OOPS FRAMEWORK :

Oops concepts: Object, Class, Method, Inheritance, Polymorphism,Data abstraction, Encapsulation, Python Frameworks: Explore django framework with an example

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, 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.

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:

```
class car:
   def__init__(self,modelname, year):
      self.modelname = modelname
      self.year = year
   def display(self):
         print(self.modelname,self.year)

   c1 = car("Toyota", 2016)
   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.

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>
```

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.

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

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

Example

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

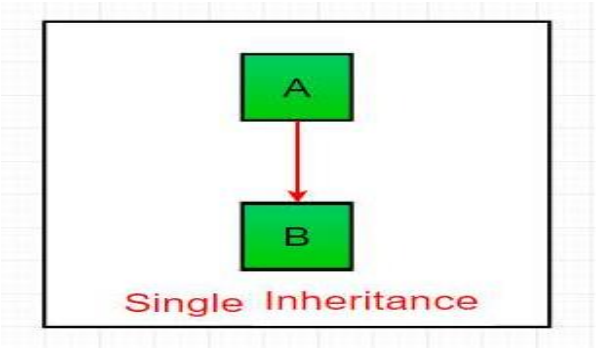
Python Inheritance

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.

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



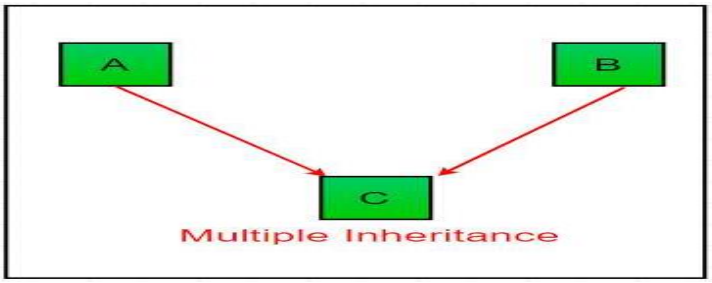
Example:

```
# 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.
```

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



Example:

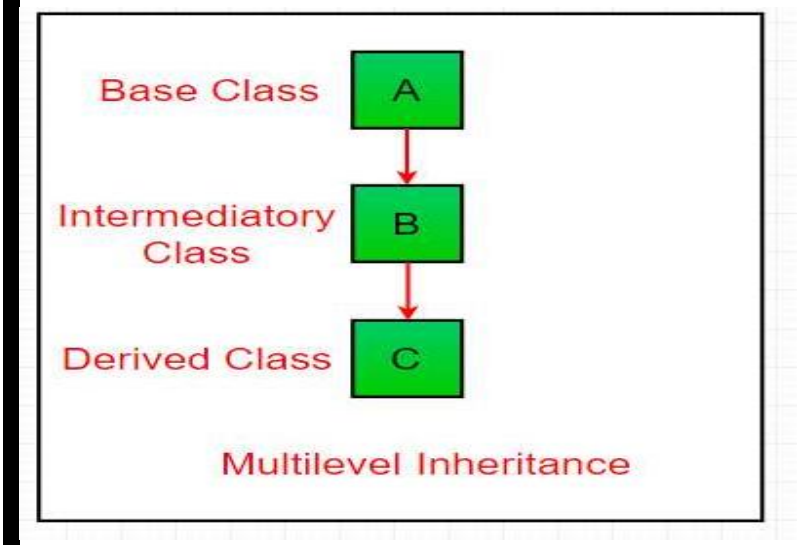
```
# Python program to demonstrate
# multiple inheritance


# 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
```

3. 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 grandfather.



Example:

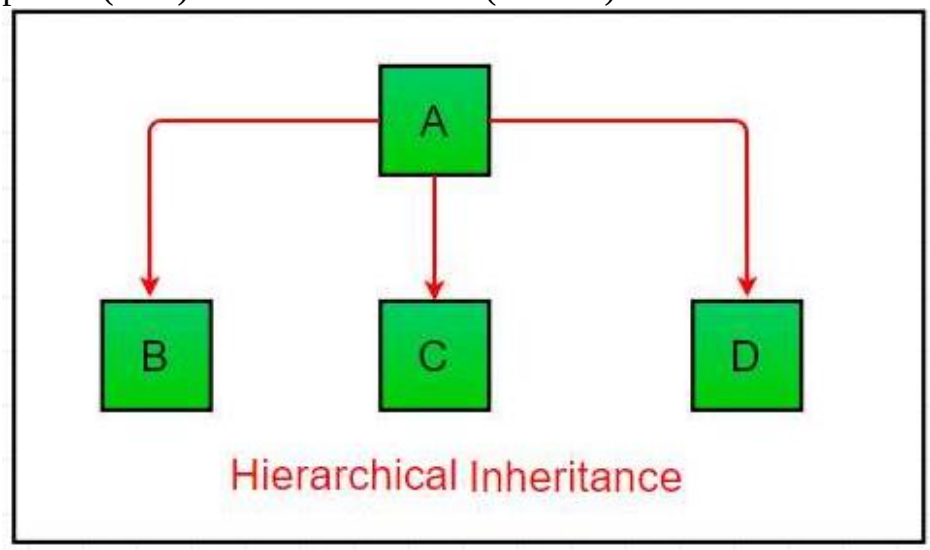
```
# Python program to demonstrate
# multilevel inheritance


# Base class
class Grandfather:
    grandfathername =""
    def grandfather(self):
        print(self.grandfathername)

# Intermediate class
class Father(Grandfather):
    fathername = ""
    def father(self):
        print(self.fathername)

# Derived class
class Son(Father):
    def parent(self):
        print("GrandFather :", self.grandfathername)
        print("Father :", self.fathername)
```

```
# Driver's code
s1 = Son()
s1.grandfathername = "Srinivas"
s1.fathername = "Ankush"
s1.parent()
```

