Python OOPs Concepts

Python is an object-oriented programming language. You can easily create and use classes and objects in Python.

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

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

Object

Object is an entity that has state and behavior. It may be anything. It may be physical and logical. For example: 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 doc string defined in the function source code.

Class

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:**

**class** ClassName:

    <statement-1>

    .

    .

    .

    <statement-N>

Method

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

Inheritance

Inheritance is a feature of object-oriented programming. It specifies that one object acquires all the properties and behaviors of parent object. By using inheritance you can define a new class with a little or no changes to the existing class. The new class is known as derived class or child class and from which it inherits the properties is called base class or parent class.

It provides re-usability of the code.

Polymorphism

Polymorphism is made by two words "poly" and "morphs". Poly means many and Morphs means form, shape. It defines that one task can be performed in different ways. For example: You have a class animal and all animals talk. But they talk differently. Here, the "talk" behavior is polymorphic in the sense and totally depends on the animal. So, the abstract "animal" concept does not actually "talk", but specific animals (like dogs and cats) have a concrete implementation of the action "talk".

Encapsulation

Encapsulation is also the feature 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.

Capsule-🡪other layer-🡪portecting the medicine

Data Abstraction

Data abstraction and encapsulation both are often used as synonyms. Both are nearly synonym 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

|  |  |  |
| --- | --- | --- |
| **Index** | **Object-oriented Programming** | **Procedural Programming** |
| 1. | Object-oriented programming is an approach to problem solving where computation is done by using objects. | Procedural programming uses a list of instructions to do computation step by step. |
| 2. | It makes development and maintenance easier. | In procedural programming, It is not easy to maintain the codes when 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 in 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 are: C++, Java, .Net, Python, C# etc. | Example of procedural languages are: C, Fortran, Pascal, VB etc. |

Python Class

A class is a blueprint for the object. 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, doors, windows, etc. we can make another buildings (as many as we want) based on these details. So building is a class and we can create many objects from a class.

An object is also called an instance of a class and the process of creating this object is known as instantiation.

Python classes contain all the standard features of Object Oriented Programming. A python class is a mixture of class mechanism of C++ and Modula-3.

Define a class in Python

In Python, a class is defined by using a keyword class like a function definition begins with the keyword def.

**Syntax of a class definition:**

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

A class creates a new local namespace to define its all attribute. These attributes may be data or functions.

Python Object Class Example

**class** Student:

**def** \_\_init\_\_(self, rollno, name):

      self.rollno = rollno

      self.name = name

**def** displayStudent(self):

**print** "rollno : ", self.rollno,  ", name: ", self.name

emp1 = Student(121, "Ajeet")

emp2 = Student(122, "Sonoo")

emp1.displayStudent()

emp2.displayStudent()

**Output:**

1. rollno :  121 , name:  Ajeet
2. rollno :  122 , name:  Sonoo

Python Constructors

A constructor is a special type of method (function) that is called when it instantiates an object using the definition found in your class. The constructors are normally used to initialize (assign values) to the instance variables. Constructors also verify that there are enough resources for the object to perform any start-up task.

**Creating a constructor:**

A constructor is a class function that begins with double underscore (\_). The name of the constructor is always the same \_\_init\_\_().

While creating an object, a constructor can accept arguments if necessary. When you create a class without a constructor, Python automatically creates a default constructor that doesn't do anything.

Every class must have a constructor, even if it simply relies on the default constructor.

