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

In Python, an abstract superclass is a class that cannot be instantiated but can be subclassed. The purpose of an abstract superclass is to provide a common interface for its subclasses and to ensure that they implement certain methods.

An abstract superclass is defined using the abc module (Abstract Base Class), which is part of the Python Standard Library. This module provides the ABC class, which is used to define an abstract superclass.

To define an abstract superclass, you need to derive a class from ABC and mark abstract methods using the @abstractmethod decorator. For example:

from abc import ABC, abstractmethod

class Shape(ABC):

@abstractmethod

def area(self):

pass

@abstractmethod

def perimeter(self):

pass

In this example, Shape is an abstract superclass that defines two abstract methods, area and perimeter. Any class that inherits from Shape must implement these methods, otherwise it will also be considered an abstract class and cannot be instantiated.

Here's an example of how you could define a concrete subclass of Shape:

class Circle(Shape):

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

self.radius = radius

def area(self):

return 3.14 \* self.radius \* self.radius

def perimeter(self):

return 2 \* 3.14 \* self.radius

In this example, Circle is a concrete subclass of Shape that implements both area and perimeter methods, so it can be instantiated.

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

When a class statement contains a basic assignment statement at its top level, the assignment creates a class-level variable, also known as a class attribute. This variable is shared among all instances of the class and can be accessed using the name of the class, for example ClassName.variable.

Here's an example to illustrate this:

class MyClass:

x = 10

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

self.value = value

# Access class variable

print(MyClass.x) # Output: 10

# Create an instance of MyClass

obj = MyClass(20)

# Access instance variable

print(obj.value) # Output: 20

In this example, the class MyClass has a class-level variable x with a value of 10. You can access this variable using the class name MyClass.x.

You can also modify the value of a class variable using the class name, for example MyClass.x = 20. This change will be reflected in all instances of the class:

MyClass.x = 20

print(MyClass.x) # Output: 20

print(obj.x) # Output: 20

It's important to note that while class-level variables are shared among all instances of a class, each instance has its own instance-level variables, which can have different values. In this example, value is an instance-level variable.

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

In Python, a class needs to manually call a superclass's \_\_init\_\_ method because inheritance in Python is based on the "implicit method resolution order" (MRO), which is the order in which a method is called from its subclasses to its superclasses. By default, the MRO does not automatically call the \_\_init\_\_ method of the superclass, so it needs to be done manually.

Here's an example to illustrate this:

class Shape:

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

self.sides = sides

def info(self):

print("I am a shape with", self.sides, "sides.")

class Triangle(Shape):

def \_\_init\_\_(self, sides, height):

Shape.\_\_init\_\_(self, sides)

self.height = height

def info(self):

Shape.info(self)

print("I am a triangle with height", self.height, ".")

tri = Triangle(3, 5)

tri.info()

In this example, the Triangle class inherits from the Shape class. The Triangle class needs to call the \_\_init\_\_ method of the Shape class explicitly in its own \_\_init\_\_ method using Shape.\_\_init\_\_(self, sides) in order to initialize the sides attribute. This allows the Triangle class to extend the functionality of the Shape class by adding its own attributes and methods.

Without calling the \_\_init\_\_ method of the Shape class, the sides attribute would not be initialized and would raise an AttributeError when trying to access it.

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

To augment an inherited method instead of completely replacing it, you can use the super() function to call the implementation from the superclass. This allows you to add additional functionality to the method, while still using the implementation from the superclass.

Here's an example to illustrate this:

class Shape:

def info(self):

print("I am a shape.")

class Triangle(Shape):

def info(self):

super().info()

print("I am a triangle.")

tri = Triangle()

tri.info()

In this example, the Triangle class inherits from the Shape class and overrides the info method. The Triangle class uses super().info() to call the implementation of the info method from the Shape class. The Triangle class then adds its own additional functionality by printing "I am a triangle.".

The output of this code would be:

I am a shape.

I am a triangle.

This demonstrates how you can augment an inherited method by calling the superclass's implementation and adding your own additional functionality to it.

**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 Python in several ways:

Persistence: The local scope of a class is persistent and lasts as long as the class itself exists. The local scope of a function, on the other hand, is only active while the function is executing and disappears once the function returns.

Accessibility: Variables defined in the local scope of a class are accessible to all methods of the class, while variables defined in the local scope of a function are only accessible within the function.

Naming collisions: Variables defined in the local scope of a class are only accessible within the class and are not accessible from outside the class. Variables defined in the local scope of a function, on the other hand, may cause naming collisions with variables defined in the global scope.

self parameter: The local scope of a class is always associated with a specific instance of the class, and the instance is passed to the class's methods as the first parameter, typically named self. In a function, the local scope is not associated with any specific object, and there is no equivalent to the self parameter.

Here's an example to illustrate these differences:

class MyClass:

x = 10

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

self.value = value

def my\_method(self):

y = 20

print("x:", self.x)

print("value:", self.value)

print("y:", y)

def my\_function():

x = 30

y = 40

print("x:", x)

print("y:", y)

obj = MyClass(50)

obj.my\_method()

my\_function()

In this example, MyClass has a class-level variable x and an instance-level variable value, both of which are accessible within the methods of the class. The local variable y defined in the my\_method method is only accessible within the method.

The my\_function has its own local variables x and y, which are only accessible within the function.

The output of this code would be:

x: 10

value: 50

y: 20

x: 30

y: 40

This demonstrates how the local scope of a class and a function are different in terms of persistence, accessibility, naming collisions, and the presence of the self parameter.