Object Oriented Programming

Python is a multi-paradigm programming language. It supports different programming approaches.

One of the popular approaches to solve a programming problem is by creating objects. This is known as Object-Oriented Programming (OOP).

An object has two characteristics:

1.attributes
2.behaviour
Let's take an example:

A parrot is an object, as it has the following properties:

name, age, color as attributes
singing, dancing as behavior

The concept of OOP in Python focuses on creating reusable code. This concept is also known as DRY (Don't Repeat Yourself).

In Python, the concept of OOP follows some basic principles:

Class
A class is a blueprint for the object.

We can think of class as a sketch of a parrot with labels. It contains all the details about the name, colors, size etc. Based on these descriptions, we can study about the parrot. Here, a parrot is an object.

The example for class of parrot can be :

```
class Parrot:
    pass
```
Here, we use the class keyword to define an empty class Parrot. From class, we construct instances. An instance is a specific object created from a particular class.

Object

An object (instance) is an instantiation of a class. When class is defined, only the description for the object is defined. Therefore, no memory or storage is allocated.

The example for object of parrot class can be:

```
obj = Parrot()
```
Here, obj is an object of class Parrot.

Suppose we have details of parrots. Now, we are going to show how to build the class and objects of parrots.

Example 1: Creating Class and Object in Python

```
class Parrot:

    # class attribute
    species = "bird"

    # instance attribute
    def __init__(self, name, age):
        self.name = name
        self.age = age

# instantiate the Parrot class
blu = Parrot("Blu", 10)
woo = Parrot("Woo", 15)
```

```
# access the class attributes
print("Blu is a {}".format(blu.__class__.species))
print("Woo is also a {}".format(woo.__class__.species))

# access the instance attributes
print("{} is {} years old".format( blu.name, blu.age))
print("{} is {} years old".format( woo.name, woo.age))
```
Run Code
Output

```
Blu is a bird
Woo is also a bird
Blu is 10 years old
Woo is 15 years old
```

In the above program, we created a class with the name Parrot. Then, we define attributes. The attributes are a characteristic of an object.

These attributes are defined inside the __init__ method of the class. It is the initializer method that is first run as soon as the object is created.

Then, we create instances of the Parrot class. Here, blu and woo are references (value) to our new objects.

We can access the class attribute using __class__.species. Class attributes are the same for all instances of a class. Similarly, we access the instance attributes using blu.name and blu.age. However, instance attributes are different for every instance of a class.

To learn more about classes and objects, go to Python Classes and Objects

Methods

Methods are functions defined inside the body of a class. They are used to define the behaviors of an object.

Example 2 : Creating Methods in Python

```
class Parrot:
```

```
    # instance attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # instance method
    def sing(self, song):
        return "{} sings {}".format(self.name, song)

    def dance(self):
        return "{} is now dancing".format(self.name)

# instantiate the object
blu = Parrot("Blu", 10)

# call our instance methods
print(blu.sing("'Happy'"))
print(blu.dance())
```

Run Code

Output

```
Blu sings 'Happy'
Blu is now dancing
```

In the above program, we define two methods i.e sing() and dance(). These are called instance methods because they are called on an instance object i.e blu.

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).

Example 3: 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()
```

Run Code

Output

```
Bird is ready
Penguin is ready
Penguin
Swim faster
Run faster
```

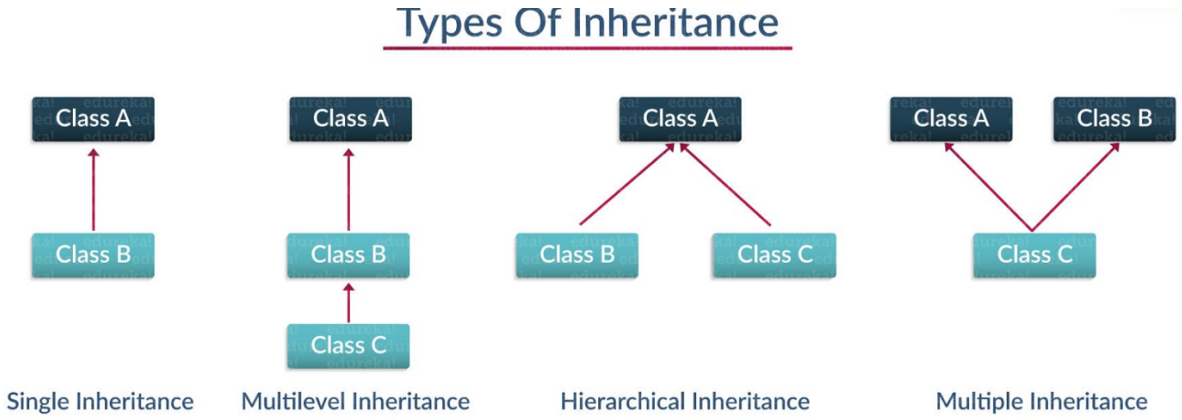
In the above program, we created two classes i.e. Bird (parent class) and Penguin (child class). The child class inherits the functions of parent class. We can see this from the swim() method.

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.

Types of Inheritance in Python

Types of Inheritance depend upon the number of child and parent classes involved. There are four types of inheritance in Python:



Single Inheritance:
Single inheritance enables a derived class to inherit properties from a single parent class, thus enabling code reusability and the 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.
```
Multiple Inheritance:
When a class can be derived from more than one base class this type of inheritance is called multiple inheritances. In multiple inheritances, 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
```

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 a grandfather.

Example:

```
# Python program to demonstrate
# multilevel inheritance

# Base class
class Grandfather:

    def __init__(self, grandfathername):
        self.grandfathername = grandfathername

# Intermediate class


class Father(Grandfather):
    def __init__(self, fathername, grandfathername):
        self.fathername = fathername

        # invoking constructor of Grandfather class
        Grandfather.__init__(self, grandfathername)

# Derived class


class Son(Father):
    def __init__(self, sonname, fathername, grandfathername):
        self.sonname = sonname
```

```
        # invoking constructor of Father class
        Father.__init__(self, fathername, grandfathername)

    def print_name(self):
        print('Grandfather name :', self.grandfathername)
        print("Father name :", self.fathername)
        print("Son name :", self.sonname)


#  Driver code
s1 = Son('Prince', 'Rampal', 'Lal mani')
print(s1.grandfathername)
s1.print_name()
```

Output:

```
Lal mani
Grandfather name : Lal mani
Father name : Rampal
Son name : Prince
```

Hierarchical Inheritance:

When more than one derived class are created from a single base this type of inheritance 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.
```

Hybrid Inheritance:

Inheritance consisting of multiple types of inheritance is called hybrid inheritance.

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
.
```

Object-oriented programming (OOP) is a notion that depends on the concept of objects. In OOP, objects are defined with their own set of attributes/properties.

It is important because it helps the developers in writing clean, modular code that can be reused throughout the development. . With modular coding, we get control of functions and modules.

It comes in handy primarily in the case of large application development.

Class, Instance/Object, __init__method

If you are a beginner, pay a good amount of time to understand the terminology explained below.

Class: It is a user-defined blueprint of an object with a predefined set of attributes common .

Instance/Object: It is an individual entity that is created from a class.

__init__ method: __init__ method in OOP is nothing but a special function. Special functions are the functions that are used to enrich the class. These can be easily identified as they have double underscores on either side. __init__ method is used to initialize the attributes. It is called a constructor in other programming languages.

Creating Classes and Objects

Creating a class: The *class* statement creates a new class with a given ClassName.

The name of the class in Python follows Pascal Case. It is a naming convention where each word starts with a capital letter without any special characters used.

Initializing the attributes/variables: We are initializing the object with n attributes namely attr1, attr2,…., attrn.

```
def __init__(self, attr1, attr2,...attrn):
    self.attr1 = attr1
    self.attr2 = attr2
    .
    .
    .
    self.attrn = attrn
```

Creating Methods: Method is nothing but a function with a set of rules based on which objects behave. Created two methods names method1, method2. Choose the method inputs based on the requirement. Here, method1 does not take any inputs other than the object. Whereas, method2 takes self.attr2 and does something to it.

```
def method1(self):
    # code

def method2(self, self.attr2):
    # this method does something to the attr2
```

Complete Syntax of creating a class:

```
class ClassName:
        def __init__(self, attr1, attr2):

        self.attr1 = attr1

        self.attr2 = attr2

def method1(self):
```

```
pass

def method2(self, attr2):

pass
```

Example Clas

Then create two methods – one that prints the author's details and another that updates the number of articles written by an author and are published.

```
class BlogathonAuthors:

    def __init__(self, author_name, num_articles):

    self.author_name = author_name

    self.num_articles = num_articles

    print("Created new author object")

def show(self):

    """This method prints the details of the author"""

    print("In show method")

    print(f"Author Name: {self.author_name}nNum of published articles:
    {self.num_articles}")

def update(self, num_articles):

    """This method updates the number of published articles"""

    print("In update method")

    self.num_articles = num_articles
```

Creating Instances/Objects:

The process of creating an instance or object from a class is called Instantiation. While creating an object, we should pass the arguments that are defined in the __init__ method.

Syntax: object = ClassName(arguments)

```
author1 = BlogathonAuthors("Harika", 10)
```

The above code creates an object named “author1” whose name is “Harika” and has written 10 articles.

Similarly, we can create any number of objects required.

```
author2 = BlogathonAuthors("Joey", 23)
```

Accessing attributes and Calling methods

The syntax for accessing the attributes is object.attribute

The author’s name is accessed using author1.author_name, and the number of articles is accessed using author1.num_articles.

