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

In object-oriented programming (OOP), a class is a blueprint or template for creating objects, while an object is an instance of a class. A class defines the common properties (attributes) and behaviors (methods) that objects of that class will possess. It encapsulates data and functions into a single entity, providing a way to organize and structure code.

Let's take an example to understand the concept of class and object. Suppose we have a class called "Car" that represents the general characteristics and behaviors of a car. The class would define attributes like "brand," "color," "fuel type," and methods like "start," "accelerate," and "stop."

class Car:

def \_\_init\_\_(self, brand, color, fuel\_type):

self.brand = brand

self.color = color

self.fuel\_type = fuel\_type

self.speed = 0

def start(self):

print("The car has started.")

def accelerate(self, speed\_increase):

self.speed += speed\_increase

print(f"The car is now moving at {self.speed} km/h.")

def stop(self):

self.speed = 0

print("The car has stopped.")

# Creating objects/instances of the Car class

car1 = Car("Toyota","Red", "Gasoline")

car2 = Car("BMW", "Blue", "Diesel")

# Accessing object attributes

print(car1.brand) # Output: Toyota

print(car2.color) # Output: Blue

# Calling object methods

car1.start() # Output: The car has started.

car2.accelerate(50) # Output: The car is now moving at 50 km/h.

car1.stop() # Output: The car has stoppe

the "Car" class serves as a blueprint. We create two objects,’ **car1’** and’ **car2’**, by calling the class as if it were a function. Each object has its own set of attributes (brand, color, fuel type) and can perform actions through its methods (start, accelerate, stop). By accessing the attributes and invoking the methods of the objects, we can interact with and manipulate their individual properties.

This is a simplified representation of the class and object concept in OOP. In real-world scenarios, classes often have more attributes and methods, and objects interact with each other to achieve complex functionalities. OOP provides a modular and reusable approach to programming by allowing us to define and manipulate objects that represent entities in our problem domain.

Q2. Name the four pillars of OOPs.

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

Encapsulation: Encapsulation is the process of bundling data and the methods (or functions) that operate on that data into a single unit called an object. It allows for the hiding of internal implementation details and provides a clear interface to interact with the object. Encapsulation helps in achieving data security and code modularity.

Inheritance: Inheritance enables the creation of new classes (derived classes) from existing classes (base classes). The derived classes inherit the properties (fields and methods) of the base class, allowing for code reuse and the implementation of a hierarchical structure. Inheritance promotes the concept of "is-a" relationship, where a derived class is considered to be a specialized version of the base class.

Polymorphism: Polymorphism refers to the ability of objects to take on multiple forms. It allows objects of different classes to be treated as objects of a common base class. Polymorphism can be achieved through method overriding and method overloading. Method overriding allows a subclass to provide a different implementation of a method already defined in its superclass, while method overloading involves having multiple methods with the same name but different parameters.

Abstraction: Abstraction focuses on representing the essential features of an object while hiding the unnecessary details. It provides a way to create abstract classes and interfaces that define the common characteristics and behaviors of a group of objects. Abstraction helps in managing complexity by breaking down a system into smaller, more manageable units, and it allows for the creation of reusable code components.

These four pillars collectively form the foundation of Object-Oriented Programming and are fundamental concepts in designing and implementing object-oriented systems.

Q3. Explain why the ‘\_\_init\_\_()’ function is used. Give a suitable example.

The ‘**\_\_init\_\_()’** function in Python is a special method that is automatically called when an object of a class is created. It is commonly known as the constructor method. The purpose of the‘**\_\_init\_\_()’** function is to initialize the attributes or properties of an object.

When a class is defined, it may have certain attributes that need to be set to specific values when an object is created. The ‘ **\_\_init\_\_()**  method allows us to define these initial values for the attributes.

Here's an example to illustrate the usage of the ‘**\_\_init\_\_()’** function:

class Car:

def \_\_init\_\_(self, brand, model, year):

self.brand = brand

self.model = model

self.year = year

def get\_details(self):

return f"{self.brand} {self.model} ({self.year})"

# Creating objects of the Car class

car1 = Car("Toyota", "Camry", 2020)

car2 = Car("Honda", "Civic", 2021)

# Accessing object attributes and methods

print(car1.get\_details()) # Output: Toyota Camry (2020)

print(car2.get\_details()) # Output: Honda Civic (2021)

a ‘**Car’** class with attributes like ‘**brand’**, ‘**model’**, and ‘**year’**. The ‘**\_\_init\_\_()’** method takes these attributes as parameters along with the special’ **self’** parameter, which refers to the instance of the class. Inside the ‘**\_\_init\_\_()’** method, we assign the parameter values to the corresponding object attributes using the dot notation (‘**self.attribute\_name = parameter’**).

