1. What is the concept of an abstract superclass?

2. What happens when a class statement&#39;s top level contains a basic assignment statement?

3. Why does a class need to manually call a superclass&#39;s \_\_init\_\_ method?

4. How can you augment, instead of completely replacing, an inherited method?

5. How is the local scope of a class different from that of a function?

### **1. What is the concept of an abstract superclass?**

An abstract superclass is a class that is designed to be a base class for other classes and is not intended to be instantiated on its own. It typically contains one or more abstract methods, which are methods declared but not implemented in the abstract class. Subclasses are expected to provide concrete implementations of these abstract methods.

In Python, abstract superclasses can be created using the abc module. The ABC class from this module serves as a base for defining abstract superclasses, and the @abstractmethod decorator is used to define abstract methods:

python

Copy code

from abc import ABC, abstractmethod

class MyAbstractClass(ABC):

@abstractmethod

def my\_abstract\_method(self):

pass

### **2. What happens when a class statement's top level contains a basic assignment statement?**

When a class statement's top level contains a basic assignment statement, the assigned value becomes a class attribute. This means the attribute is shared among all instances of the class and can be accessed via the class itself or its instances. For example:

python

Copy code

class MyClass:

class\_variable = 10 # This is a class attribute

instance1 = MyClass()

instance2 = MyClass()

print(instance1.class\_variable) # Outputs: 10

print(instance2.class\_variable) # Outputs: 10

In this example, class\_variable is a class attribute that both instance1 and instance2 share.

### **3. Why does a class need to manually call a superclass's \_\_init\_\_ method?**

A class needs to manually call a superclass's \_\_init\_\_ method to ensure that the initialization logic defined in the superclass is executed when an instance of the subclass is created. This is important when the superclass performs essential setup tasks, such as initializing attributes or setting up resources, that the subclass relies on. If the superclass's \_\_init\_\_ method is not called, these tasks will not be performed, potentially leading to errors.

In Python, this is done using the super() function:

python

Copy code

class SuperClass:

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

self.value = value

class SubClass(SuperClass):

def \_\_init\_\_(self, value, extra):

super().\_\_init\_\_(value) # Calls the superclass's \_\_init\_\_ method

self.extra = extra

### **4. How can you augment, instead of completely replacing, an inherited method?**

You can augment an inherited method by overriding the method in the subclass and calling the superclass's version of the method within the overridden method. This allows you to add additional functionality while still retaining the behavior of the inherited method.

Here's an example:

python

Copy code

class SuperClass:

def method(self):

print("SuperClass method")

class SubClass(SuperClass):

def method(self):

super().method() # Call the superclass's method

print("SubClass method") # Add additional functionality

instance = SubClass()

instance.method()

In this example, the SubClass method augments the SuperClass method by calling it first and then adding its own behavior.

### **5. How is the local scope of a class different from that of a function?**

The local scope of a class is different from that of a function in several key ways:

1. **Access to Class Attributes**: In a class, you can directly access class attributes and methods within the class body. However, in a function, you typically only have access to local variables or those passed as arguments, unless you explicitly access global or nonlocal variables.
2. **Namespace**: The class scope acts as a namespace where class attributes and methods are defined and can be referenced. Once the class body is fully executed, this namespace is turned into a class object. In contrast, a function’s local scope only exists while the function is executing and is discarded afterward.
3. **Definition Context**: Variables and functions defined within a class are scoped to that class, while variables defined within a function are local to that function. A class can contain multiple methods that can share access to class-level attributes, while a function’s local variables are only accessible within that specific function call.
4. **Binding and Lookup**: In a class, when you reference a name, Python first looks in the class's namespace and then in the global and built-in namespaces. In a function, Python first looks in the local scope of the function, then in the enclosing scopes (if any), then in the global scope, and finally in the built-in scope.

Here’s a simple comparison:

python

Copy code

class MyClass:

class\_var = 10 # Class-level scope

def method(self):

local\_var = 20 # Function local scope

print(self.class\_var) # Access class-level scope

def my\_function():

local\_var = 30 # Function local scope

print(local\_var)