Object

The object is an entity that has state and behaviour. It may be any real-world object like the mouse, keyboard, chair, table, pen, etc. Everything in Python is an object, and almost everything has attributes and methods.

Class

The class can be defined as a collection of objects. It is a logical entity that has some specific attributes and methods. For example: if you have an employee class then it should contain an attribute and method, i.e. an email id, name, age, salary, etc.

Inheritance

It specifies that the child object acquires all the properties and behaviours of the parent object. By using inheritance, we can create a class which uses all the properties and behaviour of another class. The new class is known as a derived class or child class, and the one whose properties are acquired is known as a base class or parent class. It provides re-usability of the code.

Polymorphism

Poly means many and Morphs means form, shape. By polymorphism, we understand that one task can be performed in different ways. For example You have a class animal, and all animals speak. But they speak differently. Here, the "speak" behaviour is polymorphic in the sense and depends on the animal. So, the abstract "animal" concept does not actually "speak", but specific animals (like dogs and cats) have a concrete implementation of the action "speak".

Encapsulation

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

The way to initialize the value of a class attribute without a constructor:

```
>>> class three:
        val = 7
>>> three.val
```

We can also do this inside class functions:

```
>>> class three:
        def func(self,val):
            self.val=val
>>> t = three()
>>> t.func(8)
>>> t.val
```

Or we can ask the user for input.

```
>>> class three:
        def __init__(self):
            self.val = input("What value?")
>>> t = three()
```

Object Creation

__new__ is a static class method that lets us control object creation.

```
1. class Employee:
2.     id = 10;
3.     name = "Vivek"
4.     def display (self):
5.         print(self.id,self.name)
```

self is used as a reference variable which refers to the current class object. Whenever we make a call to the class constructor, it makes a call to __new__.

```
1. >>> class demo:
        1. def __new__(self):
                1. return 'Vivek'
2. >>> d = demo()
3. >>> type(d)
```

```
1. class Employee:
2.     id = 10;
3.     name = "Vivek"
4.     def display (self):
5.         print("ID: %d \nName: %s"%(self.id, self.name))
6. emp = Employee()
7. emp.display()
```

Python Constructor

A constructor is a special type of method (function) which is used to initialize the instance members of the class.

Constructors can be of two types.

1. Parameterized Constructor
2. Non-parameterized Constructor

Constructor definition is executed when we create the object of this class. Constructors also verify that there are enough resources for the object to perform any start-up task.

Notes:

1. The name of the **constructor** must always be the same as the class name.
2. A **constructor** cannot have a return type not even *void* as the implicit return type of a constructor is the class type where belongs. But a constructor can have *access specifiers*.
3. There is a default constructor added automatically in case we do not provide any custom implementation of the constructors.
4. A **constructor** is not considered as a member of the class. The reason is that it is an *initializer* and thus cannot be *inherited*. It simply belongs to the class for which it is created.

```
1.    def __init__(self,name,id):
2.        self.id = id;
3.        self.name = name;
```

Python Non-Parameterized Constructor Example

The **constructors** that have an empty parameter are known as *non-parameterized constructors*. They are used to initialize the object with *default values* or certain specific constants depending upon the user.

```
1. class Student:
2.     def __init__(self):
3.         print("This is non parametrized constructor")
4.     def show(self, name):
5.         print("Hello”, name)
6. student = Student()
7. student.show("John")
```

Python In-built class functions

1	getattr(obj,name,default)	It is used to access the attribute of the object.
2	setattr(obj, name,value)	It is used to set a particular value to the specific attribute of an object.
3	delattr(obj, name)	It is used to delete a specific attribute.
4	hasattr(obj, name)	It returns true if the object contains some specific attribute.

```
1.  class Student:
2.      def __init__(self,name,id,age):
3.          self.name = name;
4.          self.id = id;
5.          self.age = age
6.
7.  s = Student("Vivek",101,22)
8.
9.  #prints the attribute name of the object s
10. print(getattr(s,'name'))
11.
12. # reset the value of attribute age to 23
13. setattr(s,"age",23)
14.
15. # prints the modified value of age
16. print(getattr(s,'age'))
17.
18. # prints true if the student contains the attribute with name id
19.
20. print(hasattr(s,'id'))
21. # deletes the attribute age
22. delattr(s,'age')
23.
24. # this will give an error since the attribute age has been deleted
25. print(s.age)
```

Built-in class attributes

1	__dict__	It provides the dictionary containing the information about the class namespace.
2	__doc__	It contains a string which has the class documentation
3	__name__	It is used to access the class name.
4	__module__	It is used to access the module in which, this class is defined.
5	__bases__	It contains a tuple including all base classes.

```
1. class Student:
2.     def __init__(self,name,id,age):
3.         self.name = name;
4.         self.id = id;
5.         self.age = age
6.     def display_details(self):
7.         print("Name:%s, ID:%d, age:%d"%(self.name,self.id))
8. s = Student("Vivek",101,22)
9. print(s.__doc__)
10. print(s.__dict__)
11. print(s.__module__)
```