Q1. What is the meaning of multiple inheritance?

Q2. What is the concept of delegation?

Q3. What is the concept of composition?

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

Q5. What is the purpose of pseudoprivate attributes?

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

**Multiple inheritance** is a feature in object-oriented programming where a class can inherit attributes and methods from more than one parent class. This allows a derived class to combine the behaviors and properties of multiple base classes.

In Python, multiple inheritance is supported, and it can be implemented by listing multiple parent classes in the class definition:

python

Copy code

class Parent1:

def method1(self):

print("Method from Parent1")

class Parent2:

def method2(self):

print("Method from Parent2")

class Child(Parent1, Parent2):

pass

obj = Child()

obj.method1() # Outputs: Method from Parent1

obj.method2() # Outputs: Method from Parent2

**Advantages**:

* Code reuse from multiple sources.
* Combining functionality from various base classes.

**Challenges**:

* Potential for ambiguity, particularly with the **diamond problem**, where a derived class inherits from two classes that have a common base class.

Python resolves the ambiguity using the Method Resolution Order (MRO), which determines the order in which base classes are searched when looking for a method.

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

**Delegation** is a design pattern where an object handles a request by passing it to a second "delegate" object. Instead of performing the task itself, the object delegates it to another object that can perform the task. This pattern allows for composition of objects and separation of concerns.

In Python, delegation can be implemented by creating an instance of another class within a class and forwarding method calls to that instance:

python

Copy code

class Delegate:

def method(self):

print("Method from Delegate")

class Delegator:

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

self.delegate = Delegate()

def method(self):

return self.delegate.method()

obj = Delegator()

obj.method() # Outputs: Method from Delegate

**Advantages**:

* Promotes code reuse.
* Reduces the need for multiple inheritance by composing behavior.

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

**Composition** is a design principle where one class is composed of one or more objects of other classes, meaning that it has objects of these classes as members. Composition represents a "has-a" relationship, whereas inheritance represents an "is-a" relationship.

For example, a Car class might be composed of Engine, Wheels, and Body classes:

python

Copy code

class Engine:

def start(self):

print("Engine started")

class Wheels:

def roll(self):

print("Wheels rolling")

class Car:

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

self.engine = Engine()

self.wheels = Wheels()

def drive(self):

self.engine.start()

self.wheels.roll()

car = Car()

car.drive()

In this example, Car is composed of Engine and Wheels, meaning a car "has an" engine and "has wheels."

**Advantages**:

* Encourages modular design.
* Easier to maintain and extend because components can be replaced or modified independently.

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

**Bound methods** are methods that are associated with an instance of a class. When you access a method from an instance, Python automatically passes the instance as the first argument (usually named self). This is what makes the method "bound" to that instance.

Example:

python

Copy code

class MyClass:

def method(self, value):

print(f"Called method with {value} on instance {self}")

obj = MyClass()

bound\_method = obj.method # This is a bound method

bound\_method(10) # Outputs: Called method with 10 on instance <MyClass object>

**Key Characteristics**:

* A bound method knows which object it is bound to (self).
* You can call the method directly on the instance or store it in a variable and call it later.

**Usage**:

* Bound methods are useful in callbacks or event-driven programming where you may need to store references to methods and call them later with the correct instance.

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

**Pseudoprivate attributes** are attributes in a class that are intended to be private but are not strictly enforced by the language. In Python, pseudoprivate attributes are created by prefixing the attribute name with double underscores (\_\_). This triggers name mangling, where Python internally changes the attribute name to include the class name, making it harder to accidentally access or override in subclasses.

Example:

python

Copy code

class MyClass:

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

self.\_\_private = "I am private"

def get\_private(self):

return self.\_\_private

obj = MyClass()

print(obj.get\_private()) # Outputs: I am private

# print(obj.\_\_private) # This would raise an AttributeError

print(obj.\_MyClass\_\_private) # Outputs: I am private (name mangling)

**Purpose**:

* To prevent accidental access or modification of attributes by external code or subclasses.
* To signal that an attribute is intended for internal use within the class.

**Note**: Pseudoprivate attributes are not truly private; they are only "obscured" by name mangling. This feature is meant to discourage but not prevent access to these attributes.