

Let's go through some basic concepts of OOPS in python now that you have knowledge of classes and objects.

Encapsulation –

Encapsulation is also an important 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.

In simple terms we can say that encapsulation is implemented through classes. In fact the data members of a class can be accessed through its member functions only. It keeps the data safe from any external interference and misuse. The only way to access the data is through the functions of the class.

Data Hiding –

Data hiding can be defined as the mechanism of hiding the data of a class from the outside world or to be precise, from other classes. This is done to protect the data from any accidental or intentional access. In a class, data may be made private or public. Private data or function of a class cannot be accessed from outside the class while public data or functions can be accessed from anywhere. So data hiding is achieved by making the members of the class private. Access to private members is restricted and is only available to the member functions of the same class. However the public part of the object is accessible outside the class.

Data Abstraction –

The process of identifying and separating the essential features without including the internal details is abstraction. Only the essential information is provided to the outside world while the background details are hidden. Classes use the concept of abstraction. A class encapsulates the relevant data and functions that operate on data by hiding the complex implementation details from the user. The user needs to focus on what a class does rather than how it does.

Abstraction and Encapsulation are complementary concepts. Through encapsulation only we are able to enclose the components of the object into a single unit and separate the private and public members. It is through abstraction that only the essential behaviours of the objects are made visible to the outside world. So we can say that encapsulation is the way to implement data abstraction.





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 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. So, the derived class contains features of the base class as well as of itself. So we see that the base class can be reused again and again to define new classes. Another advantage of inheritance is its transitive nature. It provides re-usability of the code.

Let's dive into different types of **inheritance**:

1. **Single Inheritance**: When a child class inherits from only one parent class, it is called as single inheritance. We saw an example above.
2. **Multiple Inheritance**: When a child class inherits from multiple parent classes, it is called as multiple inheritance.

```
# Python example to show working of multiple
# inheritance
class Base1(object):
    def __init__(self):
        self.str1 = "Geek1"
        print "Base1"

class Base2(object):
    def __init__(self):
        self.str2 = "Geek2"
        print "Base2"

class Derived(Base1, Base2):
    def __init__(self):

        # Calling constructors of Base1
        # and Base2 classes
        Base1.__init__(self)
        Base2.__init__(self)
        print "Derived"

    def printStrs(self):
        print(self.str1, self.str2)

ob = Derived()
ob.printStrs()
```

Output:

```
Base1
Base2
Derived
('Geek1', 'Geek2')
```

3. **Multilevel inheritance:** When we have child and grand child relationship.

```
# A Python program to demonstrate inheritance

# Base or Super class. Note object in bracket.
# (Generally, object is made ancestor of all classes)
# In Python 3.x "class Person" is
# equivalent to "class Person(object)"
class Base(object):

    # Constructor
    def __init__(self, name):
        self.name = name

    # To get name
    def getName(self):
        return self.name

# Inherited or Sub class (Note Person in bracket)
class Child(Base):

    # Constructor
    def __init__(self, name, age):
        Base.__init__(self, name)
        self.age = age

    # To get name
    def getAge(self):
        return self.age

# Inherited or Sub class (Note Person in bracket)
class GrandChild(Child):

    # Constructor
    def __init__(self, name, age, address):
        Child.__init__(self, name, age)
        self.address = address

    # To get address
    def getAddress(self):
        return self.address

# Driver code
g = GrandChild("Geek1", 23, "Noida")
print(g.getName(), g.getAge(), g.getAddress())
```

4. **Hierarchical inheritance:** More than one derived classes are created from a single base.
5. **Hybrid inheritance:** This form combines more than one form of inheritance. Basically, it is a blend of more than one type of inheritance.

Polymorphism –

Polymorphism contains two words "poly" and "morphs". Poly means many and Morphs means form, shape. By polymorphism, we understand that one task can be performed in different ways. So, polymorphism is the ability to use an operator or function in various forms. That is a single function or an operator behaves differently depending upon the data provided to them. So, polymorphism is the ability to use an operator or function in various forms. That is a single function or an operator behaves differently depending upon the data provided to them.

Polymorphism can be achieved in 2 ways-

1. Operator Overloading:

You must have noticed that the '+' operator behaves differently with different data types. With integers it adds the two numbers and with strings it concatenates or joins two strings. For example:

- a. Print 8+9 will give 17 and,
- b. Print "Python" + "programming" will give the output as Pythonprogramming.