**Let's take an example:**

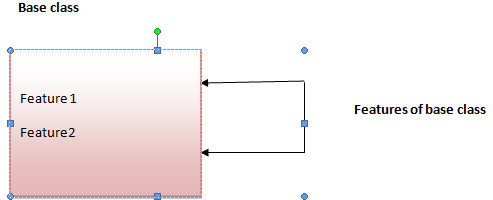
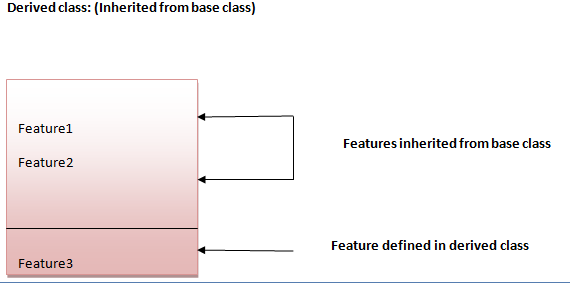
Let's create a class named ComplexNumber, having two functions \_\_init\_\_() function to initialize the variable and getData() to display the number properly.

What is Inheritance

Inheritance is used to specify that one class will get most or all of its features from its parent class. It is a feature of Object Oriented Programming. It is a very powerful feature which facilitates users to create a new class with a few or more modification to an existing class. The new class is called child class or derived class and the main class from which it inherits the properties is called base class or parent class.

The child class or derived class inherits the features from the parent class, adding new features to it. It facilitates re-usability of code.

**Image representation:**

**Syntax 1:**

**class** DerivedClassName(BaseClassName):

    <statement-1>

    .

    .

    .

    <statement-N>

**Syntax 2:**

**class** DerivedClassName(modulename.BaseClassName):

    <statement-1>

    .

    .

    .

    <statement-N>

Parameter explanation:

The name BaseClassName must be defined in a scope containing the derived class definition. You can also use other arbitrary expressions in place of a base class name. This is used when the base class is defined in another module.

Python Inheritance Example

Let's see a simple python inheritance example where we are using two classes: Animal and Dog. Animal is the parent or base class and Dog is the child class.

Here, we are defining eat() method in Animal class and bark() method in Dog class. In this example, we are creating instance of Dog class and calling eat() and bark() methods by the instance of child class only. Since, parent properties and behaviors are inherited to child object automatically, we can call parent and child class methods by the child instance only.

**class** Animal:

**def** eat(self):

**print** 'Eating...'

**class** Dog(Animal):

**def** bark(self):

**print** 'Barking...'

d=Dog()

d.eat()

d.bark()

**Output:**

1. Eating...
2. Barking...

Multilevel Inheritance in Python

Multilevel inheritance is also possible in Python unlike other programming languages. You can inherit a derived class from another derived class. This is known as multilevel inheritance. In Python, multilevel inheritance can be done at any depth.

Python Multilevel Inheritance Example

**class** Animal:

**def** eat(self):

**print** 'Eating...'

**class** Dog(Animal):

**def** bark(self):

**print** 'Barking...'

**class** BabyDog(Dog):

**def** weep(self):

**print** 'Weeping...'

d=BabyDog()

d.eat()

d.bark()

d.weep()

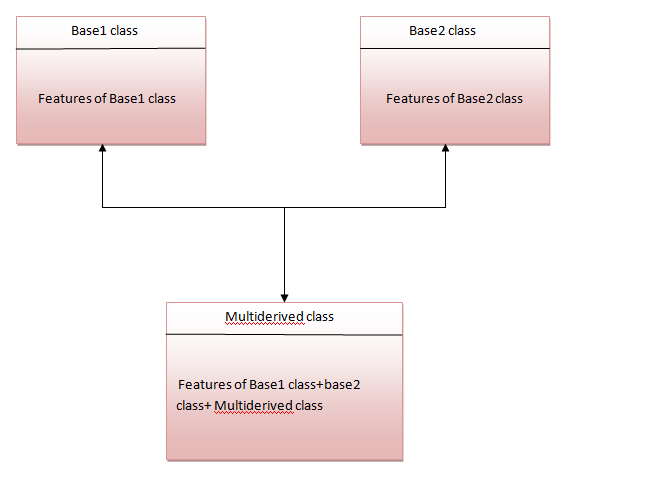
**Output:**

1. Eating...
2. Barking...
3. Weeping

Multiple Inheritance in Python

Python supports multiple inheritance also. You can derive a child class from more than one base (parent) class.

