OOPS Concepts in Python

Programming languages are emerging constantly, and so are different methodologies.

Object-oriented programming is one such methodology that has become quite popular over past few years.

What is Object Oriented Programming?

Object Oriented means directed towards objects. In other words, it means functionally directed towards modelling objects. This is one of the many techniques used for modelling complex systems by describing a collection of interacting objects via their data and behavior.

Python, an Object-Oriented programming (OOP), is a way of programming that focuses on using objects and classes to design and build applications. Major pillars of Object-Oriented Programming (OOP) are **Inheritance**, **Polymorphism**, **Abstraction**, and **Encapsulation**.

Why to Choose Object-Oriented Programming?

Python was designed with an object-oriented approach. OOP offers the following advantages:

- Provides a clear program structure, which makes it easy to map real world problems and their solutions.
- Facilitates easy maintenance and modification of existing code.
- Enhances program modularity because each object exists independently and new features can be added easily without disturbing the existing ones.
- Presents a good framework for code libraries where supplied components can be easily adapted and modified by the programmer.
- Imparts code reusability.

Let us understand each of the pillars of object-oriented programming in brief:

Encapsulation

This property hides unnecessary details and makes it easier to manage the program structure. Each object's implementation and state are hidden behind well-defined boundaries and that provides a clean and simple interface for working with them. One way to accomplish this is by making the data private.

Inheritance

Inheritance, also called generalization, allows us to capture a hierarchal relationship between classes and objects. For instance, a 'fruit' is a generalization of 'orange'. Inheritance is very useful from a code reuse perspective.

Abstraction

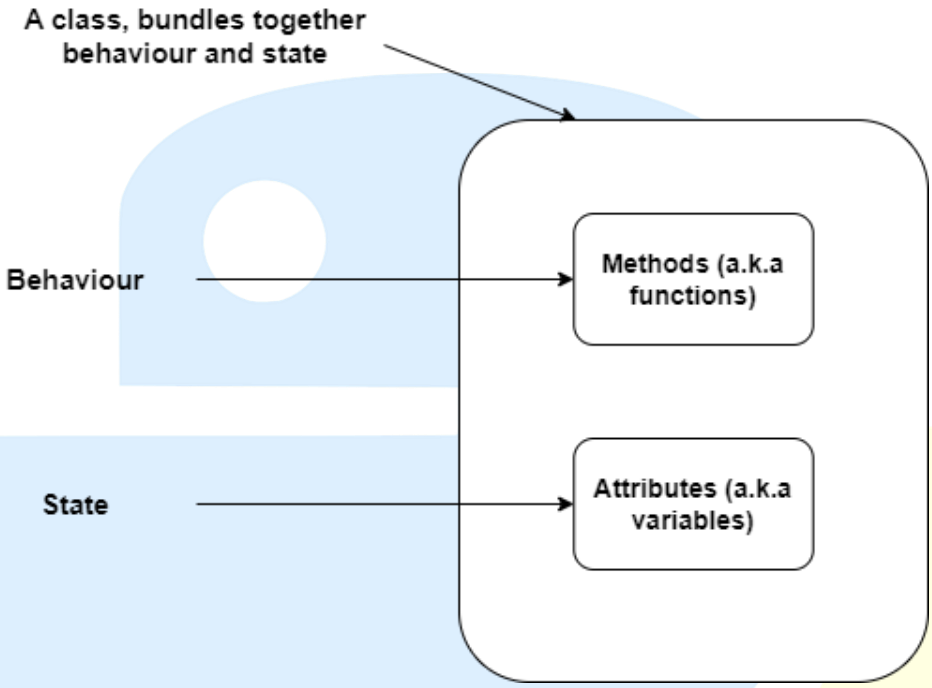
This property allows us to hide the details and expose only the essential features of a concept or object. For example, a person driving a scooter knows that on pressing a horn, sound is emitted, but he has no idea about how the sound is generated on pressing the horn.

Polymorphism

Poly-morphism means many forms. That is, a thing or action is present in different forms or ways. One good example of polymorphism is constructor overloading in classes.

Classes

A class will let you bundle together the behavior and state of an object.



The following points are worth notable when discussing class bundles:

- The word **behavior** is identical to **function** – it is a piece of code that does something (or implements a behavior).
- The word **state** is identical to **variables** – it is a place to store values within a class.
- When we assert a class behavior and state together, it means that a class packages functions and variables.

