Python Advanced Assignment-5

Q1. What is the meaning of multiple inheritance?

Ans.

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 the subclass to combine functionality from multiple sources, but it can also introduce complexity, especially when the same attribute or method is present in more than one parent class. In Python, the Method Resolution Order (MRO) helps resolve conflicts by determining the order in which base classes are searched when an attribute or method is accessed.

Example:

```
class A:
    def method(self):
        print("Method in A")


class B:
    def method(self):
        print("Method in B")


class C(A, B):
    pass


obj = C()
obj.method()  # Output: "Method in A" due to MRO
```

Q2. What is the concept of delegation?

Ans.

Delegation is a design pattern where an object handles a request by passing it to a second object (a delegate). This is useful when you want to extend or customize the behavior of an object without subclassing it. In Python, you can achieve delegation by storing an instance of another class in an attribute and forwarding method calls to it.

Example:

```
class Engine:
    def start(self):
        return "Engine started"

class Car:
    def __init__(self):
        self.engine = Engine()

    def start(self):
        return self.engine.start()  # Delegates the call to the Engine instance
```

In this example, the `Car` class delegates the `start` method to its `engine` attribute.

Q3. What is the concept of composition?

Ans.

Composition is a design principle in object-oriented programming where a class is composed of one or more objects from other classes, using them as building blocks. It represents a "has-a" relationship (e.g., a `Car` has an `Engine`). Unlike inheritance, which defines an "is-a" relationship, composition allows for more flexibility and modularity, as the composed objects can be replaced or modified independently.

Example:

```
class Engine:
    def start(self):
        return "Engine started"

class Car:
    def __init__(self, engine):
        self.engine = engine  # Car is composed of an Engine

    def start(self):
        return self.engine.start()
```

Here, `Car` is composed of an `Engine` object, illustrating composition.

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

Ans.

Bound methods are methods associated with an instance of a class. When a method is called on an instance, Python automatically binds the method to that instance, passing the instance (`self`) as the first argument. This allows methods to access and modify the instance's attributes.

Example:

```
class MyClass:
    def method(self):
        print("Method called on", self)


obj = MyClass()
obj.method()  # This is a bound method; it has access to 'obj' as 'self'
```

In this case, `obj.method` is a bound method because it is tied to the `obj` instance. Bound methods allow us to access the instance's attributes and methods directly.

Q5. What is the purpose of pseudoprivate attributes?

Ans.

Pseudoprivate attributes in Python are used to prevent name clashes in subclasses and to make attributes less accessible outside the class. These attributes are defined by prefixing the attribute name with `__` (double underscores). Python internally changes the name of the attribute to include the class name (name mangling), which helps avoid accidental overriding in subclasses.

Example:

```
class MyClass:
    def __init__(self):
        self.__private = "Private attribute"


    def get_private(self):
        return self.__private
```

```
obj = MyClass()
print(obj.get_private())  # Accesses the private attribute correctly
print(obj.__private)      # Raises an AttributeError due to name mangling
```

Pseudoprivate attributes are not fully private but are used to reduce the likelihood of attribute conflicts and unintentional access from outside the class.