1. What is the concept of an abstract superclass?

**Anwer:** An abstract superclass is a class in object-oriented programming that is designed to serve as a basis for other classes. An abstract superclass is similar to a regular superclass in that it can provide a common functionality to its subclasses, but it cannot be instantiated on its own.

The key difference between a regular superclass and an abstract superclass is that an abstract superclass defines one or more abstract methods. An abstract method is a method that does not have an implementation in the abstract superclass but is instead intended to be overridden by its subclasses.

When a class inherits from an abstract superclass, it must provide an implementation for all of the abstract methods defined in the superclass. This ensures that all subclasses of the abstract superclass have certain behavior in common, but still allows each subclass to implement that behavior in its way.

In summary, an abstract superclass is a class that provides a common interface and behavior to its subclasses but does not provide a concrete implementation of that behavior. It is designed to be subclassed and extended by other classes, which will provide concrete implementations for its abstract methods.

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

**Answer:** When a class statement's top level contains a basic assignment statement, the assignment statement creates a class variable that is shared by all instances of the class. This class variable can be accessed using the class name and the dot notation.

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

**Answer:** In object-oriented programming, a class can inherit from another class, which is called its superclass or parent class. When a subclass is created, it inherits all the attributes and methods of its superclass. However, in order to properly initialize the inherited attributes of the superclass, the subclass needs to call the superclass's **\_\_init\_\_** method explicitly.

The **\_\_init\_\_** method is a special method that is called when an object is instantiated from a class. It is responsible for initializing the object's attributes, including those inherited from its superclass. If the **\_\_init\_\_** method of a subclass does not call the **\_\_init\_\_** method of its superclass, the inherited attributes will not be properly initialized, and the subclass may not function as expected.

By convention, the **\_\_init\_\_** method of a subclass should start with a call to the **\_\_init\_\_** method of its superclass, using the **super()** function. This ensures that the superclass's **\_\_init\_\_** method is called before the subclass's **\_\_init\_\_** method, and that the inherited attributes are properly initialized.

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

**Answer:** In object-oriented programming, it is common for a subclass to inherit methods from its superclass. However, in some cases, the subclass may want to modify or extend the behavior of an inherited method without completely replacing it. This can be accomplished by using method overriding and calling the superclass's method within the overridden method.

Method overriding is a feature of object-oriented programming that allows a subclass to provide a specific implementation of a method that is already provided by its superclass. To override a method, the subclass defines a method with the same name as the method in the superclass. When an instance of the subclass calls the method, the overridden method in the subclass is executed instead of the method in the superclass.

To augment an inherited method instead of completely replacing it, the subclass can call the superclass's method within the overridden method using the **super()** function. The **super()** function returns a temporary object of the superclass, which allows the subclass to call the superclass's method.

For example,

class Animal:

def make\_sound(self):

print("The animal makes a sound")

class Dog(Animal):

def make\_sound(self):

super().make\_sound()

print("The dog barks")

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

**Answer**: In Python, a class is a type of object that defines a set of attributes and methods. Like functions, classes have their own scope, but the local scope of a class is different from that of a function in several ways.

1. Access to instance attributes and methods: In a class, the local scope includes the instance attributes and methods. These are accessed using the **self** parameter in methods. In contrast, in a function, the local scope only includes the parameters and variables defined within the function.
2. Access to class attributes and methods: In a class, the local scope also includes the class attributes and methods. These are accessed using the class name instead of **self**. In contrast, in a function, the local scope does not include the class attributes and methods.
3. Persistence across method calls: In a class, the local scope persists across method calls. This means that any variables defined within a method are still accessible in other methods of the class using the **self** parameter. In contrast, in a function, the local scope is destroyed once the function returns, and any variables defined within the function are no longer accessible.
4. Inheritance: In a class, the local scope includes the attributes and methods of its parent classes. These are accessed using the **super()** function. In contrast, in a function, there is no concept of inheritance.
5. Access to the global scope: In a class, the local scope can access the global scope using the **global** keyword. In contrast, in a function, the local scope can access the global scope using the **global** keyword or by using the **globals()** function.

In summary, the local scope of a class is different from that of a function because it includes instance attributes and methods, class attributes and methods, and persists across method calls. It also includes the attributes and methods of its parent classes and can access the global scope using the **global** keyword.