Class has methods and attributes

In Python, creating a method defines a class behavior. The word method is the OOP name given to a function that is defined within a class. To sum up:

- **Class functions** is synonym for **methods.**
- **Class variables** is synonym for **name attributes**.
- **Class** – a blueprint for an instance with exact behavior.
- **Object** – one of the instances of the class, perform functionality defined in the class.
- **Type** – indicates the class the instance belongs to
- **Attribute** – Any object value: object.attribute
- **Method** – a "callable attribute" defined in the class

Creating a **class** and **objects** in python:

```
class People:
    legs = 2
    hands = 2

    def greet(self):
        print("Hello this is a greet method")

    def leave(self):
        print("Hello this is a leave method")


a = People()
print(a.legs)
a.greet()
a.leave()

b = People()
print(b.hands)
b.greet()
b.leave()
```

Output

```
2
Hello this is a greet method
Hello this is a leave method
2
Hello this is a greet method
Hello this is a leave method
```

Understand the following concepts:

- We created a class named **People.**
- There are two attributes (variables) associated with the class.
- There are two methods (functions) associated with the class.
- To access the attributes and methods in the class, we need to create objects. We have created 2 objects named ***a*** and ***b***.
- From ***a***, we can access the attributes and methods of the classes

A function defined in a class is called a **method**. An instance method requires an instance in order to call it and requires no decorator. When creating an instance method, the first parameter is always **self**. Though we can call it (self) by any other name, it is recommended to use self, as it is a naming convention.

```
class MyClass(object):
    var = 9

    def first(self):
        print("hello, World")


obj = MyClass()
print(obj.var)
obj.first()
```

Output

```
9
hello, World
```

Encapsulation

Encapsulation is one of the fundamentals of OOP. OOP enables us to hide the complexity of the internal working of the object which is advantageous to the developer in the following ways:

- Simplifies and makes it easy to understand to use an object without knowing the internals.
- Simplifies and makes it easy to understand to use an object without knowing the internals.

Encapsulation provides us the mechanism of restricting the access to some of the object's components, this means that the internal representation of an object cannot be seen from outside of the object definition. Access to this data is typically achieved through special methods: **Getters** and **Setters**.

```
class Human:
    def setGender(self, g):
        self.gender = g

    def getGender(self):
        return self.gender


obj1 = Human()
obj1.setGender("Male")
print(obj1.getGender())

obj2 = Human()
obj2.setGender("Female")
print(obj2.getGender())
```

Output

```
Male
Female
```

We can see, now we have declared a variable/attribute using **self**-keyword. This will create a class wide variable for that object and we can access it from outside.

INIT Constructor

The **__init__** method is implicitly called as soon as an object of a class is instantiated.

This will initialize the object.

```
obj1 = Human()
```

The line of code shown above will create a new instance and assigns this object to the local variable **obj1**.

The instantiation operation, that is calling a class object, creates an empty object. Many classes like to create objects with instances customized to a specific initial state. Therefore, a class may define a special method named '**__init__()**' as shown:

```
def __init__(self):
    self.legs = 2
    self.hands = 2
```

The **__init__()** method can have single or multiple arguments for a greater flexibility. The **init** stands for initialization, as it initializes attributes of the instance. It is called the constructor of a class.

```
def __init__(self, name):
    self.legs = 2
    self.hands = 2
    self.name = name
```

See the code below for better understanding

```
class Human:

    def __init__(self, name):
        self.legs = 2
        self.hands = 2
        self.name = name

    def displayInfo(self):
        print(f"My name is {self.name} and I have {self.legs}
legs and {self.hands} hands")


obj1 = Human("Elon")
obj2 = Human("Akshay")

obj1.displayInfo()
obj2.displayInfo()
```

Output

```
My name is Elon and I have 2 legs and 2 hands
My name is Akshay and I have 2 legs and 2 hands
```

Inheritance and Polymorphism

One of the major advantages of Object-Oriented Programming is re-use. Inheritance is one of the mechanisms to achieve the same. Inheritance allows programmer to create a general or a base class first and then later extend it to more specialized class. It allows programmer to write better code.

Using inheritance, you can use or inherit all the data fields and methods available in your base class. Later you can add you own methods and data fields; thus, inheritance provides a way to organize code, rather than rewriting it from scratch.

