Q1. What is the meaning of multiple inheritance?

Sol:-

Multiple inheritance refers to the capability of a class to inherit attributes and methods from more than one parent class. In Python, a class can inherit from multiple base classes, allowing it to combine features and behaviors from different sources.

When a class inherits from multiple parent classes, it gains access to the attributes and methods defined in each of those parent classes. This means that instances of the derived class will inherit and be able to utilize the attributes and methods of all the parent classes.

class DerivedClass(BaseClass1, BaseClass2, ...):

# class body

Q2. What is the concept of delegation?

Sol:-

The concept of delegation in object-oriented programming refers to a design pattern where an object forwards a specific task or responsibility to another object to handle it. Instead of performing the task itself, the delegating object delegates the responsibility to a different object, known as the delegate or helper object.

Delegation allows objects to collaborate and divide responsibilities, promoting code reuse, modularity, and flexibility. It helps in achieving the Single Responsibility Principle (SRP) by separating concerns and ensuring that each object is responsible for a specific task.

Here's how delegation works in practice:

The delegating object holds a reference to the delegate object and invokes its methods to perform certain tasks.

When a method is called on the delegating object, it redirects the request to the corresponding method of the delegate object.

The delegate object is responsible for handling the delegated task and returns the result back to the delegating object if needed.

Q3. What is the concept of composition?

Sol:-

The concept of composition in object-oriented programming refers to the design principle where complex objects or classes are built by combining simpler objects or classes as components. Composition allows objects to have relationships with other objects, forming a "has-a" relationship, rather than an "is-a" relationship as seen in inheritance.

In composition, an object is composed of one or more other objects, which are usually encapsulated within the containing object. The containing object delegates certain responsibilities to its component objects, utilizing their functionality to perform specific tasks.

Key features of composition include:

Encapsulation: The component objects are encapsulated within the containing object, ensuring that their internal details are hidden and can only be accessed through the containing object's interface.

Reusability: Composition promotes code reuse by allowing objects to be composed of reusable components. Components can be shared among different containing objects, enhancing modularity and reducing code duplication.

Flexibility: As the components are separate objects, they can be added, removed, or replaced at runtime, providing flexibility in the behavior and structure of the containing object.

Modularity: Composition supports a modular design approach by breaking down complex functionality into smaller, manageable components. Each component has a well-defined responsibility, making the codebase more organized and easier to maintain.

class Engine:

def start(self):

print("Engine started.")

class Car:

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

self.engine = Engine()

def start(self):

print("Starting the car...")

self.engine.start()

# Creating a Car instance and starting it

my\_car = Car()

my\_car.start()

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

Sol:-

In Python, a bound method is a method that is associated with an instance of a class. When a method is accessed through an instance, it becomes a bound method. Bound methods are automatically passed the instance as the first argument (conventionally named self) when called.

To use a bound method, you simply invoke it as you would any other method, using the instance on which it is bound. The instance is automatically passed as the first argument, allowing the method to access and manipulate the instance's data.

class MyClass:

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

self.value = value

def print\_value(self):

print(self.value)

# Create an instance of MyClass

my\_object = MyClass(42)

# Call the bound method on the instance

my\_object.print\_value()

Q5. What is the purpose of pseudoprivate attributes?

Sol:-

Pseudoprivate attributes in Python are attributes that have names starting with double underscores (\_\_). The purpose of pseudoprivate attributes is to introduce name mangling, which is a mechanism to make attributes within a class more "private" and avoid accidental name clashes with subclasses or other classes.

When a class attribute is named with double underscores as a prefix (e.g., \_\_attribute), Python performs name mangling by adding a prefix to the attribute name to make it unique. The added prefix includes the class name, which is prefixed with a single underscore. This process ensures that the attribute is not easily accessible or overridden by subclasses.

The purpose of pseudoprivate attributes can be summarized as follows:

Encapsulation: Pseudoprivate attributes help enforce encapsulation by making attributes less accessible from outside the class. By prefixing attribute names with double underscores, it signals that the attribute is intended for internal use within the class.

Name collision avoidance: Pseudoprivate attributes prevent accidental name clashes that may occur when subclasses or other classes define attributes with the same name. By performing name mangling, each class has its unique name-mangled version of the attribute, reducing the chances of conflicts.

Subclass protection: Pseudoprivate attributes are also useful in preventing accidental attribute overrides by subclasses. Since the name-mangled attribute is different from the original attribute name, subclasses cannot inadvertently override or access the pseudoprivate attribute unless they explicitly use the mangled name.