Assignment 6

August 31, 2023

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

ANS:

Class: A class is a user-defined data type that encapsulates data (attributes) and functions (methods) that operate on that data. It serves as a blueprint for creating objects of that class. A class defines the structure and behavior of its objects.

Object: An object is an instance of a class. It represents a specific entity or concept with its own set of attributes and behavior as defined by the class. Objects are created using the constructor of the class and can interact with each other by invoking their methods.

[1]:
```
# Defining a class
class Car:
    def __init__(self, make, model):
        self.make = make
        self.model = model
        self.speed = 0

    def accelerate(self, increment):
        self.speed += increment

    def brake(self, decrement):
        if self.speed >= decrement:
            self.speed -= decrement
        else:
            self.speed = 0
```

[2]:
```
# Creating objects (instances) of the Car class
car1 = Car("Toyota", "Camry")
car2 = Car("Tesla", "Model 3")
```

[6]:
```
# Using object methods to interact with the objects
car1.accelerate(80)
car2.accelerate(45)

print(f"{car1.make} {car1.model} Speed: {car1.speed} km/h")
print(f"{car2.make} {car2.model} Speed: {car2.speed} km/h")
```

```
Toyota Camry Speed: 270 km/h
Tesla Model 3 Speed: 155 km/h
```

[9]:
```
car1.brake(70)
car2.brake(50)

print(f"{car1.make} {car1.model} Speed after braking: {car1.speed} km/h")
print(f"{car2.make} {car2.model} Speed after braking: {car2.speed} km/h")
```

```
Toyota Camry Speed after braking: 140 km/h
Tesla Model 3 Speed after braking: 80 km/h
```

[ ]:
```

```

Q2. Name the four pillars of OOPs.

ANS:
The four pillars of Object-Oriented Programming (OOP) are:

Encapsulation: Encapsulation refers to the bundling of data (attributes) and the methods (functions) that operate on that data into a single unit called a class. It hides the internal implementation details of an object from the outside world, providing access only through well-defined interfaces. Encapsulation helps achieve data hiding and protects the integrity of the data.

Abstraction: Abstraction involves simplifying complex reality by modeling classes based on the essential properties and behaviors of objects, while ignoring irrelevant details. It allows programmers to focus on the high-level structure and functionality of objects without getting bogged down in implementation specifics.

Inheritance: Inheritance enables the creation of a new class (subclass or derived class) based on an existing class (base class or parent class). The subclass inherits the attributes and methods of the base class and can also extend or override them as needed. Inheritance promotes code reuse and the creation of hierarchical relationships among classes.

Polymorphism: Polymorphism allows objects of different classes to be treated as objects of a common superclass. It involves the ability of different classes to provide a common interface (methods with the same names) but with varying implementations. This enables more flexible and generic coding, as the same code can be used to operate on objects of different classes, as long as they adhere to the common interface.

[ ]:
```

```

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

ANS: In object-oriented programming, the **init**() function (constructor) is used to initialize the attributes of an object when it is created from a class. It's a special method that gets automatically called when an object is instantiated. This is where you can set up the initial state of the object by assigning values to its attributes.

[10]:
```
class Person:
    def __init__(self, name, age):
```

```
        self.name = name
        self.age = age

    def introduce(self):
        print(f"Hi, my name is {self.name} and I am {self.age} years old.")
```

[13]:
```
# Creating objects of the Person class
person1 = Person("Manish", 30)
person2 = Person("Bradley", 25)
```

[14]:
```
# Calling the introduce method on the objects
person1.introduce()
person2.introduce()
```

```
Hi, my name is Manish and I am 30 years old.
Hi, my name is Bradley and I am 25 years old.
```

[ ]:

Q4. Why self is used in OOPs?

ANS:

In Object-Oriented Programming (OOP), the self keyword is used to refer to the instance of the class within its methods. It allows you to access the attributes and methods of the instance from within those methods. The self parameter is the first parameter in most instance methods and serves as a reference to the object itself.

Here's why self is used and why it's important:

Instance Context: In OOP, you often define classes to create objects with specific attributes and behavior. The self parameter allows methods to know which particular instance of the class is currently being operated on. It provides a way for methods to access and modify the attributes of that instance.

Attribute Access: By using self, you can access instance attributes (variables) within the methods of the class. It helps you differentiate between the attributes of the current instance and any other attributes that might exist in the class.

Method Invocation: When a method is called on an instance of a class, the instance is automatically passed as the self argument to the method. This allows the method to operate on the specific instance, making it possible to modify its state.

[ ]:

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

ANS:

Inheritance is a fundamental concept in object-oriented programming (OOP) that allows you to create a new class (subclass or derived class) based on an existing class (base class or parent class). The subclass inherits attributes and methods from the base class, and it can also extend or override

them. Inheritance promotes code reuse and facilitates the creation of hierarchical relationships among classes.

There are different types of inheritance:

Single Inheritance: In single inheritance, a subclass inherits from a single base class.

Multiple Inheritance: In multiple inheritance, a subclass inherits from more than one base class.

Multilevel Inheritance: In multilevel inheritance, a subclass inherits from another subclass, forming a chain.

Hierarchical Inheritance: In hierarchical inheritance, multiple subclasses inherit from a single base class.

Multilevel Inheritance:

[32]:
```
#Single Inheritance
class animal:
    def speak(self):
        pass

class dog(animal):
    def speak(self):
        return "Woof!"

class cat(animal):
    def speak(self):
        return "Meow!"
```

[37]:
```
dog1=dog()
dog2=dog()
cat1=cat()
cat2=cat()
```

[41]:
```
dog.speak('tommy')
```

[41]:
```
'Woof!'
```

[42]:
```
cat.speak('kitty')
```

[42]:
```
'Meow!'
```

[ ]:
```

```

[43]:
```
#Multiple Inheritance
class Bird:
    def fly(self):
        return "I can fly!"

class Mammal:
```

```
    def feed_milk(self):
        return "I can feed milk!"

class Bat(Bird, Mammal):
    pass
```

[44]:
```
Bat1=Bat()
```

[46]:
```
Bat1.feed_milk()
```

[46]: 'I can feed milk!'

[48]:
```
Bat1.fly()
```

[48]: 'I can fly!'

[ ]:

[49]:
```
#Multilevel Inheritance
class Vehicle:
    def start_engine(self):
        return "Engine started."

class Car(Vehicle):
    def drive(self):
        return "Car is being driven."

class ElectricCar(Car):
    def charge(self):
        return "Car is charging."
```

[53]:
```
car = Car()
electric_car = ElectricCar()
```

[55]:
```
car.start_engine()
```

[55]: 'Engine started.'

[56]:
```
car.drive()
```

[56]: 'Car is being driven.'

[57]:
```
electric_car.charge()
```

[57]: 'Car is charging.'

[ ]:

[58]:
```
#Hierarchical Inheritance
class Shape:
    def area(self):
        pass

class Circle(Shape):
    def area(self, radius):
        return 3.14 * radius * radius

class Rectangle(Shape):
    def area(self, length, width):
        return length * width
```

[59]:
```
circle=Circle()
rectangle=Rectangle()
```

[61]:
```
circle.area(5)
```

[61]: 78.5

[62]:
```
rectangle.area(4,6)
```

[62]: 24

[ ]:

[ ]: