**Q1. What is the meaning of multiple inheritance?**

**Answer:-**

Multiple inheritance refers to the ability of a class to inherit attributes and behaviors from multiple parent classes. In Python, a class can inherit from multiple base classes, allowing it to acquire characteristics and functionality from all of them.

When a class inherits from multiple parent classes, it gains access to the attributes and methods defined in each parent class. This enables the subclass to combine and utilize the features of multiple classes, promoting code reuse and flexibility in the class hierarchy.

**class ParentClass1:**

**def method1(self):**

**print("Method 1 from Parent Class 1")**

**class ParentClass2:**

**def method2(self):**

**print("Method 2 from Parent Class 2")**

**class ChildClass(ParentClass1, ParentClass2):**

**def method3(self):**

**print("Method 3 in Child Class")**

**# Creating an instance of ChildClass**

**my\_instance = ChildClass()**

**# Accessing methods from both parent classes**

**my\_instance.method1() # Output: Method 1 from Parent Class 1**

**my\_instance.method2() # Output: Method 2 from Parent Class 2**

**my\_instance.method3() # Output: Method 3 in Child Class**

**Q2. What is the concept of delegation?**

**Answer:-**

The concept of delegation in object-oriented programming refers to the process of one object (delegator) assigning responsibility for a particular task or behavior to another object (delegate). Instead of implementing the behavior directly, the delegator object forwards or delegates the task to the delegate object, which is responsible for performing the actual implementation.

Delegation allows objects to collaborate and distribute responsibilities effectively. It promotes modularity, code reuse, and separation of concerns by encapsulating specific behaviors in separate objects.

class Delegator:

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

self.delegate = Delegate()

def perform\_task(self):

self.delegate.do\_task()

class Delegate:

def do\_task(self):

print("Delegate performs the task")

# Creating an instance of Delegator

my\_delegator = Delegator()

# Delegating the task to the delegate

my\_delegator.perform\_task()

**Q3. What is the concept of composition?**

**Answer:-**

The concept of composition in object-oriented programming refers to the practice of creating complex objects by combining or "composing" simpler objects. It involves creating a class that contains other objects as member variables, forming a "whole-part" relationship between the composite object and its components.

Composition allows for the creation of more complex and specialized objects by assembling smaller, reusable objects. It promotes code reuse, modular design, and separation of concerns. The composed objects can have their own behavior and can be used to implement specific features or functionalities within the composite object.

**class Engine:**

**def start(self):**

**print("Engine started")**

**def stop(self):**

**print("Engine stopped")**

**class Car:**

**def \_\_init\_\_(self):**

**self.engine = Engine() # Composition: Car has an Engine**

**def start\_car(self):**

**print("Starting the car")**

**self.engine.start()**

**def stop\_car(self):**

**print("Stopping the car")**

**self.engine.stop()**

**# Creating an instance of Car**

**my\_car = Car()**

**# Starting and stopping the car**

**my\_car.start\_car()**

**my\_car.stop\_car()**

**Q4. What are bound methods and how do we use them?**

**Answer:-**

Bound methods are a concept in Python that refer to methods bound to a specific instance of a class. When a method is accessed through an instance of a class, it is automatically bound to that instance, allowing the method to access the instance's attributes and perform actions specific to that instance.

Bound methods are created dynamically when an instance of a class calls a method defined in its class. They maintain a reference to the instance as the first argument (self by convention), allowing the method to operate on the instance's data.

**class MyClass:**

**def \_\_init\_\_(self, value):**

**self.value = value**

**def my\_method(self):**

**print("Value:", self.value)**

**# Creating an instance of MyClass**

**my\_object = MyClass(42)**

**# Accessing and calling a bound method**

**my\_object.my\_method() # Output: Value: 42**

**Q5. What is the purpose of pseudoprivate attributes?**

**Answer:-**

Pseudoprivate attributes in Python are created by prefixing a double underscore (\_\_) to an attribute name. Their purpose is to indicate that the attribute is intended for internal use within the class and discourage direct access or modification from outside the class. However, they are not true private attributes and can still be accessed using the modified name.

**class MyClass:**

**def \_\_init\_\_(self):**

**self.\_\_private\_attr = 42**

**def get\_private\_attr(self):**

**return self.\_\_private\_attr**

**# Creating an instance of MyClass**

**my\_object = MyClass()**

**# Accessing pseudoprivate attribute through a method**

**print(my\_object.get\_private\_attr()) # Output: 42**