically the management of the files on a file system. Every operating system has its own way to store files.

Python File handling is useful to work with files in our programs. We don't have to worry about the underlying operating system and its file system rules and operations.

Opening and Closing Files

Until now, you have been reading and writing to the standard input and output. Now, we will see how to use actual data files.

Python provides basic functions and methods necessary to manipulate files by default. You can do most of the file manipulation using a **file** object.

The *open* Function

Before you can read or write a file, you have to open it using Python's built-in *open()* function. This function creates a **file** object, which would be utilized to call other support methods associated with it.

Syntax

file object = open(file_name [, access_mode][, buffering])

Here are parameter details −

- **file_name** − The file_name argument is a string value that contains the name of the file that you want to access.
- **access_mode** − The access_mode determines the mode in which the file has to be opened, i.e., read, write, append, etc. A complete list of possible values is given below in the table. This is optional parameter and the default file access mode is read (r).

- **buffering** − If the buffering value is set to 0, no buffering takes place. If the buffering value is 1, line buffering is performed while accessing a file. If you specify the buffering value as an integer greater than 1, then buffering action is performed with the indicated buffer size. If negative, the buffer size is the system default(default behavior).

Sr.No.	Modes & Description
1	**r** Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
2	**rb** Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file. This is the default mode.
3	**r+** Opens a file for both reading and writing. The file pointer placed at the beginning of the file.
4	**rb+** Opens a file for both reading and writing in binary format. The file pointer placed at the beginning of the file.

5	**w** Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
6	**wb** Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
7	**w+** Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
8	**wb+** Opens a file for both writing and reading in binary format. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
9	**a** Opens a file for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
10	**ab** Opens a file for appending in binary format. The file pointer is

	at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
11	**a+** Opens a file for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.
12	**ab+** Opens a file for both appending and reading in binary format. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.

The *file* Object Attributes

Once a file is opened and you have one *file* object, you can get various information related to that file.

Here is a list of all attributes related to file object −

Sr.No.	Attribute & Description
1	**file.closed** Returns true if file is closed, false otherwise.

2	**file.mode** Returns access mode with which file was opened.
3	**file.name** Returns name of the file.
4	**file.softspace** Returns false if space explicitly required with print, true otherwise.

```
#!/usr/bin/python

# Open a file
fo = open("foo.txt", "wb")
print "Name of the file: ", fo.name
print "Closed or not : ", fo.closed
print "Opening mode : ", fo.mode
print "Softspace flag : ", fo.softspace
```

```
Name of the file:  foo.txt
Closed or not :  False
Opening mode :  wb
Softspace flag :  0
```

The *close()* Method

The close() method of a *file* object flushes any unwritten information and closes the file object, after which no more writing can be done.

Python automatically closes a file when the reference object of a file is reassigned to another file. It is a good practice to use the close() method to close a file.

Syntax

fileObject.close()

Example

```
#!/usr/bin/python

# Open a file
fo = open("foo.txt", "wb")
print "Name of the file: ", fo.name

# Close opend file
fo.close()
```

This produces the following result −

Name of the file: foo.txt

Reading and Writing Files

The *file* object provides a set of access methods to make our lives easier. We would see how to use *read()* and *write()* methods to read and write files.

The *write()* Method

The *write()* method writes any string to an open file. It is important to note that Python strings can have binary data and not just text.

The write() method does not add a newline character ('\n') to the end of the string −

Syntax

fileObject.write(string)

Here, passed parameter is the content to be written into the opened file.

```
#!/usr/bin/python

# Open a file
fo = open("foo.txt", "wb")
fo.write( "Python is a great language.\nYeah its great!!\n")

# Close opend file
fo.close()
```

The above method would create *foo.txt* file and would write given content in that file and finally it would close that file. If you would open this file, it would have following content.

Python is a great language.
Yeah its great!!

The *read()* Method

The *read()* method reads a string from an open file. It is important to note that Python strings can have binary data. apart from text data.

