Python Classes/Objects

- Python is an object oriented programming language.
- Almost everything in Python is an object, with its properties and methods.
- A Class is like an object constructor, or a "blueprint" for creating objects.

Create a Class

To create a class, use the keyword class:

Example

Create a class named MyClass, with a property named x:

```
class MyClass:
  x = 5
print(MyClass)
```

Output:

Create Object

Now we can use the class named MyClass to create objects:

Example

Create an object named p1, and print the value of x:

```
class MyClass:
  x = 5
p1 = MyClass()
print(p1.x)
```

Output: 5

The __init__() Function

- All classes have a function called __init__(), which is always executed when the class is being initiated.

- Use the __init__() function to assign values to object properties, or other operations that are necessary to do when the object is being created:

Example

Create a class named Person, use the __init__() function to assign values for name and age:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age
p1 = Person("John", 36)
print(p1.name)
print(p1.age)
```

Output: John
36

Note: The __init__() function is called automatically every time the class is being used to create a new object.

Object Methods

- Objects can also contain methods. Methods in objects are functions that belong to the object. Let us create a method in the Person class:

Example

Insert a function that prints a greeting, and execute it on the p1 object:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age
  def myfunc(self):
    print("Hello my name is " + self.name)
p1 = Person("John", 36)
p1.myfunc()
```

Output: Hello my name is John

Note: The self parameter is a reference to the current instance of the class, and is used to access variables that belong to the class.

The self Parameter

- The `self` parameter is a reference to the current instance of the class, and is used to access variables that belongs to the class.

- It does not have to be named `self` , you can call it whatever you like, but it has to be the first parameter of any function in the class:

Example

Use the words *mysillyobject* and *abc* instead of *self*:

```
class Person:
  def __init__(mysillyobject, name, age):
    mysillyobject.name = name
    mysillyobject.age = age
  def myfunc(abc):
    print("Hello my name is " + abc.name)
p1 = Person("John", 36)
p1.myfunc()
```

Output: Hello my name is John

Modify Object Properties

You can modify properties on objects like this:

Example

Set the age of p1 to 40:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age
  def myfunc(self):
    print("Hello my name is " + self.name)
p1 = Person("John", 36)
p1.age = 40
print(p1.age)
```

Output: 40

Delete Object Properties

You can delete properties on objects by using the `del` keyword:

Example

Delete the age property from the p1 object:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age
  def myfunc(self):
    print("Hello my name is " + self.name)
p1 = Person("John", 36)
del p1.age
print(p1.age)
```

Output: [redacted]

Delete Objects

You can delete objects by using the `del` keyword:

Example

Delete the p1 object:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age
  def myfunc(self):
    print("Hello my name is " + self.name)
p1 = Person("John", 36)
del p1
print(p1)
```

Output:

```
Traceback (most recent call last):
  File "demo_class8.py", line 13, in <module>
    print(p1)
NameError: 'p1' is not defined
```

The pass Statement

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

Example

```
class Person:
  pass
# having an empty class definition like this, would raise an error without the pass statement.
```

Python Inheritance

Inheritance allows us to define a class that inherits all the methods and properties from another class.

- **Parent class** is the class being inherited from, also called base class.
- **Child class** is the class that inherits from another class, also called derived class.

Create a Parent Class

Any class can be a parent class, so the syntax is the same as creating any other class:

Example

Create a class named `Person`, with `firstname` and `lastname` properties, and a `printname` method:

```
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname
  def printname(self):
    print(self.firstname, self.lastname)
#Use the Person class to create an object, and then execute the printname method:
x = Person("John", "Doe")
x.printname()
```

Output: John Doe

Create a Child Class

To create a class that inherits the functionality from another class, send the parent class as a parameter when creating the child class:

Example

Create a class named `Student`, which will inherit the properties and methods from the `Person` class:

```
Class Student(Person):
        Pass
```

Note: Use the pass keyword when you do not want to add any other properties or methods to the class.

Now the Student class has the same properties and methods as the Person class.

Example

Use the `Student` class to create an object, and then execute the `printname` method:

```
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname
  def printname(self):
```

```
    print(self.firstname, self.lastname)
class Student(Person):
  pass
x = Student("Mike", "Olsen")
x.printname()
```

Output: Mike Olsen

Add the __init__() Function

So far we have created a child class that inherits the properties and methods from its parent. We want to add the `__init__()` function to the child class (instead of the `pass` keyword).

Note: The __init__() function is called automatically every time the class is being used to create a new object.

Example

Add the `__init__()` function to the `Student` class:

```
class Student(Person):
        def __init__(self, fname, lname):
        #add properties etc
```

When you add the `__init__()` function, the child class will no longer inherit the parent's `__init__()` function.

Note: The child's `__init__()` function **overrides** the inheritance of the parent's `__init__()` function.

To keep the inheritance of the parent's `__init__()` function, add a call to the parent's `__init__()` function:

Example

```
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname
  def printname(self):
    print(self.firstname, self.lastname)
class Student(Person):
  def __init__(self, fname, lname):
    Person.__init__(self, fname, lname)
x = Student("Mike", "Olsen")
x.printname()
```

Output: Mike Olsen

Now we have successfully added the __init__() function, and kept the inheritance of the parent class, and we are ready to add functionality in the `__init__()` function.

Use the super() Function

Python also has a `super()` function that will make the child class inherit all the methods and properties from its parent:

Example

```
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname
  def printname(self):
    print(self.firstname, self.lastname)
class Student(Person):
  def __init__(self, fname, lname):
    super().__init__(fname, lname)
x = Student("Mike", "Olsen")
x.printname()
```

Output: Mike Olsen

By using the `super()` function, you do not have to use the name of the parent element, it will automatically inherit the methods and properties from its parent.

Add Properties

Example

Add a property called `graduationyear` to the `Student` class:

```
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname
  def printname(self):
    print(self.firstname, self.lastname)
class Student(Person):
  def __init__(self, fname, lname):
    super().__init__(fname, lname)
    self.graduationyear = 2019
x = Student("Mike", "Olsen")
print(x.graduationyear)
```

Output: 2019

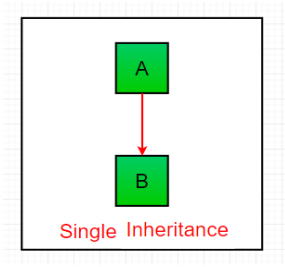
Inheritance

Inheritance is defined as the mechanism of inheriting the properties of the base class to the child class.

- Inheritance allows us to define a class that inherits all the methods and properties from another class.
- **Parent (Base/ Super) class** is the class that is being inherited from another class.
- **Child (Derived/ Sub) class** is the class that inherits from another class.
- In Python, every class whether built-in or user-defined is derived from the **object** class and all the objects are instances of the class **object**. Hence, **object class is the base class for all the other classes**.

Types of Inheritance in Python
Types of Inheritance depend upon the number of child and parent classes involved. There are four types of inheritance in Python:

Single Inheritance:
Single inheritance enables a derived class to inherit properties from a single parent class, thus enabling code reusability and the addition of new features to existing code.



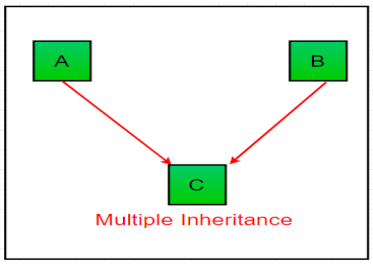
Example:

```
# Python program to demonstrate
# single inheritance
# Base class
class Parent:
        def func1(self):
                print("This function is in parent class.")

# Derived class
class Child(Parent):
        def func2(self):
                print("This function is in child class.")

# Driver's code
object = Child()
object.func1()
object.func2()
```

Output:

This function is in parent class.
This function is in child class.

Multiple Inheritance:
When a class can be derived from more than one base class this type of inheritance is called multiple inheritances. In multiple inheritances, all the features of the base classes are inherited into the derived class.



Example:

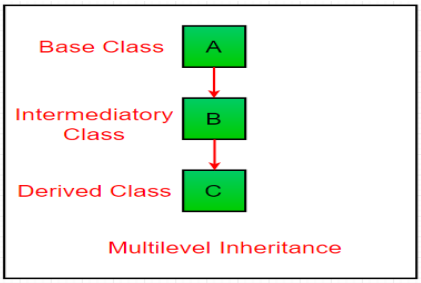
```
# Python program to demonstrate
# multiple inheritance
# Base class1
class Mother:
        mothername = ""
        def mother(self):
                print(self.mothername)

# Base class2
class Father:
        fathername = ""
        def father(self):
                print(self.fathername)

# Derived class
class Son(Mother, Father):
        def parents(self):
                print("Father :", self.fathername)
                print("Mother :", self.mothername)

