# **Python OOPs Concepts**

Like other general-purpose programming languages, Python is also an object-oriented language since its beginning. It allows us to develop applications using an Object-Oriented approach. In [Python](https://www.javatpoint.com/python-tutorial), we can easily create and use classes and objects.

An object-oriented paradigm is to design the program using classes and objects. The object is related to real-word entities such as book, house, pencil, etc. The oops concept focuses on writing the reusable code. It is a widespread technique to solve the problem by creating objects.

Major principles of object-oriented programming system are given below.

* Class
* Object
* Method
* Inheritance
* Polymorphism
* Data Abstraction
* Encapsulation

## **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.

**Syntax**

1. **class** ClassName:
2. <statement-1>
3. .
4. .
5. <statement-N>

## **Object**

The object is an entity that has state and behavior. 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. All functions have a built-in attribute \_\_doc\_\_, which returns the docstring defined in the function source code.

When we define a class, it needs to create an object to allocate the memory. Consider the following example.

**Example:**

1. **class** car:
2. **def** \_\_init\_\_(self,modelname, year):
3. self.modelname = modelname
4. self.year = year
5. **def** display(self):
6. **print**(self.modelname,self.year)
8. c1 = car("Toyota", 2016)
9. c1.display()

**Output:**

Toyota 2016

In the above example, we have created the class named car, and it has two attributes modelname and year. We have created a c1 object to access the class attribute. The c1 object will allocate memory for these values. We will learn more about class and object in the next tutorial.

## **Method**

The method is a function that is associated with an object. In Python, a method is not unique to class instances. Any object type can have methods.

## **Inheritance**

Inheritance is the most important aspect of object-oriented programming, which simulates the real-world concept of inheritance. It specifies that the child object acquires all the properties and behaviors of the parent object.

By using inheritance, we can create a class which uses all the properties and behavior 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 the re-usability of the code.

## **Polymorphism**

Polymorphism contains two words "poly" and "morphs". Poly means many, and morph means 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" behavior is polymorphic in a 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**

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.

## **Data Abstraction**

Data abstraction and encapsulation both are often used as synonyms. Both are nearly synonyms because data abstraction is achieved through encapsulation.

Abstraction is used to hide internal details and show only functionalities. Abstracting something means to give names to things so that the name captures the core of what a function or a whole program does.

## **Object-oriented vs. Procedure-oriented Programming languages**

The difference between object-oriented and procedure-oriented programming is given below:

|  |  |  |
| --- | --- | --- |
| **Index** | **Object-oriented Programming** | **Procedural Programming** |
| 1. | Object-oriented programming is the problem-solving approach and used where computation is done by using objects. | Procedural programming uses a list of instructions to do computation step by step. |
| 2. | It makes the development and maintenance easier. | In procedural programming, It is not easy to maintain the codes when the project becomes lengthy. |
| 3. | It simulates the real world entity. So real-world problems can be easily solved through oops. | It doesn't simulate the real world. It works on step by step instructions divided into small parts called functions. |
| 4. | It provides data hiding. So it is more secure than procedural languages. You cannot access private data from anywhere. | Procedural language doesn't provide any proper way for data binding, so it is less secure. |
| 5. | Example of object-oriented programming languages is C++, Java, .Net, Python, C#, etc. | Example of procedural languages are: C, Fortran, Pascal, VB etc |

# **Python Class and Objects**

We have already discussed in previous tutorial, a class is a virtual entity and can be seen as a blueprint of an object. The class came into existence when it instantiated. Let's understand it by an example.

Suppose a class is a prototype of a building. A building contains all the details about the floor, rooms, doors, windows, etc. we can make as many buildings as we want, based on these details. Hence, the building can be seen as a class, and we can create as many objects of this class.

On the other hand, the object is the instance of a class. The process of creating an object can be called instantiation.

In this section of the tutorial, we will discuss creating classes and objects in Python. We will also discuss how a class attribute is accessed by using the object.

## **Creating classes in Python**

In Python, a class can be created by using the keyword class, followed by the class name. The syntax to create a class is given below.

**Syntax**

1. **class** ClassName:
2. #statement\_suite

In Python, we must notice that each class is associated with a documentation string which can be accessed by using **<class-name>.\_\_doc\_\_.** A class contains a statement suite including fields, constructor, function, etc. definition.

Consider the following example to create a class **Employee** which contains two fields as Employee id, and name.

The class also contains a function **display()**, which is used to display the information of the **Employee.**

**Example**

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

Here, the **self** is used as a reference variable, which refers to the current class object. It is always the first argument in the function definition. However, using **self** is optional in the function call.