Syntax

fileObject.read([count])

Here, passed parameter is the number of bytes to be read from the opened file. This method starts reading from the beginning of the file and if *count* is missing, then it tries to read as much as possible, maybe until the end of file.

Example

Let's take a file *foo.txt*, which we created above.

```
#!/usr/bin/python

# Open a file
fo = open("foo.txt", "r+")
str = fo.read(10);
print "Read String is : ", str
# Close opend file
fo.close()
```

This produces the following result −

```
Read String is :  Python is
```

File Positions

The *tell()* method tells you the current position within the file; in other words, the next read or write will occur at that many bytes from the beginning of the file.

The *seek(offset[, from])* method changes the current file position. The *offset* argument indicates the number of bytes to be moved. The *from* argument specifies the reference position from where the bytes are to be moved.

If *from* is set to 0, it means use the beginning of the file as the reference position and 1 means use the current position as the reference position and if it is set to 2 then the end of the file would be taken as the reference position.

Example

Let us take a file *foo.txt*, which we created above.

```
#!/usr/bin/python

# Open a file
fo = open("foo.txt", "r+")
str = fo.read(10)
print "Read String is : ", str

# Check current position
position = fo.tell()
print "Current file position : ", position

# Reposition pointer at the beginning once again
position = fo.seek(0, 0);
str = fo.read(10)
print "Again read String is : ", str
# Close opend file
fo.close()
```

This produces the following result −

```
Read String is :  Python is
Current file position :  10
Again read String is :  Python is
```

Renaming and Deleting Files

Python **os** module provides methods that help you perform file-processing operations, such as renaming and deleting files.

To use this module you need to import it first and then you can call any related functions.

The rename() Method

The *rename()* method takes two arguments, the current filename and the new filename.

Syntax

os.rename(current_file_name, new_file_name)

Example

Following is the example to rename an existing file *test1.txt* −

```
#!/usr/bin/python
import os

# Rename a file from test1.txt to test2.txt
os.rename( "test1.txt", "test2.txt" )
```

The *remove()* Method

You can use the *remove()* method to delete files by supplying the name of the file to be deleted as the argument.

Syntax

os.remove(file_name)

Example

Following is the example to delete an existing file *test2.txt* –

```
#!/usr/bin/python
import os

# Delete file test2.txt
os.remove("text2.txt")
```

Directories in Python

All files are contained within various directories, and Python has no problem handling these too. The **os** module has several methods that help you create, remove, and change directories.

The *mkdir()* Method

You can use the *mkdir()* method of the **os** module to create directories in the current directory. You need to supply an argument to this method which contains the name of the directory to be created.

Syntax

```
os.mkdir("newdir")
```

Example

Following is the example to create a directory *test* in the current directory −

```
#!/usr/bin/python
import os

# Create a directory "test"
os.mkdir("test")
```

The *chdir()* Method

You can use the *chdir()* method to change the current directory. The chdir() method takes an argument, which is the name of the directory that you want to make the current directory.

Syntax

os.chdir("newdir")

Example

Following is the example to go into "/home/newdir" directory −

```
#!/usr/bin/python
import os

# Changing a directory to "/home/newdir"
os.chdir("/home/newdir")
```

The *getcwd()* Method

The *getcwd()* method displays the current working directory.

Syntax

os.getcwd()

Example

Following is the example to give current directory −

```
#!/usr/bin/python
import os

# This would give location of the current directory
os.getcwd()
```

The *rmdir()* Method

The *rmdir()* method deletes the directory, which is passed as an argument in the method.

Before removing a directory, all the contents in it should be removed.

Syntax

os.rmdir('dirname')

Example

Following is the example to remove "/tmp/test" directory. It is required to give fully qualified name of the directory, otherwise it would search for that directory in the current directory.

```
#!/usr/bin/python
import os

# This would  remove "/tmp/test"  directory.
os.rmdir( "/tmp/test"  )
```