Object Oriented Programming

Using Python

- In Python, object-oriented Programming (OOPs) is a programming paradigm that uses objects and classes in programming.
- It aims to implement real-world entities like inheritance, polymorphisms, encapsulation, etc. in the programming.
- The main concept of OOPs is to bind the data and the functions that work on that together as a single unit so that no other part of the code can access this data.

OOPs Concepts in Python

- Class
- Objects
- Polymorphism
- Encapsulation
- Inheritance
- Data Abstraction

Python Class

- A class is a collection of objects. A class contains the blueprints or the prototype from which the objects are being created. It is a logical entity that contains some attributes and methods.
- To understand the need for creating a class let’s consider an example, let’s say you wanted to track the number of dogs that may have different attributes like breed, and age. If a list is used, the first element could be the dog’s breed while the second element could represent its age. Let’s suppose there are 100 different dogs, then how would you know which element is supposed to be which? What if you wanted to add other properties to these dogs? This lacks organization and it’s the exact need for classes.

- 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.: Myclass.Myattribute

Class Definition Syntax:

```
class ClassName:
# Statement-1
.
.
.
# Statement-N
```

Creating an Empty Class in Python

- In the below example, we have created a class named Dog using the class keyword.

```
# Python3 program to
# demonstrate defining
# a class

class Dog:
	pass
```

Python Objects

- The object is an entity that has a state and behavior associated with it. It may be any real-world object like a mouse, keyboard, chair, table, pen, etc. Integers, strings, floating-point numbers, even arrays, and dictionaries, are all objects. More specifically, any single integer or any single string is an object. The number 12 is an object, the string “Hello, world” is an object, a list is an object that can hold other objects, and so on. You’ve been using objects all along and may not even realize it.

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.

To understand the state, behavior, and identity let us take the example of the class dog (explained above).

- The identity can be considered as the name of the dog.
- State or Attributes can be considered as the breed, age, or color of the dog.
- The behavior can be considered as to whether the dog is eating or sleeping.

Creating an Object

- This will create an object named obj of the class Dog defined above. Before diving deep into objects and classes let us understand some basic keywords that will we used while working with objects and classes.
- obj = Dog()

The Python self

1. Class methods must have an extra first parameter in the method definition. We do not give a value for this parameter when we call the method, Python provides it
2. If we have a method that takes no arguments, then we still have to have one argument.
3. This is similar to this pointer in C++ and this reference in Java.

- When we call a method of this object as myobject.method(arg1, arg2), this is automatically converted by Python into MyClass.method(myobject, arg1, arg2) – this is all the special self is about.

The Python __init__ Method

- The __init__ method is similar to constructors in C++ and Java. It is run 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. Now let us define a class and create some objects using the self and __init__ method.

Creating a class and object with class and instance attributes

- class Dog:

- 		# class attribute
- 		attr1 = "mammal"

- 		# Instance attribute
- 		def __init__(self, name):
- 				self.name = name

- # Driver code
- # Object instantiation
- Rodger = Dog("Rodger")
- Tommy = Dog("Tommy")

- # Accessing class attributes
- print("Rodger is a {}".format(Rodger.__class__.attr1))
- print("Tommy is also a {}".format(Tommy.__class__.attr1))

- # Accessing instance attributes
- print("My name is {}".format(Rodger.name))
- print("My name is {}".format(Tommy.name))

Output

```
Rodger is a mammal
Tommy is also a mammal
My name is Rodger
My name is Tommy
```

Creating Classes and objects with methods

- Here, The Dog class is defined with two attributes:
- attr1 is a class attribute set to the value “mammal”. Class attributes are shared by all instances of the class.
- __init__ is a special method (constructor) that initializes an instance of the Dog class. It takes two parameters: self (referring to the instance being created) and name (representing the name of the dog). The name parameter is used to assign a name attribute to each instance of Dog.
  The speak method is defined within the Dog class. This method prints a string that includes the name of the dog instance.

- The driver code starts by creating two instances of the Dog class: Rodger and Tommy. The __init__ method is called for each instance to initialize their name attributes with the provided names.
- The speak method is called in both instances (Rodger.speak() and Tommy.speak()), causing each dog to print a statement with its name.

```
• class Dog:

•       # class attribute
•       attr1 = "mammal"

•       # Instance attribute
•       def __init__(self, name):
•           self.name = name
•
•       def speak(self):
•           print("My name is {}".format(self.name))

• # Driver code
• # Object instantiation
• Rodger = Dog("Rodger")
• Tommy = Dog("Tommy")

• # Accessing class methods
• Rodger.speak()
• Tommy.speak()
```

Output

- My name is Rodger
- My name is Tommy

Python Inheritance

- Inheritance is the capability of one class to derive or inherit the properties from another class. The class that derives properties is called the derived class or child class and the class from which the properties are being derived is called the base class or parent class. The benefits of inheritance are:
- It represents real-world relationships well.
- It provides the reusability of a code. We don’t have to write the same code again and again. Also, it allows us to add more features to a class without modifying it.
- It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.

Types of Inheritance

- **Single Inheritance**: Single-level inheritance enables a derived class to inherit characteristics from a single-parent class.
- **Multilevel Inheritance:** Multi-level inheritance enables a derived class to inherit properties from an immediate parent class which in turn inherits properties from his parent class.
- **Hierarchical Inheritance:** Hierarchical-level inheritance enables more than one derived class to inherit properties from a parent class.
- **Multiple Inheritance:** Multiple-level inheritance enables one derived class to inherit properties from more than one base class.

- we have created two classes i.e. Person (parent class) and Employee (Child Class). The Employee class inherits from the Person class.
- We can use the methods of the person class through the employee class as seen in the display function in the above code.
- A child class can also modify the behavior of the parent class as seen through the details() method.

```
• # Python code to demonstrate how parent constructors
• # are called.

• # parent class
• class Person(object):

•       # __init__ is known as the constructor
•       def __init__(self, name, idnumber):
•               self.name = name
•               self.idnumber = idnumber

•       def display(self):
•               print(self.name)
•               print(self.idnumber)
• 
•       def details(self):
•               print("My name is {}".format(self.name))
•               print("IdNumber: {}".format(self.idnumber))
• 
```

```
# child class
class Employee(Person):
        def __init__(self, name, idnumber, salary, post):
                self.salary = salary
                self.post = post

                # invoking the __init__ of the parent class
                Person.__init__(self, name, idnumber)

        def details(self):
                print("My name is {}".format(self.name))
                print("IdNumber: {}".format(self.idnumber))
                print("Post: {}".format(self.post))


# creation of an object variable or an instance
a = Employee('Rahul', 886012, 200000, "Intern")

# calling a function of the class Person using
# its instance
a.display()
a.details()
```

Output

- Rahul
- 886012
- My name is Rahul
- IdNumber: 886012
- Post: Intern

Assignment

- Design and implement a class representing a rectangle and operations like area,perimeter, change dimensions, report dimensions, on it. Write code to test your classes.