### **The self-parameter**

The self-parameter refers to the current instance of the class and accesses the class variables. We can use anything instead of self, but it must be the first parameter of any function which belongs to the class.

## **Creating an instance of the class**

A class needs to be instantiated if we want to use the class attributes in another class or method. A class can be instantiated by calling the class using the class name.

The syntax to create the instance of the class is given below.

1. <object-name> = <**class**-name>(<arguments>)

The following example creates the instance of the class Employee defined in the above example.

**Example**

1. **class** Employee:
2. id = 10
3. name = "John"
4. **def** display (self):
5. **print**("ID: %d \nName: %s"%(self.id,self.name))
6. # Creating a emp instance of Employee class
7. emp = Employee()
8. emp.display()

**Output:**

ID: 10

Name: John

In the above code, we have created the Employee class which has two attributes named id and name and assigned value to them. We can observe we have passed the self as parameter in display function. It is used to refer to the same class attribute.

We have created a new instance object named **emp.** By using it, we can access the attributes of the class.

## **Delete the Object**

We can delete the properties of the object or object itself by using the del keyword. Consider the following example.

**Example**

1. **class** Employee:
2. id = 10
3. name = "John"
5. **def** display(self):
6. **print**("ID: %d \nName: %s" % (self.id, self.name))
7. # Creating a emp instance of Employee class
9. emp = Employee()
11. # Deleting the property of object
12. **del** emp.id
13. # Deleting the object itself
14. **del** emp
15. emp.display()

It will through the Attribute error because we have deleted the object **emp**

# **Python Constructor**

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

In C++ or Java, the constructor has the same name as its class, but it treats constructor differently in Python. It is used to create an object.

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.

## **Creating the constructor in python**

In Python, the method the **\_\_init\_\_()** simulates the constructor of the class. This method is called when the class is instantiated. It accepts the **self**-keyword as a first argument which allows accessing the attributes or method of the class.

We can pass any number of arguments at the time of creating the class object, depending upon the **\_\_init\_\_()** definition. It is mostly used to initialize the class attributes. Every class must have a constructor, even if it simply relies on the default constructor.

Consider the following example to initialize the **Employee** class attributes.

### **Example**

1. **class** Employee:
2. **def** \_\_init\_\_(self, name, id):
3. self.id = id
4. self.name = name
6. **def** display(self):
7. **print**("ID: %d \nName: %s" % (self.id, self.name))

10. emp1 = Employee("John", 101)
11. emp2 = Employee("David", 102)
13. # accessing display() method to print employee 1 information
15. emp1.display()
17. # accessing display() method to print employee 2 information
18. emp2.display()

**Output:**

ID: 101

Name: John

ID: 102

Name: David

### **Counting the number of objects of a class**

The constructor is called automatically when we create the object of the class. Consider the following example.

### **Example**

1. **class** Student:
2. count = 0
3. **def** \_\_init\_\_(self):
4. Student.count = Student.count + 1
5. s1=Student()
6. s2=Student()
7. s3=Student()
8. **print**("The number of students:",Student.count)

**Output:**

The number of students: 3

## **Python Non-Parameterized Constructor**

The non-parameterized constructor uses when we do not want to manipulate the value or the constructor that has only self as an argument. Consider the following example.

### **Example**

1. **class** Student:
2. # Constructor - non parameterized
3. **def** \_\_init\_\_(self):
4. **print**("This is non parametrized constructor")
5. **def** show(self,name):
6. **print**("Hello",name)
7. student = Student()
8. student.show("John")

## **Python Parameterized Constructor**

The parameterized constructor has multiple parameters along with the **self**. Consider the following example.

### **Example**

1. **class** Student:
2. # Constructor - parameterized
3. **def** \_\_init\_\_(self, name):
4. **print**("This is parametrized constructor")
5. self.name = name
6. **def** show(self):
7. **print**("Hello",self.name)
8. student = Student("John")
9. student.show()

**Output:**

This is parametrized constructor

Hello John

## **Python Default Constructor**

When we do not include the constructor in the class or forget to declare it, then that becomes the default constructor. It does not perform any task but initializes the objects. Consider the following example.

### **Example**

1. **class** Student:
2. roll\_num = 101
3. name = "Joseph"
5. **def** display(self):
6. **print**(self.roll\_num,self.name)
8. st = Student()
9. st.display()

**Output:**

101 Joseph

## **More than One Constructor in Single class**

Let's have a look at another scenario, what happen if we declare the two same constructors in the class.