# Driver's code
s1 = Son()
s1.fathername = "RAM"
s1.mothername = "SITA"
s1.parents()
```

Output:

```
Father : RAM
Mother : SITA
```

Multilevel Inheritance :

In multilevel inheritance, features of the base class and the derived class are further inherited into the new derived class. This is similar to a relationship representing a child and a grandfather.



Example:

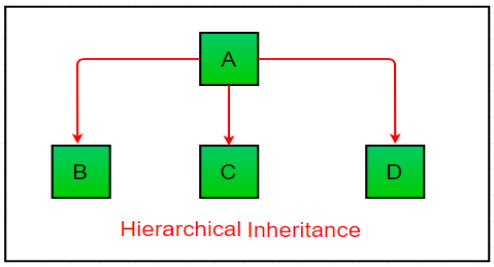
```
# Python program to demonstrate multilevel inheritance
# Base class
class Grandfather:
        def __init__(self, grandfathername):
                self.grandfathername = grandfathername

# Intermediate class
class Father(Grandfather):
        def __init__(self, fathername, grandfathername):
                self.fathername = fathername

                # invoking constructor of Grandfather class
                Grandfather.__init__(self, grandfathername)

# Derived class
class Son(Father):
        def __init__(self, sonname, fathername, grandfathername):
                self.sonname = sonname

                # invoking constructor of Father class
                Father.__init__(self, fathername, grandfathername)
        def print_name(self):
                print('Grandfather name :', self.grandfathername)
                print("Father name :", self.fathername)
                print("Son name :", self.sonname)

# Driver code
s1 = Son('Prince', 'Rampal', 'Lal mani')
print(s1.grandfathername)
s1.print_name()
```

Output:

```
Lal mani
```

Grandfather name : Lal mani
Father name : Rampal
Son name : Prince

Hierarchical Inheritance:
When more than one derived class are created from a single base this type of inheritance is called hierarchical inheritance. In this program, we have a parent (base) class and two child (derived) classes.



Example:

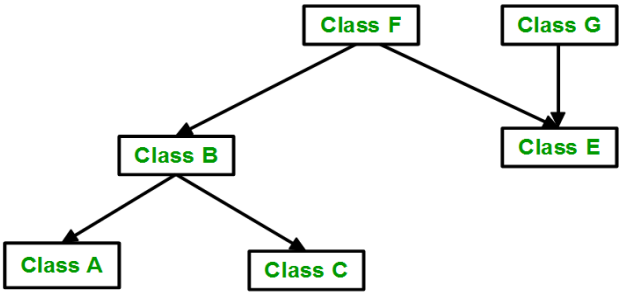
```
# Python program to demonstrate
# Hierarchical inheritance
# Base class

class Parent:
        def func1(self):
                print("This function is in parent class.")

# Derived class1
class Child1(Parent):
        def func2(self):
                print("This function is in child 1.")

# Derivied class2
class Child2(Parent):
        def func3(self):
                print("This function is in child 2.")

# Driver's code
object1 = Child1()
object2 = Child2()
object1.func1()
object1.func2()
object2.func1()
object2.func3()
```

Output:

```
This function is in parent class.
This function is in child 1.
```

This function is in parent class.
This function is in child 2.

Hybrid Inheritance:
Inheritance consisting of multiple types of inheritance is called hybrid inheritance.



Example:

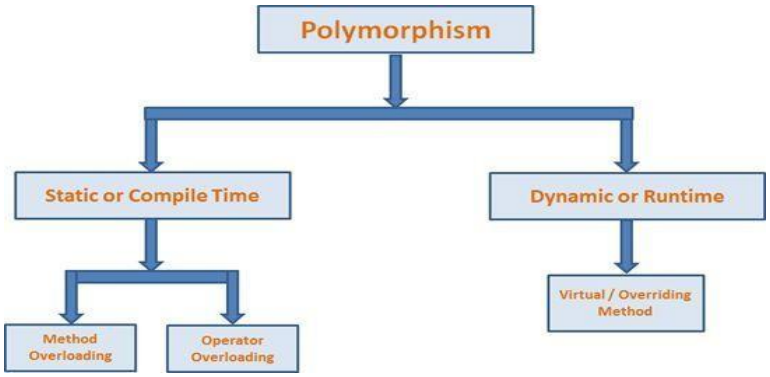
```
# Python program to demonstrate
# hybrid inheritance

class School:
        def func1(self):
                print("This function is in school.")
class Student1(School):
        def func2(self):
                print("This function is in student 1. ")
class Student2(School):
        def func3(self):
                print("This function is in student 2.")
class Student3(Student1, School):
        def func4(self):
                print("This function is in student 3.")

