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

**Answer:**

* An **abstract superclass** is a class that **cannot be instantiated** and serves as a blueprint for subclasses.
* It defines **abstract methods** that must be implemented by subclasses.
* In Python, abstract superclasses are created using the ABC (Abstract Base Class) module.

**Example:**

from abc import ABC, abstractmethod

class Animal(ABC): # Abstract superclass

@abstractmethod

def make\_sound(self):

pass # Abstract method

class Dog(Animal): # Concrete subclass

def make\_sound(self):

return "Woof!"

# animal = Animal() # TypeError: Can't instantiate abstract class

dog = Dog()

print(dog.make\_sound()) # Output: Woof!

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

**Answer:**

* **Class attributes** are created.
* These attributes are shared among **all instances** of the class.

**Example:**

class Car:

wheels = 4 # Class attribute

c1 = Car()

c2 = Car()

print(c1.wheels, c2.wheels) # Output: 4 4

Car.wheels = 6 # Modifying class attribute

print(c1.wheels, c2.wheels) # Output: 6 6

* The attribute wheels is stored at the **class level** and is shared by all instances.

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

**Answer:**

* When a subclass **overrides** the \_\_init\_\_ method, the superclass’s \_\_init\_\_ method **does not run automatically**.
* To ensure proper **initialization of the parent class**, you must call super().\_\_init\_\_().

**Example:**

class Parent:

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

print("Parent initialized")

class Child(Parent):

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

super().\_\_init\_\_() # Calls Parent's \_\_init\_\_

print("Child initialized")

c = Child()

# Output:

# Parent initialized

# Child initialized

* Without super().\_\_init\_\_(), "Parent initialized" would **not** be printed.

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

**Answer:**

* Use super() to **call the superclass method** and **add extra functionality** instead of overriding it completely.

**Example:**

class Animal:

def make\_sound(self):

print("Some generic sound")

class Dog(Animal):

def make\_sound(self):

super().make\_sound() # Call parent method

print("Woof!")

dog = Dog()

dog.make\_sound()

# Output:

# Some generic sound

# Woof!

* super().make\_sound() **augments** the behavior of the parent class instead of replacing it.

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

**Answer:**

| **Feature** | **Class Scope** | **Function Scope** |
| --- | --- | --- |
| **Persistence** | Exists as long as the class exists. | Exists only during function execution. |
| **Attributes** | Class attributes are shared among instances. | Function variables are isolated per call. |
| **Access** | Methods and attributes can be accessed using self. | Function variables can’t be accessed outside. |

**Example:**

class Demo:

x = 10 # Class scope

def method(self):

y = 20 # Function scope

return y

print(Demo.x) # Works (Class scope)

# print(Demo.y) # Error (y is function-scoped)

* Variables inside a **class body** are persistent.
* Variables inside a **function** exist **only during execution**.