In object-oriented terminology when class X extend class Y, then Y is called super/parent/base class and X is called subclass/child/derived class. One point to note here is that only data fields and method which are not private are accessible by child classes. Private data fields and methods are accessible only inside the class.

Syntax:

```
class BaseClass:
    Body of base class

class DerivedClass(BaseClass):
    Body of derived class
```

See the example below:

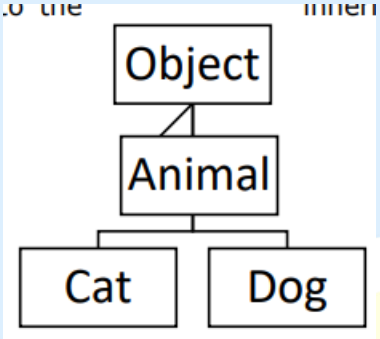
```
class Human:
    def getHuman(self):
        print("I am a human")


# Class Student inheriting class Human
class Student(Human):
    def getStudent(self):
        print("I am a student")


h = Human()
h.getHuman()

s = Student()
s.getHuman()
s.getStudent()
```

Output

```
I am a human
I am a human
I am a student
```

We first created a class called **Human**. Later we created another class called **Student** and called the **Human** class as an argument. Through this call we get access to all the data and attributes of **Human** class into the **Student** class. Because of that when we try to get the **getHuman()** method from the **Student** class object we created earlier possible.

Inheritance Examples



Let us create couple of classes to participate in examples:

- Animal: Class simulate an animal
- Cat: Subclass of Animal
- Dog: Subclass of Animal

In Python, constructor of class used to create an object (instance), and assign the value for the attributes.

Constructor of subclasses always called to a constructor of parent class to initialize value for the attributes in the parent class, then it starts assign value for its attributes.

```python
#Inheritance Example
class Animal(object):

    def __init__(self, name):
        self.name = name
    def eat(self, food):          # Common method(or property) of both subclass
        print ('%s is eating %s. ' %(self.name, food))

class Dog(Animal):

    def fetch(self, thing):
        print('%s goes after the %s!' %(self.name, thing))

class Cat(Animal):

    def swatstring(self):
        print('%s shreds the string!' %(self.name))

d = Dog('Ranger')        # Created Dog object, d
c = Cat('MeOw')          # Created Cat object, c

d.fetch('ball')         # Rover goes after the paper!=
c.swatstring()           # Fluffy shreds the string!
d.eat('Dog Food')        # Rover is eating dog Food
c.eat('Cat Food')        #Fluffy is eating cat food.
d.swatstring()           #Attribute Error: 'Dog' object has no Attribute 'swatstring'
```

Output

```
Ranger goes after the ball!
MeOw shreds the string!
Ranger is eating Dog Food.
MeOw is eating Cat Food.
Traceback (most recent call last):
  File "animal.py", line 27, in <module>
    d.swatstring()          #Attribute Error: 'Dog' object has no Attribute 'swatstring'
AttributeError: 'Dog' object has no attribute 'swatstring'
```

In the above example, we see the command attributes or methods we put in the parent class so that all subclasses or child classes will inherits that property from the parent class.

If a subclass try to inherits methods or data from another subclass then it will through an error as we see when Dog class try to call swatstring() methods from that cat class, it throws an error(like AttributeError in our case).

Polymorphism (“MANY SHAPES”)

Polymorphism is an important feature of class definition in Python that is utilized when you have commonly named methods across classes or subclasses. This permit functions to use entities of different types at different times. So, it provides flexibility and loose coupling so that code can be extended and easily maintained over time.

This allows functions to use objects of any of these polymorphic classes without needing to be aware of distinctions across the classes.

Polymorphism can be carried out through inheritance, with subclasses making use of base class methods or overriding them.

Let us see an example below:

```
class Animal:
    def speak(self):
        print("Animal is speaking")


class Cat(Animal):
    def speak(self):
        print("I am meowing")


class Dog(Animal):
    def speak(self):
        print("I am barking")


c = Cat()
d = Dog()
a = Animal()

a.speak()
c.speak()
d.speak()
```

Output

```
Animal is speaking
I am meowing
I am barking
```

As we can see the code, ***speak*** method is in all the classes but has different forms depending upon the class it is in.

Python itself have classes that are polymorphic. Example, the len() function can be used with multiple objects and all return the correct output based on the input parameter.