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

**Ans:**

Inheritance is nothing but reusing the code of Parent class by the child class. Similary when a child class inherits its properties from multiple Parent classes this scenario is called **Multiple Inheritence**

class Parent\_one:  
 pass  
class Parent\_two:  
 pass  
class child(Parent\_one,Parent\_two):  
 pass

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

**Ans:**

Delegation provides a proxy object for any class thay you want on top of the main class. its like a wrapper to your class so that you can access limited resources of the main class.

it Wraps the object of main class into a smaller object with limited access

Simply Delegation means that you can include a instance of another class as an instance variable, and forward messages to the instance.

class Myclass:  
 def sayHi(self):  
 print('Hey iam back')  
 def whoAmI(self):  
 print('Iam the main class')  
class NewClass:  
 def \_\_init\_\_(self,obj):  
 self.main = obj  
 def welcome(self):  
 self.main.sayHi()  
  
m = Myclass()   
n = NewClass(m)  
m.sayHi()  
n.main.sayHi()  
n.welcome()  
n.main.whoAmI()

Hey iam back  
Hey iam back  
Hey iam back  
Iam the main class

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

**Ans:**

In the concept of Composition, a class refers to one or more other classes by using instances of those classes as a instance variable. 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.

class Salary:  
 def \_\_init\_\_(self,pay):  
 self.pay = pay  
 def get\_total(self):  
 return self.pay\*12  
   
class Employee:  
 def \_\_init\_\_(self,pay,bonus):  
 self.pay = pay  
 self.bonus = bonus  
 self.obj\_salary = Salary(self.pay)  
 def annual\_salary(self):  
 return f'Total Salary : {str(self.obj\_salary.get\_total())}'  
  
obj\_emp = Employee(800,500)  
print(obj\_emp.annual\_salary())

Total Salary : 9600

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

**Ans:**

If a function is an attribute of class and it is accessed via the instances, they are called **bound methods**. A bound method is one that has **self** as its first argument. Since these are dependent on the instance of classes, these are also known as **instance methods**.

class Test:  
 def method\_one(self): # bound method  
 print("Called method\_one")  
 @classmethod  
 def method\_two(cls): # unbound method  
 print("Called method\_two")  
 @staticmethod  
 def method\_three(): # static method  
 print("Called method\_three")  
  
test = Test()  
test.method\_one() # accessing through instance object  
test.method\_two() # accessing through instance object  
Test.method\_two() # accessing directly through class  
Test.method\_three() # accessing directly through class

Called method\_one  
Called method\_two  
Called method\_two  
Called method\_three

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

**Ans:**

Pseudoprivate attributes are also useful in larger frameworks or tools, both 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. If a method is intended for use only within a class that may be mixed into other classes, the double underscore prefix ensures that the method won't interfere with other names in the tree, especially in multiple-inheritance scenarios

Pseudoprivate names also prevent subclasses from accidentally redefining the internal method's names,

class Super:  
 def method(self): # A real application method  
 pass  
class Tool:  
 def \_method(self): # becomes \_Tool\_method  
 pass  
 def other(self): # uses internal method  
 self.\_method()  
class Subl(Tool,Super):  
 def actions(self):  
 self.method()  
class Sub2(Tool):  
 def \_\_init\_\_(self):  
 self.method = 99