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

Ans : When you inherit a child class from more than one base classes, that situation is known as Multiple Inheritance. It, however, exhibits the same behavior as does the single inheritance.

The syntax for Multiple Inheritance is also similar to the single inheritance. By the way, in Multiple Inheritance, the child class claims the properties and methods of all the parent classes.

In Python, the projects and packages follow a principle called DRY, i.e., don’t-repeat-yourself. And Class inheritance is an excellent way to design a class reusing the features of another one and remain DRY.

# Parent class 1

class TeamMember(object):

def \_\_init\_\_(self, name, uid):

self.name = name

self.uid = uid

# Parent class 2

class Worker(object):

def \_\_init\_\_(self, pay, jobtitle):

self.pay = pay

self.jobtitle = jobtitle

# Deriving a child class from the two parent classes

class TeamLeader(TeamMember, Worker):

def \_\_init\_\_(self, name, uid, pay, jobtitle, exp):

self.exp = exp

TeamMember.\_\_init\_\_(self, name, uid)

Worker.\_\_init\_\_(self, pay, jobtitle)

print("Name: {}, Pay: {}, Exp: {}".format(self.name, self.pay, self.exp))

TL = TeamLeader('Jake', 10001, 250000, 'Scrum Master', 5)

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

Ans : Delegation is an object oriented technique (also called a design pattern). Let's say you have an object x and want to change the behaviour of just one of its methods. You can create a new class that provides a new implementation of the method you're interested in changing and delegates all other methods to the corresponding method of x.

Python programmers can easily implement delegation. For example, the following class implements a class that behaves like a file but converts all written data to uppercase:

class UpperOut:

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

self.\_outfile = outfile

def write(self, s):

self.\_outfile.write(s.upper())

def \_\_getattr\_\_(self, name):

return getattr(self.\_outfile, name)

Here the UpperOut class redefines the write() method to convert the argument string to uppercase before calling the underlying self.\_\_outfile.write() method. All other methods are delegated to the underlying self.\_\_outfile object. The delegation is accomplished via the \_\_getattr\_\_ method; consult the language reference for more information about controlling attribute access.

Note that for more general cases delegation can get trickier. When attributes must be set as well as retrieved, the class must define a \_\_setattr\_\_() method too, and it must do so carefully. The basic implementation of \_\_setattr\_\_() is roughly equivalent to the following:

class X:

...

def \_\_setattr\_\_(self, name, value):

self.\_\_dict\_\_[name] = value

...

Most \_\_setattr\_\_() implementations must modify self.\_\_dict\_\_ to store local state for self without causing an infinite recursion.

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

Ans :

Composition is a concept that models a has a relationship. It enables creating complex types by combining objects of other types. This means that a class Composite can contain an object of another class Component . This relationship means that a Composite has a Component .

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

Ans : A bound method is the one which is dependent on the instance of the class as the first argument. It passes the instance as the first argument which is used to access the variables and functions. In Python 3 and newer versions of python, all functions in the class are by default bound methods.

**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.