Q1. What is the meaning of multiple inheritance?

**When a class is derived from more than one base class** it is called multiple Inheritance. The derived class inherits all the features of the base case. Syntax: Class Base1: Body of the class Class Base2: Body of the class Class Derived(Base1, Base2): Body of the class.

**Syntax:**

Class Base1:

Body of the class

Class Base2:

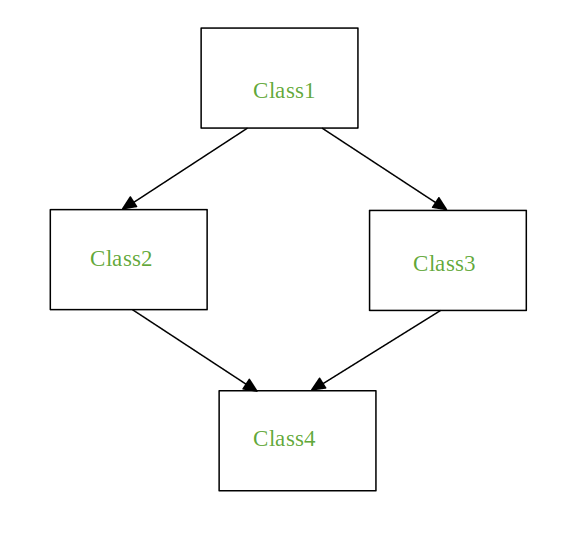
Body of the class

Class Derived(Base1, Base2):

Body of the class

In the coming section, we will see the problem faced during multiple inheritance and how to tackle it with the help of examples.

**The Diamond Problem**



It refers to an ambiguity that arises when two classes Class2 and Class3 inherit from a superclass Class1 and class Class4 inherits from both Class2 and Class3. If there is a method **“m”** which is an overridden method in one of Class2 and Class3 or both then the ambiguity arises which of the method “m” Class4 should inherit.

# Python Program to depict multiple inheritance

# when method is overridden in both classes

class Class1:

    def m(self):

        print("In Class1")

class Class2(Class1):

    def m(self):

        print("In Class2")

class Class3(Class1):

    def m(self):

        print("In Class3")

class Class4(Class2, Class3):

    pass

obj = Class4()

obj.m()

**Output:**

In Class2

**Note:** When you call obj.m() (m on the instance of Class4) the output is In Class2. If Class4 is declared as Class4(Class3, Class2) then the output of obj.m() will be In Class3.

Q2. What is the concept of delegation?

Delegation is **the assignment of authority to another person (normally from a manager to a subordinate) to carry out specific activities**. It is the process of distributing and entrusting work to another person, and therefore one of the core concepts of management leadership.

Let’s say we have a Dog class that is a subclass (and thus inherits the functionality of) an Animal class. If Animal has a method called get\_number\_of\_legs, any instantiation of the Dog class can call the get\_number\_of\_legs method. In Python, an implementation might look like this:

class Animal:

def \_\_init\_\_(self, name, num\_of\_legs):

self.name = name

self.num\_of\_legs = num\_of\_legs

def get\_number\_of\_legs(self):

print(f"I have {self.num\_of\_legs} legs")

class Dog(Animal):

def \_\_init\_\_(self, name, num\_of\_legs):

super().\_\_init\_\_(name, num\_of\_legs)

dog = Dog('Fido', 4)

dog.get\_number\_of\_legs()

# Outputs "I have 4 legs"

It would be technically incorrect to say that Dog delegates get\_number\_of\_legs to Animal because the Dog class actually has that method since it inherits the Animal class. This is what the Wikipedia definition is talking about when it refers to “code reuse.” This is what delegation will duplicate when we use composition.

**Kitchens are Compositions**

Let’s do a pseudocode example to understand delegation before we dig into a Python implementation. For this example, let’s think of a class called Kitchen. In real life, kitchens–as rooms–do not have any functionality, but we think of a kitchen’s functionality as a composition of the appliances it has. If my kitchen has a microwave, I can heat things up in my kitchen; likewise, with a dishwasher, I can wash dishes in my kitchen.

