1. What is the concept of an abstract superclass?

Sol:-

The concept of an abstract superclass is a fundamental concept in object-oriented programming that allows you to create a class specifically designed to serve as a blueprint for other classes. An abstract superclass, also known as an abstract base class (ABC), provides a common interface and defines a set of methods or attributes that must be implemented by its subclasses.

An abstract superclass cannot be instantiated on its own because it is incomplete and serves as a blueprint for subclasses to inherit from. It provides a contract or specification that outlines the methods or attributes that any subclass must implement. The purpose of an abstract superclass is to define a common interface and enforce a certain structure or behavior among its subclasses.

from abc import ABC, abstractmethod

class AbstractSuperclass(ABC):

@abstractmethod

def abstract\_method(self):

pass

@property

@abstractmethod

def abstract\_property(self):

pass

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

Sol:-

When a class statement's top level contains a basic assignment statement, it creates a class attribute.

It is shared among all instances of the class. When a class attribute is defined at the top level of a class statement, it becomes accessible to all instances of that class as well as the class itself.

class MyClass:

class\_attribute = 42

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

self.instance\_attribute = instance\_attribute

# Accessing the class attribute

print(MyClass.class\_attribute) # Output: 42

# Creating instances of the class

obj1 = MyClass("Instance 1")

obj2 = MyClass("Instance 2")

# Accessing instance attributes

print(obj1.instance\_attribute) # Output: Instance 1

print(obj2.instance\_attribute) # Output: Instance 2

# Modifying the class attribute

MyClass.class\_attribute = 99

# Accessing the modified class attribute through instances

print(obj1.class\_attribute) # Output: 99

print(obj2.class\_attribute) # Output: 99

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

Sol:-

In Python, a class needs to manually call a superclass's \_\_init\_\_ method when it overrides the \_\_init\_\_ method in the subclass and wants to retain the initialization behavior of the superclass.

When a subclass overrides a method from its superclass, including the \_\_init\_\_ method, the overridden method in the superclass is no longer automatically called when creating instances of the subclass. This is because the subclass's \_\_init\_\_ method takes precedence over the superclass's \_\_init\_\_ method.

To ensure that the initialization logic of the superclass is executed when creating instances of the subclass, the subclass needs to explicitly call the superclass's \_\_init\_\_ method within its own \_\_init\_\_ method using the super() function. This allows the subclass to inherit and extend the initialization behavior of the superclass.

class Superclass:

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

self.x = x

class Subclass(Superclass):

def \_\_init\_\_(self, x, y):

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

self.y = y

obj = Subclass(10, 20)

print(obj.x) # Output: 10

print(obj.y) # Output: 20

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

Sol:-

To augment, or extend, an inherited method in Python without completely replacing it, you can use method overriding and the super() function. Method overriding allows a subclass to provide its own implementation of a method inherited from its superclass, while still retaining and invoking the superclass's implementation.

class Superclass:

def some\_method(self):

print("This is the superclass's implementation.")

class Subclass(Superclass):

def some\_method(self):

super().some\_method() # Call superclass's implementation

print("This is the subclass's additional implementation.")

obj = Subclass()

obj.some\_method()

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

Sol:-

Accessibility: In a class, the local scope refers to the scope within methods and attributes defined in the class. It allows access to class attributes, instance attributes, and local variables defined within methods. The local scope of a function, on the other hand, refers to the scope within the function's body and allows access to function parameters and local variables declared within the function.

Lifetime: The local scope of a class exists as long as the class is defined and its methods or attributes are being accessed. It persists throughout the lifetime of the class. In contrast, the local scope of a function is temporary and is created each time the function is called and destroyed when the function completes execution.

Inheritance: In the local scope of a class, methods and attributes defined in the class are accessible to its instances and any subclasses that inherit from it. Subclasses can override or extend the behavior of inherited methods within their own local scope. In the local scope of a function, there is no concept of inheritance or subclasses. Function local variables and parameters are accessible only within the function and do not have any impact on other functions or scopes.

Namespace: The local scope of a class has a namespace associated with it, which includes the names of class attributes, instance attributes, and local variables within methods. This namespace allows the class to organize and encapsulate its data and behavior. In the local scope of a function, there is also a namespace that includes function parameters and local variables.

Accessing Scope: Within a class, the local scope can access attributes and methods defined within the class, as well as the global scope (module-level scope) and built-in scope. It can access class attributes using the self or the class name itself. In contrast, the local scope of a function can access variables in its own scope, as well as variables from outer scopes such as global variables or variables from enclosing functions (in the case of nested functions).