# Driver's code
object = Student3()
object.func1()
object.func2()
```

Output:
This function is in school.
This function is in student 1.

Polymorphism

The word polymorphism means having many forms. In programming, polymorphism means the same function name (but different signatures) being used for different types. The key difference is the data types and number of arguments used in function.

Types of Polymorphism



Method Overloading

Method overloading means same function name being used differently (different signatures).

Example of inbuilt polymorphic functions:

```
# Pre-defined polymorphic function
print(len("geeks"))          # len() being used for a string
print(len([10, 20, 30]))     # len() being used for a list
```

Output:

5
3

Examples of user-defined polymorphic functions:

```
def add(x, y, z = 0):
        return x + y+z

# Driver code
print(add(2, 3))
print(add(2, 3, 4))
```

Output:

5
9

Polymorphism with class methods:

The below code shows how Python can use two different class types, in the same way. We create a for loop that iterates through a tuple of objects. Then call the methods without being concerned about which class type each object is. We assume that these methods actually exist in each class.

```
class India():
        def capital(self):
                print("New Delhi is the capital of India.")
        def language(self):
                print("Hindi is the most widely spoken language of India.")
        def type(self):
                print("India is a developing country.")
```

```
class USA():
        def capital(self):
                print("Washington, D.C. is the capital of USA.")
        def language(self):
                print("English is the primary language of USA.")
        def type(self):
                print("USA is a developed country.")
obj_ind = India()
obj_usa = USA()
for country in (obj_ind, obj_usa):
        country.capital()
        country.language()
        country.type()
```

Output:

```
New Delhi is the capital of India.
Hindi is the most widely spoken language of India.
India is a developing country.
Washington, D.C. is the capital of USA.
English is the primary language of USA.
USA is a developed country.
```

Method Overriding

In Python, Polymorphism lets us define methods in the child class that have the same name as the methods in the parent class. In inheritance, the child class inherits the methods from the parent class. However, it is possible to modify a method in a child class that it has inherited from the parent class. This is particularly useful in cases where the method inherited from the parent class doesn't quite fit the child class. In such cases, we re-implement the method in the child class. This process of re-implementing a method in the child class is known as **Method Overriding**.

```
class Bird:
def intro(self):
        print("There are many types of birds.")
def flight(self):
        print("Most of the birds can fly but some cannot.")
class sparrow(Bird):
def flight(self):
        print("Sparrows can fly.")
class ostrich(Bird):
def flight(self):
        print("Ostriches cannot fly.")
obj_bird = Bird()
obj_spr = sparrow()
obj_ost = ostrich()
obj_bird.intro()
obj_bird.flight()
```

```
obj_spr.intro()
obj_spr.flight()
obj_ost.intro()
obj_ost.flight()
```

Output:

```
There are many types of birds.
Most of the birds can fly but some cannot.
There are many types of birds.
Sparrows can fly.
There are many types of birds.
Ostriches cannot fly.
```

Operator Overloading

- Operator overloading is the process of using an operator in different ways depending on the operands.
- Python has some magic methods or special functions for operator overloading.

UNARY OPERATORS	MAGIC METHOD
–	__NEG__(SELF, OTHER)
+	__POS__(SELF, OTHER)
~	__INVERT__(SELF, OTHER)

BINARY OPERATORS	MAGIC METHOD
+	__add__(self, other)
–	__sub__(self, other)
*	__mul__(self, other)
/	__truediv__(self, other)
//	__floordiv__(self, other)
%	__mod__(self, other)
**	__pow__(self, other)
>>	__rshift__(self, other)
<<	__lshift__(self, other)
&	__and__(self, other)
|	__or__(self, other)
^	__xor__(self, other)

Example:

```
#Using + and * for different purpose
print(3+4)                          #Adds two integers
print("MMMUT" + "Gorakhpur")        #Concatenate two strings

print(3*4)                          #Multiplies two numbers
print("MMMUT" * 4)                  #Repeats the strings


#Overloading a binary operator
Class A:
    def __init__(self , a):
        self.a = a
    def __add__(self , o):
        return self.a + self.o
obj1 = A(1)
obj2 = A(2)
obj3 = A("MMMUT")
obj4 = A("Gorakhpur")
print(obj1 + obj2)
print(obj3 + obj4)
```

Encapsulation

Encapsulation is also an essential aspect of object-oriented programming. It is used to restrict access to methods and variables. In encapsulation, code and data are wrapped together within a single unit from being modified by accident.

Implement Encapsulation with a Class in Python

Another example of Encapsulation can be a class because a class combines data and methods into a single unit. Here, the custom function demofunc() displays the records of students wherein we can access public data member. Using the objects st1, st2, st3, st4, we have access ed the public methods of the class demofunc() −

Example