When we create objects of the ‘**Car’** class (‘**car1’** and ‘**car2’**), the ‘**\_\_init\_\_()’** method is automatically invoked, and the provided values are used to initialize the object's attributes. Later, we can access these attributes using the dot notation (‘**object\_name.attribute\_name’**).

The ‘**\_\_init\_\_()’** function allows us to ensure that every object of a class starts with specific attribute values, providing a convenient way to set up object state during object creation.

Q4. Why self is used in OOPs?

In object-oriented programming (OOP), the concept of "self" is used to refer to the instance of a class within its own methods. It is a special parameter that allows an object to refer to itself and access its own attributes and methods.

The use of "self" (or any other name, such as "this" in some programming languages) is important in OOP for several reasons:

1. Instance-specific access: Each instance of a class has its own set of attributes and methods. By using "self", you can differentiate between the instance variables and methods of the current object and those of other objects of the same class. It allows you to access and manipulate the instance-specific data and behaviors.
2. Encapsulation: "self" is essential for encapsulation, which is one of the core principles of OOP. Encapsulation helps in organizing and controlling the access to an object's attributes and methods. By using "self", you can ensure that the object's internal state is properly managed and that the object itself is responsible for maintaining its own data integrity.
3. Method invocation: When you call a method on an object, you need a way to specify that the method should operate on the current object. "self" serves as a reference to the current object, allowing you to call its methods and access its attributes from within the class.
4. Inheritance and polymorphism: Inheritance is a fundamental concept in OOP, where a class can inherit attributes and methods from a parent class. The use of "self" ensures that the correct methods are invoked on the appropriate objects in the inheritance hierarchy. It also enables polymorphism, allowing different objects of related classes to be treated interchangeably.

Overall, "self" plays a crucial role in maintaining the integrity of objects, enabling encapsulation, facilitating method invocation, and supporting inheritance and polymorphism in object-oriented programming.

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

In object-oriented programming, inheritance is a mechanism that allows a class to inherit properties and behaviors from another class. The class that inherits is called the "subclass" or "derived class," and the class from which it inherits is called the "superclass" or "base class." This concept promotes code reuse, as subclasses can inherit and extend the functionality of their superclass.

There are several types of inheritance:

1. Single Inheritance: In single inheritance, a subclass inherits from a single superclass. It forms a hierarchy where subclasses inherit properties and methods from only one class. For example:

class Vehicle:

def \_\_init\_\_(self, brand):

self.brand = brand

def start(self):

print("Starting the vehicle.")

class Car(Vehicle):

def \_\_init\_\_(self, brand, model):

super().\_\_init\_\_(brand)

self.model = model

def drive(self):

print(f"Driving the {self.brand} {self.model}.")

my\_car = Car("Toyota", "Camry")

my\_car.start()

my\_car.drive()

the class **Car** inherits from the class **Vehicle**. The **Car** class extends the functionality of **Vehicle** by adding a **drive** method while still having access to the **start** method inherited from **Vehicle**.

1. Multilevel Inheritance: Multilevel inheritance occurs when a subclass inherits from a superclass, and another subclass inherits from this derived class, forming a chain. Each class in the hierarchy inherits properties and behaviors from the class above it. For example:

class Animal:

def \_\_init\_\_(self, name):

self.name = name

def speak(self):

print("Animal speaks.")

class Dog(Animal):

def \_\_init\_\_(self, name, breed):

super().\_\_init\_\_(name)

self.breed = breed

def bark(self):

print("Woof!")

class Bulldog(Dog):

def run(self):

print("Bulldog runs.")

my\_dog = Bulldog("Max", "Bulldog")

my\_dog.speak()

my\_dog.bark()

my\_dog.run()

In this example, the class **Animal** is the superclass of **Dog**, and **Dog** is the superclass of **Bulldog**. Each class inherits the properties and methods from its superclass.

1. Multiple Inheritance: Multiple inheritance allows a subclass to inherit from multiple superclasses. This means that the subclass can inherit and combine the properties and behaviors of multiple classes. For example:

class Father:

def speak(self):

print("Father speaks.")

class Mother:

def sing(self):

print("Mother sings.")

class Child(Father, Mother):

pass

my\_child = Child()

my\_child.speak()

my\_child.sing()