### **Example**

1. **class** Student:
2. **def** \_\_init\_\_(self):
3. **print**("The First Constructor")
4. **def** \_\_init\_\_(self):
5. **print**("The second contructor")
7. st = Student()

**Output:**

The Second Constructor

In the above code, the object **st** called the second constructor whereas both have the same configuration. The first method is not accessible by the **st** object. Internally, the object of the class will always call the last constructor if the class has multiple constructors.

#### **Note: The constructor overloading is not allowed in Python.**

## **Python built-in class functions**

The built-in functions defined in the class are described in the following table.

|  |  |  |
| --- | --- | --- |
| **SN** | **Function** | **Description** |
| 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. |

### **Example**

1. **class** Student:
2. **def** \_\_init\_\_(self, name, id, age):
3. self.name = name
4. self.id = id
5. self.age = age
7. # creates the object of the class Student
8. s = Student("John", 101, 22)
10. # prints the attribute name of the object s
11. **print**(getattr(s, 'name'))
13. # reset the value of attribute age to 23
14. setattr(s, "age", 23)
16. # prints the modified value of age
17. **print**(getattr(s, 'age'))
19. # prints true if the student contains the attribute with name id
21. **print**(hasattr(s, 'id'))
22. # deletes the attribute age
23. delattr(s, 'age')
25. # this will give an error since the attribute age has been deleted
26. **print**(s.age)

**Output:**

John

23

True

AttributeError: 'Student' object has no attribute 'age'

## **Built-in class attributes**

Along with the other attributes, a Python class also contains some built-in class attributes which provide information about the class.

The built-in class attributes are given in the below table.

|  |  |  |
| --- | --- | --- |
| **SN** | **Attribute** | **Description** |
| 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. |

### **Example**

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("John",101,22)
9. **print**(s.\_\_doc\_\_)
10. **print**(s.\_\_dict\_\_)
11. **print**(s.\_\_module\_\_)

Output:

None

{‘name’: ‘John’, ‘id’: 101, ‘age’: 22}

\_\_main\_\_

# **Python Inheritance**

Inheritance is an important aspect of the object-oriented paradigm. Inheritance provides code reusability to the program because we can use an existing class to create a new class instead of creating it from scratch.

In inheritance, the child class acquires the properties and can access all the data members and functions defined in the parent class. A child class can also provide its specific implementation to the functions of the parent class. In this section of the tutorial, we will discuss inheritance in detail.

In python, a derived class can inherit base class by just mentioning the base in the bracket after the derived class name. Consider the following syntax to inherit a base class into the derived class.

Python Inheritance

### **Syntax**

1. **class** derived-**class**(base **class**):
2. <**class**-suite>

A class can inherit multiple classes by mentioning all of them inside the bracket. Consider the following syntax.

### **Syntax**

1. **class** derive-**class**(<base **class** 1>, <base **class** 2>, ..... <base **class** n>):
2. <**class** - suite>

### **Example 1**

1. **class** Animal:
2. **def** speak(self):
3. **print**("Animal Speaking")
4. #child class Dog inherits the base class Animal
5. **class** Dog(Animal):
6. **def** bark(self):
7. **print**("dog barking")
8. d = Dog()
9. d.bark()
10. d.speak()

**Output:**

dog barking

Animal Speaking

## **Python Multi-Level inheritance**

Multi-Level inheritance is possible in python like other object-oriented languages. Multi-level inheritance is archived when a derived class inherits another derived class. There is no limit on the number of levels up to which, the multi-level inheritance is archived in python.

Python Inheritance

The syntax of multi-level inheritance is given below.

### **Syntax**

1. **class** class1:
2. <**class**-suite>
3. **class** class2(class1):
4. <**class** suite>
5. **class** class3(class2):
6. <**class** suite>
7. .
8. .

### **Example**

1. **class** Animal:
2. **def** speak(self):
3. **print**("Animal Speaking")
4. #The child class Dog inherits the base class Animal
5. **class** Dog(Animal):
6. **def** bark(self):
7. **print**("dog barking")
8. #The child class Dogchild inherits another child class Dog
9. **class** DogChild(Dog):
10. **def** eat(self):
11. **print**("Eating bread...")
12. d = DogChild()
13. d.bark()
14. d.speak()
15. d.eat()

**Output:**

dog barking

Animal Speaking

Eating bread...

## **Python Multiple inheritance**

Python provides us the flexibility to inherit multiple base classes in the child class.

Python Inheritance

The syntax to perform multiple inheritance is given below.