**Image representation:**



The multiderived class inherits the properties of both class base1 and base2.

Let's see the syntax of multiple inheritance in Python.

**Syntax:**

**class** DerivedClassName(Base1, Base2, Base3):

    <statement-1>

    .

    .

    .

    <statement-N>

**Or**

1. **class** Base1:
2. **pass**
4. **class** Base2:
5. **pass**
7. **class** MultiDerived(Base1, Base2):
8. **pass**

**Example:**

**class**  First(object):

**def** \_\_init\_\_(self):

    super(First, self).\_\_init\_\_()

**print**("first")

**class** Second(object):

**def** \_\_init\_\_(self):

    super(Second, self).\_\_init\_\_()

**print**("second")

**class** Third(Second, First):

**def** \_\_init\_\_(self):

    super(Third, self).\_\_init\_\_()

**print**("third")

Third();

**Output:**

1. first
2. second
3. third

## Inheritance

Inheritance is a way of creating a new class for using details of an existing class without modifying it. The newly formed class is a derived class (or child class). Similarly, the existing class is a base class (or parent class).

### Use of Inheritance in Python

# parent class

class Bird:

def \_\_init\_\_(self):

print("Bird is ready")

def whoisThis(self):

print("Bird")

def swim(self):

print("Swim faster")

# child class

class Penguin(Bird):

def \_\_init\_\_(self):

# call super() function

super().\_\_init\_\_()

print("Penguin is ready")

def whoisThis(self):

print("Penguin")

def run(self):

print("Run faster")

peggy = Penguin()

peggy.whoisThis()

peggy.swim()

peggy.run()

**Output**

Bird is ready

Penguin is ready

Penguin

Swim faster

Run faster

In the above program, we created two classes i.e. Bird (parent class) and Penguin (child class). The child class inherits the functions of parent class. We can see this from the swim() method.

Again, the child class modified the behavior of the parent class. We can see this from the whoisThis() method. Furthermore, we extend the functions of the parent class, by creating a new run() method.

Additionally, we use the super() function inside the \_\_init\_\_() method. This allows us to run the \_\_init\_\_() method of the parent class inside the child class.

## Encapsulation

Using OOP in Python, we can restrict access to methods and variables. This prevents data from direct modification which is called encapsulation. In Python, we denote private attributes using underscore as the prefix i.e single \_ or double \_\_.

### Data Encapsulation in Python

class Computer:

def \_\_init\_\_(self):

self.\_\_maxprice = 900

def sell(self):

print("Selling Price: {}".format(self.\_\_maxprice))

def setMaxPrice(self, price):

self.\_\_maxprice = price

c = Computer()

c.sell()

# change the price

c.\_\_maxprice = 1000

c.sell()

# using setter function

c.setMaxPrice(1000)

c.sell()

**Output**

Selling Price: 900

Selling Price: 900

Selling Price: 1000

In the above program, we defined a Computer class.

We used \_\_init\_\_() method to store the maximum selling price of Computer. We tried to modify the price. However, we can't change it because Python treats the \_\_maxprice as private attributes.

As shown, to change the value, we have to use a setter function i.e setMaxPrice() which takes price as a parameter.

## Polymorphism

Polymorphism is an ability (in OOP) to use a common interface for multiple forms (data types).

Suppose, we need to color a `shape, there are multiple shape options (rectangle, square, circle). However we could use the same method to color any shape. This concept is called Polymorphism.

### Using Polymorphism in Python

class Parrot:

def fly(self):

print("Parrot can fly")

def swim(self):

print("Parrot can't swim")

class Penguin:

def fly(self):

print("Penguin can't fly")

def swim(self):

print("Penguin can swim")

# common interface

def flying\_test(bird):

bird.fly()

#instantiate objects

blu = Parrot()

peggy = Penguin()

# passing the object

flying\_test(blu)

flying\_test(peggy)

**Output**

Parrot can fly

Penguin can't fly

In the above program, we defined two classes Parrot and Penguin. Each of them have a common fly() method. However, their functions are different.