```
author1.author_name
```

```
'Harika'
```

```
author1.num_articles
```

```
10
```

Rather than just displaying we can also change the values.

```
author1.num_articles = 9
```

Calling Methods: The two ways of calling methods are

ClassName.method(object) or object.method()

Calling the show method to display the information about author1.

BlogathonAuthors.show(author1)

```
BlogathonAuthors.show(author1)
```

```
In show method
Author Name: Harika
Num of published articles: 9
```

author1.show()

```
author1.show()
```

```
In show method
Author Name: Harika
Num of published articles: 9
```

Wait, if you are familiar with functions in Python, you may get a doubt that the “show” method accepts one argument but we didn’t pass any.

Think of a minute about what’s happening here and continue reading further to know the answer.

show(self) method accepts one argument that is the object itself. The self keyword here points to the instance/object of the class. So when we call object.method(), it is nothing but we are passing the object as an argument.

Now, calling the update method and changing the number of articles.

author1.update(20)

After the update, if we see the details of author1, the number of articles will be 20.

```
author1.show()
```

```
In show method
Author Name: Harika
Num of published articles: 9
```

```
author1.update(20)
```

```
In update method
```

```
author1.show()
```

```
In show method
Author Name: Harika
Num of published articles: 20
```

Public Access Modifier:

The members of a class that are declared public are easily accessible from any part of the program. All data members and member functions of a class are public by default.

program to illustrate public access modifier in a class

class PythonLife:

    # constructor
    def __init__(self, name, age):

        # public data members
        self.PythonLifeName = name

```
        self.PythonLifeAge = age

    # public member function
    def displayAge(self):

        # accessing public data member
        print("Age: ", self.PythonLifeAge)

# creating object of the class
obj = PythonLife("R2J", 20)

# accessing public data member
print("Name: ", obj.PythonLifeName)

# calling public member function of the class
obj.displayAge()
```

Output:
Name: R2J
Age: 20

In the above program, PythonLifeName and PythonLifeAge are public data members and displayAge() method is a public member function of the class PythonLife. These data members of the class PythonLife can be accessed from anywhere in the program.

Protected Access Modifier:

The members of a class that are declared protected are only accessible to a class derived from it. Data members of a class are declared protected by adding a single underscore '_' symbol before the data member of that class.

```
# program to illustrate protected access modifier in a class
```

```
# super class
class Student:

    # protected data members
    _name = None
    _roll = None
    _branch = None

    # constructor
    def __init__(self, name, roll, branch):
        self._name = name
        self._roll = roll
        self._branch = branch

    # protected member function
    def _displayRollAndBranch(self):

        # accessing protected data members
        print("Roll: ", self._roll)
        print("Branch: ", self._branch)


# derived class
class PythonLife(Student):

      # constructor
      def __init__(self, name, roll, branch):
            Student.__init__(self, name, roll, branch)

      # public member function
      def displayDetails(self):

            # accessing protected data members of super class
            print("Name: ", self._name)
```

```
        # accessing protected member functions of super class
        self._displayRollAndBranch()

# creating objects of the derived class
obj = PythonLife("R2J", 1706256, "Information Technology")

# calling public member functions of the class
obj.displayDetails()
```

Output:
Name: R2J
Roll: 1706256
Branch: Information Technology

In the above program, _name, _roll, and _branch are protected data members and _displayRollAndBranch() method is a protected method of the super class Student. The displayDetails() method is a public member function of the class PythonLife which is derived from the Student class, the displayDetails() method in PythonLife class accesses the protected data members of the Student class.

Private Access Modifier:
The members of a class that are declared private are accessible within the class only, private access modifier is the most secure access modifier. Data members of a class are declared private by adding a double underscore '__' symbol before the data member of that class.

```
# program to illustrate private access modifier in a class

class PythonLife:
```

```
    # private members
    __name = None
    __roll = None
    __branch = None

    # constructor
    def __init__(self, name, roll, branch):
        self.__name = name
        self.__roll = roll
        self.__branch = branch

    # private member function
    def __displayDetails(self):

        # accessing private data members
        print("Name: ", self.__name)
        print("Roll: ", self.__roll)
        print("Branch: ", self.__branch)

    # public member function
    def accessPrivateFunction(self):

        # accessing private member function
        self.__displayDetails()

# creating object
obj = PythonLife("R2J", 1706256, "Information Technology")

# calling public member function of the class
obj.accessPrivateFunction()
```

Output:

```
Name:  R2J
Roll:  1706256
Branch:  Information Technology
```

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 __.

Example : 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()
```

Run Code

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.
Here, notice the code

c.__maxprice = 1000
Here, we have tried to modify the value of __maxprice outside of the class. However, since __maxprice is a private variable, this modification is not seen on the output.

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.

Example : 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.