Q1. What is the meaning of multiple inheritance?

# Python program to demonstrate

# multiple inheritance

# Base class1

class Mother:

mothername = ""

def mother(self):

print(self.mothername)

# Base class2

class Father:

fathername = ""

def father(self):

print(self.fathername)

# Derived class

class Son(Mother, Father):

def parents(self):

print("Father :", self.fathername)

print("Mother :", self.mothername)

# Driver's code

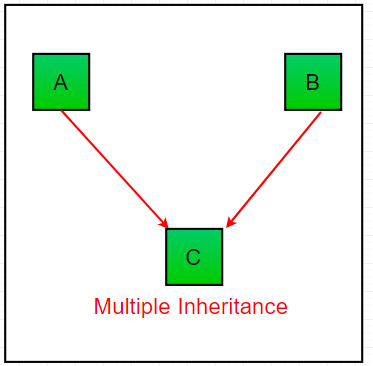
s1 = Son()

s1.fathername = "RAM"

s1.mothername = "SITA"

s1.parents()\

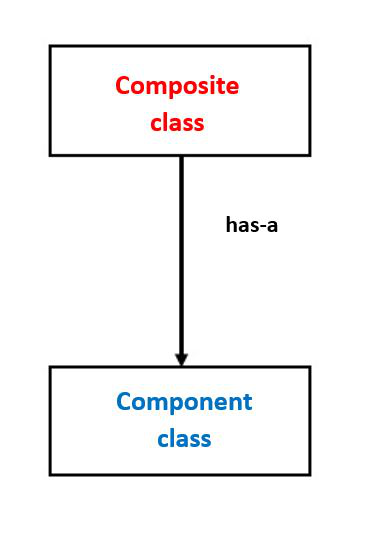
**Multiple Inheritance:** When a class can be derived from more than one base class this type of inheritance is called multiple inheritance. In multiple inheritance, all the features of the base classes are inherited into the derived class. 



Q2. What is the concept of delegation?

Q3. What is the concept of composition?

It is one of the fundamental concepts of Object-OrientedProgramming. In this concept, we will describe a class that references to one or more objects of other classes as an Instance variable. 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. This type of relationship is known as **Has-A Relation**.



composition – diagrammatic representation

class A :

# variables of class A

# methods of class A

...

...

class B :

**# by using "obj" we can access member's of class A.**

**obj = A()**

# variables of class B

# methods of class B

Composition vs Inheritance

It’s big confusing among most of the people that both the concepts are pointing to Code Reusability then what is the difference b/w Inheritance and Composition and when to use Inheritance and when to use Composition?

Inheritance is used where a class wants to derive the nature of parent class and then modify or extend the functionality of it. **Inheritance will extend the functionality with extra features allows overriding of methods, but in the case of Composition, we can only use that class we can not modify or extend the functionality of it. It will not provide extra features. Thus, when one needs to use the class as it without any modification, the composition** is recommended and when one needs to change the behavior of the method in another class, then inheritance is recommended.

class Component:

# composite class constructor

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

print('Component class object created...')

# composite class instance method

def m1(self):

print('Component class m1() method executed...')

class Composite:

# composite class constructor

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

# creating object of component class

self.obj1 = Component()

print('Composite class object also created...')

# composite class instance method

def m2(self):

print('Composite class m2() method executed...')

# calling m1() method of component class

self.obj1.m1()

# creating object of composite class

obj2 = Composite()

# calling m2() method of composite class

obj2.m2()

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

Bound methods

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.

Need for these bound methods

The methods inside the classes would take at least one argument. To make them zero-argument methods, ‘decorators‘ has to be used. Different instances of a class have different values associated with them.

For example, if there is a class “Fruits”, and instances like apple, orange, mango are possible. Each instance may have different size, color, taste, and nutrients in it. Thus to alter any value for a specific instance, the method must have ‘self’ as an argument that allows it to alter only its property.

Example:

class sample(object):

# Static variable for object number

objectNo = 0

def \_\_init\_\_(self, name1):

# variable to hold name

self.name = name1

# Increment static variable for each object

sample.objectNo = sample.objectNo + 1

# each object's unique number that can be

# considered as ID

self.objNumber = sample.objectNo

def myFunc(self):

print("My name is ", self.name,

"from object ", self.objNumber)

def alterIt(self, newName):

self.name = newName

def myFunc2():

print("I am not a bound method !!!")

# creating first instance of class sample

samp1 = sample("A")

samp1.myFunc()

# unhide the line below to see the error

# samp1.myFunc2() #----------> error line

# creating second instance of class sample

samp2 = sample("B")

samp2.myFunc()

samp2.alterIt("C")

samp2.myFunc()

samp1.myFunc()

Output:

My name is A from object 1

My name is B from object 2

My name is C from object 2

My name is A from object 1

Q5. What is the purpose of pseudoprivate attributes?

fact, there are no real private attributes and methods in Python, even the above\_\_weight with \_\_secret() They can also be accessed through special methods.

Only need to use this format when visitingObject name.\_Class name\_\_Private method name As shown in the last three lines of the following code:

class Women:

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

self.name = "Cherry"

self.\_\_weight = 100

def public(self):

print(self.\_\_weight)

def \_\_secret(self):

print(self.\_\_weight)

cherry = Women()

print(cherry.\_Women\_\_weight) '''Use a special format to access private attributes'''

cherry.\_Women\_\_secret() '''Use a special format to access private methods'''