To use polymorphism, we created a common interface i.e flying\_test() function that takes any object and calls the object's fly() method. Thus, when we passed the blu and peggy objects in the flying\_test() function, it ran effectively.

## Key Points to Remember:

* Object-Oriented Programming makes the program easy to understand as well as efficient.
* Since the class is sharable, the code can be reused.
* Data is safe and secure with data abstraction.
* Polymorphism allows the same interface for different objects, so programmers can write efficient code.

## **Polymorphism-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**

**class** Bank:

**def** getroi(self):

**return** 10;

**class** SBI(Bank):

**def** getroi(self):

**return** 7;

**class** ICICI(Bank):

**def** getroi(self):

**return** 8;

b1 = Bank()

b2 = SBI()

b3 = ICICI()

**print**("Bank Rate of interest:",b1.getroi());

**print**("SBI Rate of interest:",b2.getroi());

**print**("ICICI Rate of interest:",b3.getroi());

**Output:**

Bank Rate of interest: 10

SBI Rate of interest: 7

ICICI Rate of interest: 8

# **Abstract Classes in Python**

Last Updated: 01-04-2020

An abstract class can be considered as a blueprint for other classes. It allows you to create a set of methods that must be created within any child classes built from the abstract class. A class which contains one or more abstract methods is called an abstract class. An abstract method is a method that has a declaration but does not have an implementation. While we are designing large functional units we use an abstract class. When we want to provide a common interface for different implementations of a component, we use an abstract class.  
   
**Why use Abstract Base Classes :**  
By defining an abstract base class, you can define a common Application Program Interface(API) for a set of subclasses. This capability is especially useful in situations where a third-party is going to provide implementations, such as with plugins, but can also help you when working in a large team or with a large code-base where keeping all classes in your mind is difficult or not possible.  
   
**How Abstract Base classes work :**  
By default, Python does not provide abstract classes. Python comes with a module which provides the base for defining Abstract Base classes(ABC) and that module name is ABC. **ABC** works by decorating methods of the base class as abstract and then registering concrete classes as implementations of the abstract base. A method becomes abstract when decorated with the keyword @abstractmethod. For Example –

**Code 1:**

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| # Python program showing  # abstract base class work    from abc import ABC, abstractmethod    class Polygon(ABC):        # abstract method      def noofsides(self):          pass    class Triangle(Polygon):        # overriding abstract method      def noofsides(self):          print("I have 3 sides")    class Pentagon(Polygon):        # overriding abstract method      def noofsides(self):          print("I have 5 sides")    class Hexagon(Polygon):        # overriding abstract method      def noofsides(self):          print("I have 6 sides")    class Quadrilateral(Polygon):        # overriding abstract method      def noofsides(self):          print("I have 4 sides")    # Driver code  R = Triangle()  R.noofsides()    K = Quadrilateral()  K.noofsides()    R = Pentagon()  R.noofsides()    K = Hexagon()  K.noofsides() |

**Output:**

I have 3 sides

I have 4 sides

I have 5 sides

I have 6 sides

**Code 2:**

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| # Python program showing  # abstract base class work    from abc import ABC, abstractmethod  class Animal(ABC):        def move(self):          pass    class Human(Animal):        def move(self):          print("I can walk and run")    class Snake(Animal):        def move(self):          print("I can crawl")    class Dog(Animal):        def move(self):          print("I can bark")    class Lion(Animal):        def move(self):          print("I can roar")    # Driver code  R = Human()  R.move()    K = Snake()  K.move()    R = Dog()  R.move()    K = Lion()  K.move() |

**Output:**

I can walk and run

I can crawl

I can bark

I can roar