Encapsulation

Encapsulation is one of the key concepts of object-oriented languages like **Python**, **Java**, etc. Encapsulation 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.

Encapsulation is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit. In encapsulation, the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class.

Access Modifiers

public Access Modifier : The public member is accessible from inside or outside the class.

private Access Modifier: The private member is accessible only inside class. Define a private member by prefixing the member name with two underscores.

protected Access Modifier: The protected member is accessible. from inside the class and its sub-class. Define a protected member by prefixing the member name with an underscore.

```
# Python program for demonstrating protected members
  # first, we will create the base class
class Base1:
    def __init__(self):
            # the protected member
            self._p = 78
      # here, we will create the derived class
     class Derived1(Base):
        def __init__(self):
      # now, we will call the constructor of Base class
            Base1.__init__(self)
            print ("We will call the protected member of base class: ",
               self._p)
```

```
  # Now, we will be modifing the protected variable:
        self._p = 433
        print ("we will call the modified protected member outside the class: ",
           self._p)
    obj_1 = Derived1()
  obj_2 = Base1()
  # here, we will call the protected member
# this can be accessed but it should not be done because of convention
print ("Access the protected member of obj_1: ", obj_1._p)
  # here, we will access the protected variable outside
print ("Access the protected member of obj_2: ", obj_2._p)
```

Program to demonstrate private member

```
class Base1:
    def __init__(self):
        self.p = "Javatpoint"
        self.__q = "Javatpoint"
  # Creating a derived class
class Derived1(Base1):
     def __init__(self):
  # Calling constructor of
# Base class
     Base1.__init__(self)
     print("We will call the private member of base class: ")
     print(self.__q)
    # Driver code
obj_1 = Base1()
print(obj_1.p)
```

Polymorphism

The word polymorphism means having many forms. In programming, polymorphism means the same function name (but different signatures) being used for different types. The key difference is the data types and number of arguments used in function.

Example 1:

```
# morphic functions
# len() being used for a string
print(len("Medi-Caps University"))
# len() being used for a list
print(len([10, 20, 30]))
```

Example 2:

```
def add(x, y, z = 0):
        return x + y+z
print(add(2, 3))
print(add(2, 3, 4))
```

Example 3:

```
print(len("Medi-Caps University"))
print(len(["Python", "Java", "C"]))
print(len({"Name": "John", "Address": "Nepal"}))
```

Class Polymorphism

We can use the concept of polymorphism while creating class methods as Python allows different classes to have methods with the same name.

```
class Cat:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def info(self):
        print(f"I am a cat. My name is {self.name}. I am {self.age} years old.")
    def make_sound(self):
        print("Meow")
class Dog:
```

```
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def info(self):
        print(f"I am a dog. My name is {self.name}. I am {self.age} years old.")
    def make_sound(self):
        print("Bark")
cat1 = Cat("Kitty", 2.5)
dog1 = Dog("Fluffy", 4)
for animal in (cat1, dog1):
    animal.make_sound()
    animal.info()
    animal.make_sound()
```

Classes in Python

In Python, a class is a user-defined data type that contains both the data itself and the methods that may be used to manipulate it. In a sense, classes serve as a template to create objects. They provide the characteristics and operations that the objects will employ.

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.

- Classes are created by keyword class.
- Attributes are the variables that belong to a class.
- Attributes are always public and can be accessed using the dot (.) operator. Eg.: My class.Myattribute

Creating a Python Class

```
class Dog:
        sound = "bark"
```

Object of Python Class

An Object is an instance of a Class. A class is like a blueprint while an instance is a copy of the class with *actual values*. It's not an idea anymore, it's an actual dog, like a dog of breed pug who's seven years old. You can have many dogs to create many different instances, but without the class as a guide, you would be lost, not knowing what information is required.
An object consists of:

- **State:** It is represented by the attributes of an object. It also reflects the properties of an object.
- **Behavior:** It is represented by the methods of an object. It also reflects the response of an object to other objects.
- **Identity:** It gives a unique name to an object and enables one object to interact with other objects.

Example

```
class GFG:

        def __init__(self, name, company):

                self.name = name

                self.company = company

        def show(self):

                print("Hello my name is " + self.name+" and I" +

                        " work in "+self.company+".")

obj = GFG("John", "Medi-Caps University")

obj.show()
```

__init__() method

The __init__ method is similar to constructors in C++ and Java. Constructors are used to initializing the object's state. Like methods, a constructor also contains a collection of statements(i.e. instructions) that are executed at the time of Object creation. It runs as soon as an object of a class is instantiated. The method is useful to do any initialization you want to do with your object.

