Q1. What is the meaning of multiple inheritance?

Answer:

It is a process of inheriting the properties from more than one parent class by a child class (reusing the code of multiple parent class by the child class). This child class is derived from multiple parent or base classes and inherits all their features.

|  |
| --- |
| class Parent\_A:  pass  class Parent\_B:  pass  class Child(Parent\_A, Parent\_B):  pass |

Q2. What is the concept of delegation?

Answer:

Delegation is a design pattern in which an object, called the delegate, is responsible for performing certain tasks on behalf of another object, called the delegator. It is an alternative to inheritance. It wraps the object of main class into a smaller object with limited access.

|  |
| --- |
| class Microwave:  def \_\_init\_\_(self):  pass  def heat\_up\_food(self):  print("Food is being microwaved")  class Dishwasher:  def \_\_init\_\_(self):  pass  def wash\_dishes(self):  print("Dishwasher starting")  # create wrapper methods in the Kitchen class  class Kitchen:  def \_\_init\_\_(self):  self.microwave = Microwave()  self.dishwasher = Dishwasher()  def heat\_up\_food(self):  self.microwave.heat\_up\_food()  def wash\_dishes(self):  self.dishwasher.wash\_dishes()    kitchen = Kitchen()  kitchen.heat\_up\_food()  kitchen.wash\_dishes()  # when we call heat\_up\_food on the Kitchen class, we are actually delegating it to the Microwave class |

Q3. What is the concept of composition?

Answer:

Composition is a process where one of the classes is composed of one or more instance of other classes. Irrespective of inheritence in this approach all the parent class members are not inherited into child class, but only required methods from a class are used by using class instances. Here, by using the class name or by creating the object we can access the members of one class inside another class. In other words a class Composite can contain an object of another class Component. Perhaps, composition models a "Has-A" relationship between "composite" and "component".

|  |
| --- |
| class Salary:  def \_\_init\_\_(self, pay, bonus):  self.pay = pay  self.bonus = bonus    def ctc(self):  return (self.pay\*12) + self.bonus    class Employee:  def \_\_init\_\_(self, name, age, pay, bonus):  self.name = name  self.age = age  self.tot = Salary(pay, bonus)    def total\_salary(self):  return self.tot.ctc()    emp = Employee('xyz', 35, 65000, 100000)  print(emp.total\_salary()) |

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

Answer:

A bound method is the method that has an object associated with it where as unbound method does not have any object associated with it. The bound methods are dependent on the instance of the classes which are using to access attributes of a function of the class.

|  |
| --- |
| class DisplayNumber:  def show(self, num):  print(num)    func1 = DisplayNumber.show  print(func1)  dn = DisplayNumber()  func2 = dn.show  print(func2)  # output  <function DisplayNumber.show at 0x0000022D96B3CC20>  <bound method DisplayNumber.show of <\_\_main\_\_.DisplayNumber object at 0x0000022D9679DED0>> |

Q5. What is the purpose of pseudoprivate attributes?

In Python, pseudoprivate attributes are useful in greater frameworks or tools, it is used to avoid introducing new method names that might accidentally hide definitions elsewhere in the class tree and to reduce the chance of internal methods being replaced by names defined lower in the tree. The double underscore prefix ensures that the method won't interfere with other names in the tree when a method is intended for use only within a class that may be mixed into other classes, especially in multiple-inheritance scenarios. Pseudoprivate attributes names also prevent subclasses from accidentally redefining the internal method's names.

class Test:

def \_\_init\_\_(self, a, b, c, d):

self.\_\_a = a # \_\_ means pseudo-private variable

self.b = b

self.c = c

self.d = d

def custom(self, val):

return val - self.\_\_a

t = Test(1, 2, 3, 4)

print(t.\_Test\_\_a) # this is the way to access pseudo-private attribute

print(t.b)

print(t.custom(5))