This feature where an operator can be used in different forms is known as Operator Overloading and is one of the methods to implement polymorphism.

2. Function Overloading:

Polymorphism in case of functions is a bit different. A named function can also vary depending on the parameters it is given. For example, we define multiple functions with same name but different argument list as shown below:

```
def test():                #function 1
    print "hello"
def test(a, b):            #function 2
    return a+b
def test(a, b, c):         #function 3
    return a+b+c
```

In the example above, three functions by the same name have been defined but with different number of arguments. Now if we give a function call with no argument, say test(), *function 1* will be called. The statement test(10,20) will lead to the execution of *function 2* and if the statement test(10,20,30) is given *function 3* will be called. In either case, all the functions would be known in the program by the same name. This is another way to implement polymorphism and is known as **Function Overloading**.

NOTE: Python doesn't support function overloading. In case you declare multiple functions with same name, the first function will be

overwritten by the second one and then the second one by the third one. Python always considers the latest function definition and hence will always use *function 3* in the above example. Function overloading is supported by languages such as Java,C,C++.

Static and Dynamic Binding –

Binding is the process of linking the function call to the function definition. The body of the function is executed when the function call is made. Binding can be of two types:

1. **Static Binding:** In this type of binding, the linking of function call to the function definition is done during compilation of the program.
2. **Dynamic Binding:** In this type of binding, linking of a function call to the function definition is done at run time. That means the code of the function that is to be linked with function call is unknown until it is executed. Dynamic binding of functions makes the programs more flexible.

Now it's time for some questions!

TRY IT YOURSELF

9-4. Number Served: Start with your program from Exercise 9-1 (page 166). Add an attribute called `number_served` with a default value of 0. Create an instance called `restaurant` from this class. Print the number of customers the restaurant has served, and then change this value and print it again.

Add a method called `set_number_served()` that lets you set the number of customers that have been served. Call this method with a new number and print the value again.

Add a method called `increment_number_served()` that lets you increment the number of customers who've been served. Call this method with any number you like that could represent how many customers were served in, say, a day of business.

9-5. Login Attempts: Add an attribute called `login_attempts` to your `User` class from Exercise 9-3 (page 166). Write a method called `increment_login_attempts()` that increments the value of `login_attempts` by 1. Write another method called `reset_login_attempts()` that resets the value of `login_attempts` to 0.

Make an instance of the `User` class and call `increment_login_attempts()` several times. Print the value of `login_attempts` to make sure it was incremented properly, and then call `reset_login_attempts()`. Print `login_attempts` again to make sure it was reset to 0.

TRY IT YOURSELF

9-6. Ice Cream Stand: An ice cream stand is a specific kind of restaurant. Write a class called `IceCreamStand` that inherits from the `Restaurant` class you wrote in Exercise 9-1 (page 166) or Exercise 9-4 (page 171). Either version of the class will work; just pick the one you like better. Add an attribute called `flavors` that stores a list of ice cream flavors. Write a method that displays these flavors. Create an instance of `IceCreamStand`, and call this method.

9-7. Admin: An administrator is a special kind of user. Write a class called `Admin` that inherits from the `User` class you wrote in Exercise 9-3 (page 166) or Exercise 9-5 (page 171). Add an attribute, `privileges`, that stores a list of strings like "can add post", "can delete post", "can ban user", and so on. Write a method called `show_privileges()` that lists the administrator's set of privileges. Create an instance of `Admin`, and call your method.

9-8. Privileges: Write a separate `Privileges` class. The class should have one attribute, `privileges`, that stores a list of strings as described in Exercise 9-7. Move the `show_privileges()` method to this class. Make a `Privileges` instance as an attribute in the `Admin` class. Create a new instance of `Admin` and use your method to show its privileges.

9-9. Battery Upgrade: Use the final version of `electric_car.py` from this section. Add a method to the `Battery` class called `upgrade_battery()`. This method should check the battery size and set the capacity to 85 if it isn't already. Make an electric car with a default battery size, call `get_range()` once, and then call `get_range()` a second time after upgrading the battery. You should see an increase in the car's range.