Assignment 06: 05 Feb 2023

Q1. Explain Class and Object with respect to Object-Oriented Programming. Give a suitable example.
Q2. Name the four pillars of OOPs.
Q3. Explain why the __init__() function is used. Give a suitable example.
Q4. Why is self used in OOPS?
Q5. What is inheritance? Give an example for each type of inheritance.

Q1. Explain Class and Object with respect to Object-Oriented Programming. Give a suitable example.

Ans:
Class:

- Class creates a user-defined data structure, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class.
- A class is like a blueprint for an object.
- Some points on Python 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.: Myclass.Myattribute

Object:

- An Object is an instance of a Class.
- An object consists of :
  - **State/Attribute:** It is represented by the attributes of an object. It also reflects the properties of an object.
  - **Behaviour:** 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.
- Declaring Objects (Also called instantiating a class)
  - When an object of a class is created, the class is said to be instantiated. All the instances share the attributes and the behaviour of the class. But the values of those attributes, i.e. the state are unique for each object. A single class may have any number of instances.

```
# create a class
class Room:
    length = 0.0
    breadth = 0.0
    # method to calculate area
    def calculate_area(self):
        print("Area of Room =", self.length * self.breadth)
# create object of Room class
study_room = Room()
# assign values to all the attributes
```

```
study_room.length = 42.5
study_room.breadth = 30.8
# access method inside class
study_room.calculate_area()
```

Q2. Name the four pillars of OOPs.

Inheritance, Polymorphism, Encapsulation and Abstraction

Q3. Explain why the __init__() function is used. Give a suitable example.

Ans:

__init__() is a special python method that runs when an object of a class is created.
__init__() function is mostly used for assigning values to newly created objects.
__init__() is a magic method which means it is called automatically by Python
__init__() can also be invoked manually.
__init__() also supports inheritance.

```
# Example of _init_()
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age
p1 = Person("John", 36)
print(p1.name)
print(p1.age)
```

Q4. Why is self used in OOPS?

Ans:

- self represents the instance of the class. By using the “self” we can access the attributes and methods of the class in python. It binds the attributes with the given arguments.
- The reason you need to use self. is because Python does not use the @ syntax to refer to instance attributes. Python decided to do methods in a way that makes the instance to which the method belongs be passed automatically, but not received automatically: the first parameter of methods is the instance the method is called on.
- In more clear way you can say that SELF has following Characteristic-
    - Self is always pointing to Current Object.
    - Self is the first argument to be passed in the Constructor and Instance Method.
    - Self is a convention and not a Python keyword .
    - we want to used another parameter name in place of self

Q5. What is inheritance? Give an example for each type of inheritance.

Ans:

Inheritance is **a process of obtaining properties and characteristics(variables and methods) of another class**.

There are five types of inheritances:

1. Single Inheritance
2. Multiple Inheritance
3. Multilevel Inheritance

4. Hierarchical Inheritance
5. Hybrid Inheritance

1. Single Inheritance

This type of inheritance enables a subclass or derived class to inherit properties and characteristics of the parent class, this avoids duplication of code and improves code reusability.

```
#parent class
class Above:
    i = 5
    def fun1(self):
        print(“Hey there, you are in the parent class”)
#subclass
class Below(Above):
    i=10
    def fun2(self):
        print(“Hey there, you are in the sub class”)
temp1=Below()
temp2=Above()
temp1.fun1()
temp1.fun2()
temp2.fun1()
print(temp1.i)
print(temp2.i)
#temp2.fun2()
```

2. Multiple Inheritance

This inheritance enables a child class to inherit from more than one parent class. This type of inheritance is not supported by java classes, but python does support this kind of inheritance. It has a massive advantage if we have a requirement of gathering multiple characteristics from different classes.

```
#parent class 1
class A:
    demo1=0
    def fun1(self):
        print(self.demo1)

#parent class 2
class B:
    demo2=0
    def fun2(self):
        print(self.demo2)
#child class
class C(A, B):
    def fun3(self):
        print(“Hey there, you are in the child class”)
# Main code
c = C()
c.demo1 = 10
```

```
c.demo2 = 5
c.fun3()
print("first number is : ",c.demo1)
print("second number is : ",c.demo2)
```

3. Multilevel Inheritance

In multilevel inheritance, the transfer of the properties of characteristics is done to more than one class hierarchically. To get a better visualisation we can consider it as an ancestor to grandchildren relation or a root to leaf in a tree with more than one level.

```
#parent class 1
class vehicle:
    def functioning(self):
        print("vehicles are used for transportation")
#child class 1
class car(vehicle):
    def wheels(self):
        print("a car has 4 wheels")
#child class 2
class electric_car(car):
    def speciality(self):
        print("electric car runs on electricity")
electric=electric_car()
electric.speciality()
electric.wheels()
electric.functioning()
```

4. Hierarchical Inheritance

This inheritance allows a class to host as a parent class for more than one child class or subclass. This provides a benefit of sharing the functioning of methods with multiple child classes, hence avoiding code duplication.

```
#parent class
class Parent:
    def fun1(self):
        print("Hey there, you are in the parent class")
#child class 1
class child1(Parent):
    def fun2(self):
        print("Hey there, you are in the child class 1")
#child class 2
class child2(Parent):
    def fun3(self):
        print("Hey there, you are in the child class 2")
#child class 3
class child3(Parent):
    def fun4(self):
        print("Hey there, you are in the child class 3")

# main program
```

```
child_obj1 = child3()
child_obj2 = child2()
child_obj3 = child1()
child_obj1.fun1()
child_obj1.fun4()
child_obj2.fun1()
child_obj2.fun3()
child_obj3.fun1()
child_obj3.fun2()
```

5. Hybrid Inheritance

An inheritance is said hybrid inheritance if more than one type of inheritance is implemented in the same code. This feature enables the user to utilise the feature of inheritance at its best. This satisfies the requirement of implementing a code that needs multiple inheritances in implementation.

```
class A:
def fun1(self):
print("Hey there, you are in class A")class B(A):
def fun2(self):
print("Hey there, you are in class B")class C(A):
def fun3(self):
print("Hey there, you are in class C")class D(C,A): #line 13
def fun4(self):
print("Hey there, you are in the class D")#main program
ref = D()
ref.fun4()
ref.fun3()
ref.fun1()
```