### **Syntax**

1. **class** Base1:
2. <**class**-suite>
4. **class** Base2:
5. <**class**-suite>
6. .
7. .
8. .
9. **class** BaseN:
10. <**class**-suite>
12. **class** Derived(Base1, Base2, ...... BaseN):
13. <**class**-suite>

### **Example**

1. **class** Calculation1:
2. **def** Summation(self,a,b):
3. **return** a+b;
4. **class** Calculation2:
5. **def** Multiplication(self,a,b):
6. **return** a\*b;
7. **class** Derived(Calculation1,Calculation2):
8. **def** Divide(self,a,b):
9. **return** a/b;
10. d = Derived()
11. **print**(d.Summation(10,20))
12. **print**(d.Multiplication(10,20))
13. **print**(d.Divide(10,20))

**Output:**

30

200

0.5

## **The issubclass(sub,sup) method**

The issubclass(sub, sup) method is used to check the relationships between the specified classes. It returns true if the first class is the subclass of the second class, and false otherwise.

Consider the following example.

### **Example**

1. **class** Calculation1:
2. **def** Summation(self,a,b):
3. **return** a+b;
4. **class** Calculation2:
5. **def** Multiplication(self,a,b):
6. **return** a\*b;
7. **class** Derived(Calculation1,Calculation2):
8. **def** Divide(self,a,b):
9. **return** a/b;
10. d = Derived()
11. **print**(issubclass(Derived,Calculation2))
12. **print**(issubclass(Calculation1,Calculation2))

**Output:**

True

False

## **The isinstance (obj, class) method**

The isinstance() method is used to check the relationship between the objects and classes. It returns true if the first parameter, i.e., obj is the instance of the second parameter, i.e., class.

Consider the following example.

### **Example**

1. **class** Calculation1:
2. **def** Summation(self,a,b):
3. **return** a+b;
4. **class** Calculation2:
5. **def** Multiplication(self,a,b):
6. **return** a\*b;
7. **class** Derived(Calculation1,Calculation2):
8. **def** Divide(self,a,b):
9. **return** a/b;
10. d = Derived()
11. **print**(isinstance(d,Derived))

**Output:**

True

## **Method Overriding**

We can provide some specific implementation of the parent class method in our child class. When the parent class method is defined in the child class with some specific implementation, then the concept is called method overriding. We may need to perform method overriding in the scenario where the different definition of a parent class method is needed in the child class.

Consider the following example to perform method overriding in python.

### **Example**

1. **class** Animal:
2. **def** speak(self):
3. **print**("speaking")
4. **class** Dog(Animal):
5. **def** speak(self):
6. **print**("Barking")
7. d = Dog()
8. d.speak()

**Output:**

Barking

### **Real Life Example of method overriding**

1. **class** Bank:
2. **def** getroi(self):
3. **return** 10;
4. **class** SBI(Bank):
5. **def** getroi(self):
6. **return** 7;
8. **class** ICICI(Bank):
9. **def** getroi(self):
10. **return** 8;
11. b1 = Bank()
12. b2 = SBI()
13. b3 = ICICI()
14. **print**("Bank Rate of interest:",b1.getroi());
15. **print**("SBI Rate of interest:",b2.getroi());
16. **print**("ICICI Rate of interest:",b3.getroi());

**Output:**

Bank Rate of interest: 10

SBI Rate of interest: 7

ICICI Rate of interest: 8

## **Data abstraction in python**

Abstraction is an important aspect of object-oriented programming. In python, we can also perform data hiding by adding the double underscore (\_\_\_) as a prefix to the attribute which is to be hidden. After this, the attribute will not be visible outside of the class through the object.

Consider the following example.

### **Example**

1. **class** Employee:
2. \_\_count = 0;
3. **def** \_\_init\_\_(self):
4. Employee.\_\_count = Employee.\_\_count+1
5. **def** display(self):
6. **print**("The number of employees",Employee.\_\_count)
7. emp = Employee()
8. emp2 = Employee()
9. **try**:
10. **print**(emp.\_\_count)
11. **finally**:
12. emp.display()

**Output:**

The number of employees 2

AttributeError: 'Employee' object has no attribute '\_\_count

# **Abstraction in Python**

Abstraction is used to hide the internal functionality of the function from the users. The users only interact with the basic implementation of the function, but inner working is hidden. User is familiar with that **"what function does"** but they don't know **"how it does."**

In simple words, we all use the smartphone and very much familiar with its functions such as camera, voice-recorder, call-dialing, etc., but we don't know how these operations are happening in the background. Let's take another example - When we use the TV remote to increase the volume. We don't know how pressing a key increases the volume of the TV. We only know to press the "+" button to increase the volume.

