Q1. What is the meaning of multiple inheritance?

Multiple inheritance can be useful in cases where a subclass needs to inherit properties and behaviors from multiple parent classes.

For example:

class automobile:

  def \_\_init\_\_(self, name):

    self.name=name

  def move(self):

    print(f"{self.name} is moving")

class bus(automobile):

  def fuel(self):

    print(f"{self.name} is diesel")

class car(automobile):

  def cc(self):

    print(f"{self.name} is better")

class machine(bus,car):

  def \_\_init\_\_(self,name):

    super().\_\_init\_\_(name)

  def perf(self):

    print(f"{self.name} is four wheelers")

output:

bat=machine("maruti")

bat.fuel()-----------------------🡪maruti is diesel

bat.cc()------------------------🡪maruti is better

bat.move()----------------------🡪maruti is moving

bat.perf()---------------------🡪maruti is four wheelers

Here ‘bus ‘ and ‘car’ are subclasses of automobile class having methods as fuel and cc respectively. The machne subclass caries the inheritance of the parent class ‘bus’ and ‘car’ with method as perf. When we create a instance of machine class it will inherit the values from bus and car classes.

Q2. What is the concept of delegation?

Delegation is a technique in which an object forwards requests to another object to perform a task. This is achieved by creating an instance of another class within the current class and calling methods on that instance to perform the required functionality. Delegation is a way to achieve code reuse without inheritance.

class mathfunc:

  def add(self,a,b):

    return a+b

class calculator:

  def \_\_init\_\_(self):

    self.math=mathfunc()

  def add(self,a,b):

    return self.math.add(a,b)

input:

cal=calculator()

cal.add(2,3)

output: 5

In this example, Calculator class delegates the add operation to an instance of mathfunc class that it has created. Calculator class has its own add method that calls math.add method to perform the actual addition. The advantage of this approach is that Calculator class can reuse mathfunc class's implementation of the add method without inheriting from it.

Q3. What is the concept of composition?

Here, by using the class name or by creating the object we can access the members of one class inside another class. It enables creating complex types by combining objects of different classes. It means that a class Composite can contain an object of another class Component.

class component:

  #component class constructor

  def \_\_init\_\_(self):

    print("component object created ")

  #composite class instance method

  def m1(self):

    print("component class m1 method exec ")

class composite:

  #constructor of composite class

  def \_\_init\_\_(self):

    #creating obj of component class

    self.obj1=component()

    print("composite class obj created")

  #composite class instance method

  def m2(self):

    print("composite class m2 method exec ")

    #calling m1 method of component class

    self.obj1.m1()

# creating object of composite class

obj2 = composite()

# calling m2() method of composite class

obj2.m2()

output:

component object created

composite class obj created

composite class m2 method exec

component class m1 method exec

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

In Python, a bound method is a method that is bound to an instance of a class. When you call a method on an instance of a class, the instance is automatically passed as the first argument to the method.

For example:

class MyClass:

  def \_\_init\_\_(self, x):

    self.x = x

  def add(self, y):

    return self.x + y

input:

obj = MyClass(10)

result = obj.add(5)

print(result)

output:

15

Q5. What is the purpose of pseudoprivate attributes?

The purpose of pseudoprivate attributes is to prevent accidental name clashes with other attributes in subclasses or in other modules. When we define a pseudoprivate attribute in a class, the interpreter renames the attribute to \_classname\_\_attribute, where classname is the name of the class that the attribute belongs to. This makes it less likely that another attribute with the same name will accidentally overwrite the pseudoprivate attribute.

For example:

class MyClass:

  def \_\_init\_\_(self,value):

    self.\_\_value = value

  def get\_value(self):

    return self.\_\_value

input/output:

obj = MyClass(10)

print(obj.get\_value()) # prints 10

print(obj.\_\_value) # AttributeError: 'MyClass' object has no attribute '\_\_value'

print(obj.\_MyClass\_\_value) # prints 10