Q1. What is the purpose of Python's OOP?

The purpose of Python's object-oriented programming (OOP) is to provide a programming paradigm that allows you to organize and structure your code in a more modular and reusable way. OOP promotes the concept of creating objects that encapsulate data and methods (functions) that operate on that data.

By leveraging OOP concepts, developers can build robust, scalable, and maintainable applications, improving code quality and productivity.

Q2. Where does an inheritance search look for an attribute?

In Python, it is determined by the C3 linearization algorithm, which uses a depth-first, left-to-right approach. When searching for an attribute, Python checks the following order:

The object itself: Python first checks whether the attribute exists in the object being accessed.

The object's class: If the attribute is not found in the object itself, Python checks the object's class.

Parent classes in the order of inheritance: If the attribute is not found in the object's class, Python continues searching in the parent classes in the order they were defined. This search follows the MRO and stops as soon as the attribute is found.

Q3. How do you distinguish between a class object and an instance object?

A class object defines the structure and behavior of a class, while an instance object is a specific occurrence or instantiation of that class. The class object serves as a blueprint, whereas the instance object represents a unique object with its own set of attributes and methods.

Q4. What makes the first argument in a class’s method function special?

The first argument in a class's method function is conventionally named self, although you can technically use any valid variable name. The self parameter is special because it refers to the instance object that the method is being called on. Here's what makes the first argument special:

Instance Binding: When a method is called on an instance object, the instance itself is automatically passed as the first argument to the method. This allows the method to access and manipulate the instance's attributes and perform instance-specific operations. By convention, the first parameter is named self to indicate this instance binding.

Accessing Instance Attributes: Inside the method, you can use self to refer to the instance object and access its attributes. For example, if the instance has an attribute named name, you can access it using self.name. This enables the method to work with the specific state of the instance.

Calling Other Methods: The self parameter allows methods within a class to call other methods within the same class. By using self.method\_name(), you can invoke another method on the same instance object, enabling code reuse and encapsulation.

Creating and Modifying Attributes: The self parameter also allows methods to create new instance attributes or modify existing ones. For example, a method can assign a value to self.attribute\_name, creating or updating an attribute specific to that instance.

Q5. What is the purpose of the \_\_init\_\_ method?

The \_\_init\_\_ method, also known as the constructor, is a special method in a class that is automatically called when an instance of the class is created. It is used to initialize the attributes and perform any necessary setup for the instance. Here are the main purposes of the \_\_init\_\_ method:

Initializing Instance Attributes: The primary purpose of the \_\_init\_\_ method is to initialize the attributes of the instance object. It allows you to set initial values for the object's attributes, providing a way to define the state of the object when it is created. Inside the \_\_init\_\_ method, you can assign values to instance variables using the self parameter. This allows each instance to have its own unique set of attribute values.

Accepting Parameters during Object Creation: The \_\_init\_\_ method can accept parameters that provide values for initializing the instance attributes. These parameters are typically passed when creating an instance of the class. For example, if a class represents a person, the \_\_init\_\_ method could accept parameters like name, age, and gender, and assign them to corresponding instance attributes.

Performing Setup or Initialization Tasks: Apart from attribute initialization, the \_\_init\_\_ method can also perform other setup or initialization tasks that are necessary for the instance. This could include opening files, establishing connections, or any other actions required to prepare the instance for use.

Implicit Invocation: When you create an instance of a class, the \_\_init\_\_ method is implicitly called, allowing you to ensure that necessary initialization steps are executed automatically. This helps maintain the integrity and consistency of the instance by initializing its attributes and setting it up properly.

Q6. What is the process for creating a class instance?

# Step 1: Class Definition

class Person:

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

self.name = name

def say\_hello(self):

print("Hello, my name is", self.name)

# Step 2: Instance Creation

person1 = Person("Alice")

Q7. What is the process for creating a class?

# Step 1: Class Definition

class Rectangle:

# Step 2: Attributes

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

self.width = width

self.height = height

# Step 3: Methods

def area(self):

return self.width \* self.height

# Step 4: Instantiation

rectangle1 = Rectangle(4, 5)

Q8. How would you define the superclasses of a class?

# Step 1: Class Definition

class Animal:

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

self.name = name

def speak(self):

print("Animal speaks")

# Step 2: Inheritance Syntax

class Dog(Animal):

def \_\_init\_\_(self, name, breed):

super().\_\_init\_\_(name)

self.breed = breed

def speak(self):

print("Dog barks")

# Step 3: Accessing Superclass Features

dog = Dog("Buddy", "Labrador")

dog.speak() # Output: Dog barks

print(dog.name) # Output: Buddy