Now let’s think about this from an object oriented design position. I want to abstract away the appliance implementations and just think of heating up food and washing dishes as kitchen functions. In other words, I want to write my code like this:

# Pseudocode

kitchen = new Kitchen();

kitchen.heat\_up\_food();

# Food is being microwaved

kitchen.wash\_dishes();

# Dishwasher starting

In order to do this, in my Kitchen class definition, I probably have Microwave and Dishwasher classes as attributes of the Kitchen class, allowing the Kitchen class access to their methods. This is essentially what composition is. Now, let’s start really implementing this in Python so we can understand the rest of that Wikipedia definition.

**Pythonic Kitchen**

Ok, let’s talk about how this kitchen would look in Python code. I was thinking something like this:

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")

class Kitchen:

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

self.microwave = Microwave()

self.dishwasher = Dishwasher()

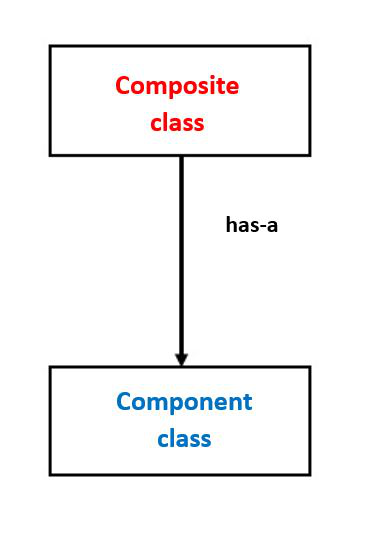
Q3. What is the concept of composition?

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.

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.

## ****What is Composition (Has-A Relation)?****

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*

 In the above figure Classes are represented as boxes with the class name **Composite**and **Component**representing Has-A relation between both of them.

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

...

...

|  |
| --- |
| 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() |

**Output**

Component class object created...

Composite class object also created...

Composite class m2() method executed...

Component class m1() method executed...

**Explanation:**

* In the above example, we created two classes **Composite**and **Component**to show the **Has-A Relation** among them.
* In the **Component class**, we have one constructor and an instance method **m1()**.
* Similarly, in **Composite class**, we have one constructor in which we created an object of **Component Class.**Whenever we create an object of **Composite Class**, the object ofthe **Component class**isautomatically created.
* Now in **m2()** method of **Composite class**we are calling **m1()** method of **Component Class**using instance variable **obj1** in which reference of **Component Class**is stored.
* Now, whenever we call **m2()** method of **Composite Class,**automatically **m1()** method of **Component Class**will be called.

## ****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**](https://www.geeksforgeeks.org/method-overriding-in-python/), 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.

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

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.

## 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](https://www.geeksforgeeks.org/self-in-python-class/)‘ 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](https://www.geeksforgeeks.org/decorators-in-python/)‘ 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

In the above example two instances namely samp1 and samp2 are created. Note that when the function alterIt() is applied to the second instance, only that particular instance’s value is changed. The line **samp1.myFunc()** will be expanded as **sample.myFunc(samp1)**. For this method no explicit argument is required to be passed. The instance samp1 will be passed as argument to the myFunc(). The line **samp1.myFunc2()** will generate the error :

Traceback (most recent call last):

File "/home/4f130d34a1a72402e0d26bab554c2cf6.py", line 26, in

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

TypeError: myFunc2() takes 0 positional arguments but 1 was given

It means that this method is **unbound**. It does not accept any instance as an argument. These functions are unbound functions.

Q5. What is the purpose of pseudoprivate attributes?

This is sometimes misleadingly called private attributes really, it's just a way to localize a name to the class that created it, and does not prevent access by code outside the class. That is, this feature is mostly intended **to avoid namespace collisions in instances, not to restrict access to names in general**.

This is sometimes misleadingly called private attributes really, it's just a way to localize a name to the class that created it, and does not prevent access by code outside the class. That is, this feature is mostly intended **to avoid namespace collisions in instances, not to restrict access to names in general**.

In many object-oriented languages, certain attributes can be declared as private, **making it impossible for users of a class to directly view or modify their values**. The designer of the class then provides methods to control the ways in which these attributes can be manipulated.

Private Keyword. The private members of a class are only accessible within the class. In Python, **a private member can be defined by using a prefix \_\_ (double underscore)**. So, in the private modifier's case, we cannot access the attribute.