1. Inheritance is nothing but reusing the code of Parent class by the child class. Similarly 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**

2. Delegation provides a proxy object for any class that you want on top of the main class. It’s 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 an instance of another class as an instance variable, and forward messages to the instance.

3. 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 inheritance 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.

4. 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*

Output: Called method\_one

Called method\_two

Called method\_two

Called method\_three

5. 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