```
class Students:
  def __init__(self, name, rank, points):
    self.name = name
    self.rank = rank
    self.points = points

  # custom function
  def demofunc(self):
    print("I am "+self.name)
    print("I got Rank ",+self.rank)
```

```
# create 4 objects
st1 = Students("Steve", 1, 100)
st2 = Students("Chris", 2, 90)
st3 = Students("Mark", 3, 76)
st4 = Students("Kate", 4, 60)

# call the functions using the objects created above
st1.demofunc()
st2.demofunc()
st3.demofunc()
st4.demofunc()
```

Output

```
I am Steve
I got Rank  1
I am Chris
I got Rank  2
I am Mark
I got Rank  3
I am Kate
I got Rank  4
```

Exception Handling

- The `try` block lets you test a block of code for errors.
- The `except` block lets you handle the error.
- The `else` block lets you execute code when there is no error.
- The `finally` block lets you execute code, regardless of the result of the try- and except blocks.

When an error occurs, or exception as we call it, Python will normally stop and generate an error message.
These exceptions can be handled using the try statement:

Example

The `try` block will generate an exception, because `x` is not defined:

```
#The try block will generate an error, because x is not defined:

try:
  print(x)
except:
  print("An exception occurred")
```

Output: An exception occurred

Since the try block raises an error, the except block will be executed. Without the try block, the program will crash and raise an error:

Many Exceptions

You can define as many exception blocks as you want, e.g. if you want to execute a special block of code for a special kind of error:

Example

Print one message if the try block raises a `NameError` and another for other errors:

```
#The try block will generate a NameError, because x is not defined:
try:
  print(x)
except NameError:
  print("Variable x is not defined")
except:
  print("Something else went wrong")
```

Output: Variable x is not defined

Else

You can use the `else` keyword to define a block of code to be executed if no errors were raised:

Example

In this example, the `try` block does not generate any error:

```
#The try block does not raise any errors, so the else block is executed:
try:
  print("Hello")
except:
  print("Something went wrong")
else:
  print("Nothing went wrong")
```

Output:

```
Hello
Nothing went wrong
```

Finally

The `finally` block, if specified, will be executed regardless if the try block raises an error or not.

```
#The finally block gets executed no matter if the try block raises any errors or not:
try:
  print(x)
except:
  print("Something went wrong")
finally:
  print("The 'try except' is finished")
```

Output:

Something went wrong
The ‘try except’ is finished

This can be useful to close objects and clean up resources:

Example
Try to open and write to a file that is not writable:

```
#The try block will raise an error when trying to write to a read-only file:
try:
  f = open("demofile.txt")
  try:
    f.write("Lorum Ipsum")
  except:
    print("Something went wrong when writing to the file")
  finally:
    f.close()
except:
  print("Something went wrong when opening the file")
```

Output:
Something went wrong when writing to the file
The program can continue, without leaving the file object open.

Raise an exception

As a Python developer you can choose to throw an exception if a condition occurs. To throw (or raise) an exception, use the `raise` keyword.

Example

Raise an error and stop the program if x is lower than 0:

```
x = -1
if x < 0:
  raise Exception("Sorry, no numbers below zero")
```

The raise keyword is used to raise an exception. You can define what kind of error to raise, and the text to print to the user.

File Handling

Till now, we were taking the input from the console and writing it back to the console to interact with the user. Sometimes, it is not enough to only display the data on the console. The data to be displayed may be very large, and only a limited amount of data can be displayed on the console since the memory is volatile, it is impossible to recover the programmatically generated data again and again.

The file handling plays an important role when the data needs to be stored permanently into the file. A file is a named location on disk to store related information. We can access the stored information (non-volatile) after the program termination. The file-handling implementation is slightly lengthy or complicated in the other programming language, but it is easier and shorter in Python.

How Python Handle Files?

If you are working in a large software application where they process a large number of data, then we cannot expect those data to be stored in a variable as the variables are volatile in nature. Hence when are you about to handle such situations, the role of files will come into the picture. As files are non-volatile in nature, the data will be stored permanently in a secondary device like Hard Disk and using python we will handle these files in our applications.

Are you thinking about how python will handle files?

Let's take an Example of how normal people will handle the files. If we want to read the data from a file or write the data into a file, then, first of all, we will open the file or will create a new file if the file does not exist and then perform the normal read/write operations, save the file and close it. Similarly, we do the same operations in python using some in-built methods or functions.

Types Of File in Python

There are two types of files in Python and each of them are explained below in detail with examples for your easy understanding.

They are:

- Binary file (audio, vedio, image etc.)
- Text file (txt file)

In Python, files are treated in two modes as text or binary. The file may be in the text or binary format, and each line of a file is ended with the special character.

Python File Handling Operations

Hence, a file operation can be done in the following order.

- Open a file
- Read or write - Performing operation
- Close the file

Opening a file

Python provides an **open()** function that accepts two arguments, file name and access mode in which the file is accessed. The function returns a file object which can be used to perform various operations like reading, writing, etc.

Syntax: file object = open(<file-name>, <access-mode>, <buffering>)

The files can be accessed using various modes like read, write, or append. The following are the details about the access mode to open a file.

Where the following mode is supported:

1. **r:** open an existing file for a read operation.
2. **w:** open an existing file for a write operation. If the file already contains some data then it will be overridden but if the file is not present then it creates the file as well.
3. **a:** open an existing file for append operation. It won't override existing data.
4. **r+:** To read and write data into the file. The previous data in the file will be overridden.
5. **w+:** To write and read data. It will override existing data.
6. **a+:** To append and read data from the file. It won't override existing data.

SN	Access mode	Description
1	r	It opens the file to read-only mode. The file pointer exists at the beginning. The file is by default open in this mode if no access mode is passed.
2	rb	It opens the file to read-only in binary format. The file pointer exists at the beginning of the file.
3	r+	It opens the file to read and write both. The file pointer exists at the beginning of the file.
5	w	It opens the file to write only. It overwrites the file if previously exists or creates a new one if no file exists with the same name. The file pointer exists at the beginning of the file.
6	wb	It opens the file to write only in binary format. It overwrites the file if it exists previously or creates a new one if no file exists. The file pointer exists at the beginning of the file.
7	w+	It opens the file to write and read both. It is different from r+ in the sense that it overwrites the previous file if one exists whereas r+ doesn't overwrite the previously written file. It creates a new file if no file exists. The file pointer exists at the beginning of the file.
8	wb+	It opens the file to write and read both in binary format. The file pointer exists at the beginning of the file.
9	a	It opens the file in the append mode. The file pointer exists at the end of the previously written file if exists any. It creates a new file if no file exists with the same name.
10	ab	It opens the file in the append mode in binary format. The pointer exists at the end of the previously written file. It creates a new file in binary format if no file exists with the same name.
11	a+	It opens a file to append and read both. The file pointer remains at the end of the file if a file exists. It creates a new file if no file exists with the same name.

Let's look at the simple example to open a file named "file.txt" (stored in the same directory) in read mode and printing its content on the console.

Example:

```
#opens the file file.txt in read mode
fileptr = open("file.txt","r")
if fileptr:
   print("file is opened successfully")
```

In the above code, we have passed **filename** as a first argument and opened file in read mode as we mentioned **r** as the second argument. The **fileptr** holds the file object and if the file is opened successfully, it will execute the print statement.

The close() method

Once all the operations are done on the file, we must close it using the **close()** method. Any unwritten information gets destroyed once the **close()** method is called on a file object.

We can perform any operation on the file externally using the file system which is the currently opened in Python; hence it is good practice to close the file once all the operations are done. The syntax to use the **close()** method is given below.

Syntax: fileobject.close()

Example:

```
# opens the file file.txt in read mode
fileptr = open("file.txt","r")
if fileptr:
   print("file is opened successfully")
#closes the opened file
fileptr.close()
```

After closing the file, we cannot perform any operation in the file. The file needs to be properly closed. If any exception occurs while performing some operations in the file then the program terminates without closing the file.

We should use the following method to overcome such type of problem.