Example:

```
# Sample class with init method

class Person:

        # init method or constructor

        def __init__(self, name):

                self.name = name

        # Sample Method

        def say_hi(self):

                print('Hello, my name is', self.name)

p = Person('Nikhil')

p.say_hi()
```

__str__() method

Python has a particular method called **__str__()**. that is used to define how a **class** object should be represented as a string. It is often used to give an object a human-readable textual representation, which is helpful for logging, debugging, or showing users object information. When a class object is used to create a string using the built-in functions print() and str(), the **__str__()** function is automatically used. You can alter how objects of a **class** are represented in strings by defining the **__str__()** method.

Example:

```
class GFG:

        def __init__(self, name, company):

                self.name = name

                self.company = company

        def __str__(self):

                return f"My name is {self.name} and I work in {self.company}."
```

```
my_obj = GFG("John", "Medi-Caps University")

print(my_obj)
```

Constructors & Destructors

In **object oriented programming**, both **constructor and destructor** are the **member functions** of a class having the same name as the class.
A **constructor** helps in initialization of an object, i.e., it allocates memory to an object. On the other hand, a **destructor** deletes the created constructor when it is of no use which means it deallocates the memory of an object.

Difference between Constructor and Destructor

Constructor	Destructor
Constructors help allocate memory to an object.	Destructors deallocate the memory of an object.
Constructors can take arguments.	Destructors don't take any arguments.
Constructors are called automatically when an object is created.	Destructors are called automatically when the block is exited or when the program terminates.
Constructors allow an object to initialize a value before it is used.	They allow objects to execute code when it is being destroyed.
They are called in the successive order of their creation.	They are called in the reverse order of their creation.
There can be multiple constructors in a single class.	There is a single destructor in a class.
Constructors can be overloaded.	Destructors cannot be overloaded.
The concept of copy constructor allows an object to get initialized from another object.	There is no such concept in the case of destructors.

__del__()

The **__del__()** method is a known as a destructor method in Python. It is called when all references to the object have been deleted i.e when an object is garbage collected.

Example

```
class Employee:
        # Initializing
        def __init__(self):
                print('Employee created.')
        # Deleting (Calling destructor)
        def __del__(self):
                print('Destructor called, Employee deleted.')
obj = Employee()
del obj
```

Multiple inheritance

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.

Multiple-Inheritance

When a class is derived from more than one base class it is called multiple Inheritance. The derived class inherits all the features of the base case.

Example

```
# Python Program to depict multiple inheritance
# when method is overridden in both classes


class Class1:
        def m(self):
                print("In Class1")
```

```
class Class2(Class1):
        def m(self):
                print("In Class2")

class Class3(Class1):
        def m(self):
                print("In Class3")

class Class4(Class2, Class3):
        pass

obj = Class4()
obj.m()
```

Operator Overloading in Python

Operator Overloading means giving extended meaning beyond their predefined operational meaning. For example operator + is used to add two integers as well as join two strings and merge two lists. It is achievable because '+' operator is overloaded by int class and str class. You might have noticed that the same built-in operator or function shows different behavior for objects of different classes, this is called *Operator Overloading*.

Example

```
# + operator for different purposes.
print(1 + 2)
# concatenate two strings
print("ABC"+"For")
# Product two numbers
print(3 * 4)
# Repeat the String
print("XYZ"*4)
```

Example 2

```
# to overload an binary + operator
class A:
        def __init__(self, a):
                self.a = a
        # adding two objects
        def __add__(self, o):
                return self.a + o.a
ob1 = A(1)
ob2 = A(2)
ob3 = A("ABC")
ob4 = A("For")
print(ob1 + ob2)
print(ob3 + ob4)
# Actual working when Binary Operator is used.
print(A.__add__(ob1 , ob2))
print(A.__add__(ob3,ob4))
#And can also be Understand as :
print(ob1.__add__(ob2))
print(ob3.__add__(ob4))
```

Example 3:

```
# Python Program to perform addition  of two complex numbers using binary  + operator overloading.
class complex:
        def __init__(self, a, b):
                self.a = a
                self.b = b
```

```
        # adding two objects

        def __add__(self, other):

                return self.a + other.a, self.b + other.b

Ob1 = complex(1, 2)

Ob2 = complex(2, 3)

Ob3 = Ob1 + Ob2

print(Ob3)
```

Emulating built-in types:

Python includes built-in mathematical data structures like complex numbers, floating-point numbers, and integers. But occasionally we might want to develop our own custom-behaved number classes. Here, the idea of imitating number classes is put into use. We can create objects that can be used in the same way as native numeric classes by simulating them.

Example :

Emulating Addition

```
class MyNumber:
    def __init__(self, x):
        self.x = x
    def __add__(self, other):
        return MyNumber(self.x + other.x)
a = MyNumber(2)
b = MyNumber(3)
c = a + b
print(c.x)
```

Emulating Comparison

```
class MyNumber:
    def __init__(self, x):
        self.x = x
    def __eq__(self, other):
        return self.x == other.x
a = MyNumber(2)
b = MyNumber(3)
c = MyNumber(2)
print(a == b)
print(a == c)
```