That is exactly the abstraction that works in the [object-oriented concept](https://www.javatpoint.com/python-oops-concepts).

## **Why Abstraction is Important?**

In Python, an abstraction is used to hide the irrelevant data/class in order to reduce the complexity. It also enhances the application efficiency. Next, we will learn how we can achieve abstraction using the [Python program](https://www.javatpoint.com/python-programs).

## **Abstraction classes in Python**

In [Python](https://www.javatpoint.com/python-tutorial), abstraction can be achieved by using abstract classes and interfaces.

A class that consists of one or more abstract method is called the abstract class. Abstract methods do not contain their implementation. Abstract class can be inherited by the subclass and abstract method gets its definition in the subclass. Abstraction classes are meant to be the blueprint of the other class. An abstract class can be useful when we are designing large functions. An abstract class is also helpful to provide the standard interface for different implementations of components. Python provides the **abc** module to use the abstraction in the Python program. Let's see the following syntax.

**Syntax**

1. from abc **import** ABC
2. **class** ClassName(ABC):

We import the ABC class from the **abc** module.

## **Abstract Base Classes**

An abstract base class is the common application program of the interface for a set of subclasses. It can be used by the third-party, which will provide the implementations such as with plugins. It is also beneficial when we work with the large code-base hard to remember all the classes.

## **Working of the Abstract Classes**

Unlike the other high-level language, Python doesn't provide the abstract class itself. We need to import the abc module, which provides the base for defining Abstract Base classes (ABC). The ABC works by decorating methods of the base class as abstract. It registers concrete classes as the implementation of the abstract base. We use the **@abstractmethod** decorator to define an abstract method or if we don't provide the definition to the method, it automatically becomes the abstract method. Let's understand the following example.

**Example -**

1. # Python program demonstrate
2. # **abstract** base **class** work
3. from abc **import** ABC, abstractmethod
4. **class** Car(ABC):
5. def mileage(self):
6. pass
8. **class** Tesla(Car):
9. def mileage(self):
10. print("The mileage is 30kmph")
11. **class** Suzuki(Car):
12. def mileage(self):
13. print("The mileage is 25kmph ")
14. **class** Duster(Car):
15. def mileage(self):
16. print("The mileage is 24kmph ")
18. **class** Renault(Car):
19. def mileage(self):
20. print("The mileage is 27kmph ")
22. # Driver code
23. t= Tesla ()
24. t.mileage()
26. r = Renault()
27. r.mileage()
29. s = Suzuki()
30. s.mileage()
31. d = Duster()
32. d.mileage()

**Output:**

The mileage is 30kmph

The mileage is 27kmph

The mileage is 25kmph

The mileage is 24kmph

**Explanation -**

In the above code, we have imported the **abc module** to create the abstract base class. We created the Car class that inherited the ABC class and defined an abstract method named mileage(). We have then inherited the base class from the three different subclasses and implemented the abstract method differently. We created the objects to call the abstract method.

Let's understand another example.

Let's understand another example.

**Example -**

1. # Python program to define
2. # **abstract** **class**
4. from abc **import** ABC
6. **class** Polygon(ABC):
8. # **abstract** method
9. def sides(self):
10. pass
12. **class** Triangle(Polygon):

15. def sides(self):
16. print("Triangle has 3 sides")
18. **class** Pentagon(Polygon):

21. def sides(self):
22. print("Pentagon has 5 sides")
24. **class** Hexagon(Polygon):
26. def sides(self):
27. print("Hexagon has 6 sides")
29. **class** square(Polygon):
31. def sides(self):
32. print("I have 4 sides")
34. # Driver code
35. t = Triangle()
36. t.sides()
38. s = square()
39. s.sides()
41. p = Pentagon()
42. p.sides()
44. k = Hexagon()
45. K.sides()

**Output:**

Triangle has 3 sides

Square has 4 sides

Pentagon has 5 sides

Hexagon has 6 sides

**Explanation -**

In the above code, we have defined the abstract base class named Polygon and we also defined the abstract method. This base class inherited by the various subclasses. We implemented the abstract method in each subclass. We created the object of the subclasses and invoke the **sides()** method. The hidden implementations for the **sides()** method inside the each subclass comes into play. The abstract method **sides()** method, defined in the abstract class, is never invoked.

## **Points to Remember**

Below are the points which we should remember about the abstract base class in Python.

* An Abstract class can contain the both method normal and abstract method.
* An Abstract cannot be instantiated; we cannot create objects for the abstract class.

Abstraction is essential to hide the core functionality from the users. We have covered the all the basic concepts of Abstraction in Python.