```
try:
  fileptr = open("file.txt")
  # perform file operations
finally:
  fileptr.close()
```

The with statement

The with statement is useful in the case of manipulating the files. It is used in the scenario where a pair of statements is to be executed with a block of code in between.

Syntax: with open(<file name>, <access mode>) as <file-pointer>:

```
#statement suite
```

The advantage of using with statement is that it provides the guarantee to close the file regardless of how the nested block exits. It is always suggestible to use the **with** statement in the case of files because, if the break, return, or exception occurs in the nested block of code then it automatically closes the file, we don't need to write the **close()** function. It doesn't let the file to corrupt.

Example:

```
with open("file.txt",'r') as f:
  content = f.read();
  print(content)
```

Writing into the file

To write some text to a file, we need to open the file using the open method with one of the following access modes.
w: It will overwrite the file if any file exists. The file pointer is at the beginning of the file.
a: It will append the existing file. The file pointer is at the end of the file. It creates a new file if no file exists.

Example:

```
# open the file.txt in append mode. Create a new file if no such file exists.
fileptr = open("file2.txt", "w")
# appending the content to the file
fileptr.write('''Python is the modern day language. It makes things so simple.
It is the fastest-growing programing language''')
# closing the opened the file
fileptr.close()
```

We have opened the file in **w** mode. The **file1.txt** file doesn't exist, it created a new file and we have written the content in the file using the **write()** function.

```
#open the file.txt in write mode.
fileptr = open("file2.txt","a")
#overwriting the content of the file
fileptr.write(" Python has an easy syntax and user-friendly interaction.")
#closing the opened file
fileptr.close()
```

We can see that the content of the file is modified. We have opened the file in **a** mode and it appended the content in the existing **file2.txt**.

To read a file using the Python script, the Python provides the **read()** method. The **read()** method reads a string from the file. It can read the data in the text as well as a binary format.

The syntax of the **read()** method is given below.

Syntax: fileobj.read(<count>)

Here, the count is the number of bytes to be read from the file starting from the beginning of the file. If the count is not specified, then it may read the content of the file until the end.

Example:

```
#open the file.txt in read mode. causes error if no such file exists.
fileptr = open("file2.txt","r")
#stores all the data of the file into the variable content
content = fileptr.read(10)
# prints the type of the data stored in the file
print(type(content))
#prints the content of the file
print(content)
#closes the opened file
fileptr.close()
```

In the above code, we have read the content of **file2.txt** by using the **read()** function. We have passed count value as ten which means it will read the first ten characters from the file.

If we use the following line, then it will print all content of the file.

```
content = fileptr.read()
print(content)
```

Read file through for loop

We can read the file using for loop. Consider the following example.

```
#open the file.txt in read mode. causes an error if no such file exists.
fileptr = open("file2.txt","r");
#running a for loop
for i in fileptr:
   print(i) # i contains each line of the file
```

Read Lines of the file

Python facilitates to read the file line by line by using a function **readline()** method. The **readline()** method reads the lines of the file from the beginning, i.e., if we use the readline() method two times, then we can get the first two lines of the file.

Consider the following example which contains a function **readline()** that reads the first line of our file **"file2.txt"** containing three lines. Consider the following example.

Example 1: Reading lines using readline() function

```
#open the file.txt in read mode. causes error if no such file exists.
fileptr = open("file2.txt","r");
#stores all the data of the file into the variable content
content = fileptr.readline()
content1 = fileptr.readline()
#prints the content of the file
print(content)
print(content1)
#closes the opened file
fileptr.close()
```

We called the **readline()** function two times that's why it read two lines from the file.
Python provides also the **readlines()** method which is used for the reading lines. It returns the list of the lines till the end of **file(EOF)** is reached.

Example 2: Reading Lines Using readlines() function

```
#open the file.txt in read mode. causes error if no such file exists.
fileptr = open("file2.txt","r");
#stores all the data of the file into the variable content
content = fileptr.readlines()
#prints the content of the file
print(content)
#closes the opened file
fileptr.close()
```

Creating a new file

The new file can be created by using one of the following access modes with the function open().

x: it creates a new file with the specified name. It causes an error a file exists with the same name.
a: It creates a new file with the specified name if no such file exists. It appends the content to the file if the file already exists with the specified name.
w: It creates a new file with the specified name if no such file exists. It overwrites the existing file.

Example:

```
#open the file.txt in read mode. causes error if no such file exists.
fileptr = open("file2.txt","x")
print(fileptr)
if fileptr:
```

```
print("File created successfully")
```

File Pointer positions

Python provides the tell() method which is used to print the byte number at which the file pointer currently exists.

Example:

```
# open the file file2.txt in read mode
fileptr = open("file2.txt","r")
#initially the filepointer is at 0
print("The filepointer is at byte :",fileptr.tell())
#reading the content of the file
content = fileptr.read();
#after the read operation file pointer modifies. tell() returns the location of the fileptr.
print("After reading, the filepointer is at:",fileptr.tell())
```

Modifying file pointer position

In real-world applications, sometimes we need to change the file pointer location externally since we may need to read or write the content at various locations. For this purpose, the Python provides us the seek() method which enables us to modify the file pointer position externally.

Syntax: <file-ptr>.seek(offset[, from)

The seek() method accepts two parameters:
offset: It refers to the new position of the file pointer within the file.

from: It indicates the reference position from where the bytes are to be moved. If it is set to 0, the beginning of the file is used as the reference position. If it is set to 1, the current position of the file pointer is used as the reference position. If it is set to 2, the end of the file pointer is used as the reference position.

Example:

```
# open the file file2.txt in read mode
fileptr = open("file2.txt","r")
#initially the filepointer is at 0
print("The filepointer is at byte :",fileptr.tell())
#changing the file pointer location to 10.
fileptr.seek(10);
#tell() returns the location of the fileptr.
print("After reading, the filepointer is at:",fileptr.tell())
```

Python OS module

Renaming the file

The Python **os** module enables interaction with the operating system. The os module provides the functions that are involved in file processing operations like renaming, deleting, etc. It provides us the rename() method to rename the specified file to a new name. The syntax to use the **rename()** method is given below.

Syntax: rename(current-name, new-name)

The first argument is the current file name and the second argument is the modified name. We can change the file name bypassing these two arguments.

Example:

```
import os
#rename file2.txt to file3.txt
os.rename("file2.txt","file3.txt")
```

The above code renamed current **file2.txt** to **file3.txt**

Removing the file

The os module provides the **remove()** method which is used to remove the specified file. The syntax to use the **remove()** method is given below.

Syntax: remove(file-name)

Example:

```
import os;
#deleting the file named file3.txt
os.remove("file3.txt")
```

Creating the new directory

The **mkdir()** method is used to create the directories in the current working directory. The syntax to create the new directory is given below.

Syntax: mkdir(directory name)

Example:

```
import os
#creating a new directory with the name new
os.mkdir("new")
```

The getcwd() method

This method returns the current working directory. The syntax to use the getcwd() method is given below.

Syntax: os.getcwd()

Example:

```
import os
os.getcwd()
```

Changing the current working directory

The chdir() method is used to change the current working directory to a specified directory. The syntax to use the chdir() method is given below.

Syntax: chdir("new-directory")

Example:

```
import os
# Changing current directory with the new directiory
os.chdir("C:\\Users\\DEVANSH SHARMA\\Documents")
#It will display the current working directory
os.getcwd()
```

Deleting directory

The rmdir() method is used to delete the specified directory. The syntax to use the rmdir() method is given below.

Syntax: os.rmdir(directory name)

Example:

```
import os
#removing the new directory
os.rmdir("directory_name")
```

It will remove the specified directory.

The file related methods

The file object provides the following methods to manipulate the files on various operating systems.

SN	Method	Description
1	file.close()	It closes the opened file. The file once closed, it can't be read or write anymore.
2	File.fileno()	It returns the file descriptor used by the underlying implementation to request I/O from the OS.
3	File.next()	It returns the next line from the file.
4	File.read([size])	It reads the file for the specified size.
5	File.readline([size])	It reads one line from the file and places the file pointer to the beginning of the new line.

6	File.readlines([sizehint])	It returns a list containing all the lines of the file. It reads the file until the EOF occurs using readline() function.
7	File.seek(offset[,from)	It modifies the position of the file pointer to a specified offset with the specified reference.
8	File.tell()	It returns the current position of the file pointer within the file.
9	File.write(str)	It writes the specified string to a file
10	File.writelines(seq)	It writes a sequence of the strings to a file.

Modules in Python

A document with definitions of functions and various statements written in Python is called a Python module.
Modules in Python can be of two types:

- Built-in Modules.
- User-defined Modules.

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.

Let's construct a module. Save the file as example_module.py after entering the following.

Code

```
# 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

Code: 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:

Code:

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

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.

Code

```
# Python program to show how to import a standard module
# We will import the math module which is a standard module
import math
print( "The value of euler's number is", math.e )
```

Importing and also Renaming

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

Code:

```
# Python program to show how to import a module and rename it
# We will import the math module and give a different name to it
import math as mt
print( "The value of euler's number is", mt.e )
```

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.

Code:

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

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:

Code:

```
# 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 )
```

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.

Code:

```
# 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 ma
th module.
```