1. What is the concept of an abstract superclass?

* The concept of an abstract superclass is related to object-oriented programming and the concept of inheritance. An abstract superclass is a class that is designed to be inherited by other classes, but it cannot be instantiated on its own. It serves as a blueprint or template for its subclasses, defining common attributes and methods that subclasses should implement.
* An abstract superclass often contains one or more abstract methods, which are methods that are declared but have no implementation in the superclass. The responsibility of implementing these abstract methods lies with the subclasses. By defining abstract methods, the abstract superclass enforces a contract that any subclass must adhere to by providing concrete implementations for these methods.
* The purpose of an abstract superclass is to provide a common interface or behavior that subclasses should follow, while allowing each subclass to provide its own specific implementation details. It promotes code reusability, encapsulation, and allows for polymorphic behavior.
* In some programming languages like Python, there are dedicated mechanisms or modules to define abstract classes and abstract methods, such as the abc module. By defining an abstract superclass using these mechanisms, you can enforce the implementation of certain methods in the subclasses and ensure adherence to the desired interface or behavior.

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, it creates a class-level variable. The class-level variable is shared among all instances of the class.
* In Python, class-level variables are defined outside of any method or constructor within the class definition. They are typically placed at the top level of the class, just like regular variable assignments. These class-level variables can be accessed and modified by all instances of the class.
* When a class-level variable is assigned a value, it becomes part of the class's namespace and is accessible using the class name itself or through any instance of the class. All instances of the class share the same copy of the class-level variable, so modifying its value in one instance will affect its value in all other instances.

Here's an example to illustrate this:

class MyClass:

class\_variable = 10

def \_\_init\_\_(self, instance\_variable):

self.instance\_variable = instance\_variable

# Accessing class-level variable using the class name

print(MyClass.class\_variable) # Output: 10

# Creating instances of the class

obj1 = MyClass(20)

obj2 = MyClass(30)

# Accessing class-level variable through instances

print(obj1.class\_variable) # Output: 10

print(obj2.class\_variable) # Output: 10

# Modifying class-level variable through one instance

obj1.class\_variable = 50

# Accessing class-level variable through other instance

print(obj2.class\_variable) # Output: 10 (not affected by obj1's modification)

# Accessing instance variable

print(obj1.instance\_variable) # Output: 20

print(obj2.instance\_variable) # Output: 30

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 when it wants to initialize the superclass's attributes or perform any other initialization tasks defined in the superclass's \_\_init\_\_ method.
* When a subclass is created, it inherits the methods and attributes of its superclass(es). However, the subclass's \_\_init\_\_ method does not automatically invoke the superclass's \_\_init\_\_ method unless explicitly called. This means that if the superclass has any initialization logic that needs to be executed, it won't be automatically triggered when creating an instance of the subclass.
* By calling the superclass's \_\_init\_\_ method within the subclass's \_\_init\_\_ method, the subclass ensures that the initialization logic defined in the superclass is executed. This allows the subclass to inherit and initialize any attributes or perform any other necessary setup defined in the superclass.

Eg:

class Vehicle:

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

self.color = color

class Car(Vehicle):

def \_\_init\_\_(self, color, brand):

super().\_\_init\_\_(color) # Call superclass's \_\_init\_\_ method

self.brand = brand

car = Car("red", "Toyota")

print(car.color) # Output: red

print(car.brand) # Output: Toyota

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

To augment an inherited method instead of completely replacing it, you can follow these steps:

1. Define a new method in the subclass with the same name as the method you want to augment in the superclass.
2. Within the subclass method, you can call the superclass method using the super() function.
3. Perform any additional actions or modifications you want to add to the method.

By calling the superclass method using super(), you ensure that the original behavior of the inherited method is preserved while allowing you to add extra functionality.

Eg:

class Vehicle:

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

self.color = color

def start(self):

print("Vehicle started.")

class Car(Vehicle):

def \_\_init\_\_(self, color, brand):

super().\_\_init\_\_(color)

def start(self):

super().start() # Call the superclass method

print("Car started.")

car = Car("red", "Toyota")

car.start()

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

the local scope of a class is closely tied to the class itself and its instances, allowing access to instance variables, class variables, and methods within the class's methods. In contrast, the local scope of a function is more limited, confined to variables defined within the function itself and any variables passed as arguments.