Output:

```
GrandFather : Srinivas
Father : Ankush
```

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



Example:

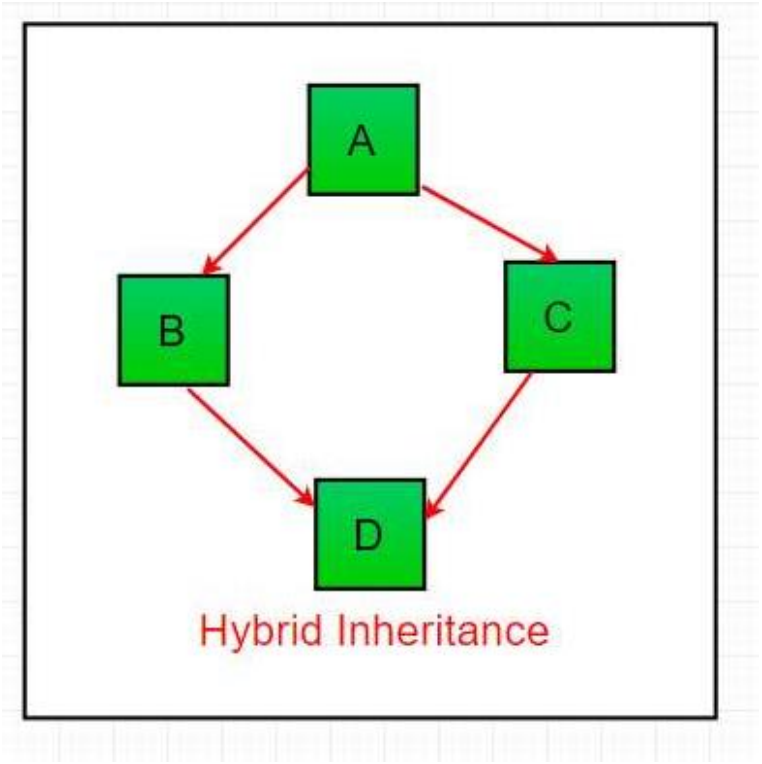
```
# Python program to demonstrate
# Hierarchical inheritance


# 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.
```

2. **Hybrid Inheritance:** Inheritence consisting of multiple types of inheritence is called hybrid inheritence.



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 in Python

The word polymorphism means having many forms. In programming, polymorphism means same function name (but different signatures) being uses for different types.

Example of inbuilt polymorphic functions :

```
# Python program to demonstrate in-built poly-
# morphic functions

# len() being used for a string
print(len("geeks"))

# len() being used for a list
print(len([10, 20, 30]))
```

Output:

```
5

3
```

Examples of used defined polymorphic functions :

```
# A simple Python function to demonstrate#
Polymorphism

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:

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.
```

Encapsulation in Python

Encapsulation is one of the fundamental concepts in object-oriented programming (OOP). It describes the idea of wrapping data and the methods that work on data within one unit. This puts restrictions on accessing variables and methods directly and can prevent the accidental modification of data. To prevent accidental change, an obj variable can only be changed by an of variables are known as **private variable**.

ect's

A class is an example of encapsulation as it encapsulates all the data that is member functions, variables, etc.

Note: The__init__method is a constructor and runs as soon as an object of a class is instantiated.

```
# Python program to
# demonstrate protected members


# Creating a base class
class Base:
    def__init__(self):

        # Protected member
        self._a = 2

# Creating a derived class
class Derived(Base):
    def__init__(self):

        # Calling constructor of
        # Base class
        Base.__init__(self)
        print("Calling protected member of base class: ")
        print(self._a)

obj1 = Derived()

obj2 = Base()

# Calling protected member
# Outside class will result in
# AttributeError
```

```
print(obj2.a)
```

Output:

```
Calling protected member of base class:
2
Traceback (most recent call last):
  File "/home/6fb1b95dfba0e198298f9dd02469eb4a.py", line 25, in
    print(obj1.a)
AttributeError: 'Base' object has no attribute 'a'
```

Abstract Classes in Python

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.

```
# 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
```