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 for the organization, structure, and management of complex code by modeling real-world objects and their interactions.

Here are some key purposes and benefits of using OOP in Python:

1.Modularity and Code Reusability: OOP allows you to break down a complex program into modular objects and classes. Objects encapsulate data and behavior, making it easier to understand and maintain code. By creating reusable classes, you can avoid duplicating code and promote code reusability across different parts of your program.

2.Abstraction and Encapsulation: OOP allows you to abstract complex systems by focusing on the essential features and interactions of objects. Encapsulation hides the internal details of objects, exposing only necessary interfaces. This promotes code organization, reduces complexity, and enhances code maintainability.

3.Inheritance and Polymorphism: Inheritance allows you to create new classes that inherit properties and behaviors from existing classes. This promotes code reuse and supports hierarchical relationships between classes. Polymorphism enables objects of different classes to be used interchangeably, providing flexibility and extensibility to your code.

4.Data Modeling and Conceptualization: OOP provides a way to model real-world entities, concepts, and relationships in your code. By mapping objects, attributes, and methods to real-world entities, you can design software systems that closely resemble the problem domain, making it easier to understand and communicate about the code.

5.Code Organization and Maintenance: OOP promotes a structured approach to code organization, making it easier to manage and maintain large projects. Objects and classes provide a natural way to partition functionality and responsibilities, allowing different team members to work on different modules independently. This supports code scalability and collaboration.

6.Code Extensibility and Flexibility: OOP enables you to extend and modify code without impacting existing functionality. You can add new classes, inherit from existing ones, and override or extend their methods. This promotes flexibility and adaptability to changing requirements.

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

In Python, when you access an attribute (such as a variable or method) on an object, the inheritance search for that attribute follows a specific order known as the Method Resolution Order (MRO). The MRO determines where Python looks for the attribute when it is not found in the immediate object.

The inheritance search for an attribute looks in the following order:

1. The instance itself: Python first checks if the attribute exists in the instance object itself. If the attribute is found, it is used, and the search stops.

2. The class of the instance: If the attribute is not found in the instance, Python looks for it in the class of the instance. If the attribute exists in the class, it is used, and the search stops.

3. The base classes in the order of inheritance: If the attribute is not found in the instance or the class, Python continues the search in the base classes in the order they are specified in the class definition. This follows the method resolution order (MRO) of the class. Python checks each base class in order until it finds the attribute or reaches the end of the inheritance chain.

4. The global and built-in namespaces: If the attribute is not found in any of the above steps, Python checks in the global namespace (module-level scope) and then the built-in namespace (which contains built-in functions and objects). If the attribute is still not found, an AttributeError is raised.

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

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

In Python, the first argument in a class's method function is conventionally named self, although any valid variable name can be used. This first argument represents the instance object on which the method is being called. It is a reference to the instance itself.

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

The \_\_init\_\_ method is a special method in Python classes that is automatically called when an instance of the class is created. It stands for "initialize" and is used to initialize the attributes of the instance object. The primary purpose of the \_\_init\_\_ method is to perform any necessary setup or initialization tasks when creating a new instance. It allows you to define and assign initial values to the instance variables and perform any other operations required before using the object.

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

The process for creating a class instance involves a few steps:

Class Definition: First, you need to define the class itself. The class serves as a blueprint or template for creating instances. It defines the structure and behavior of the objects that will be created from it.

Instance Creation: To create an instance of a class, you use the class name followed by parentheses. This is similar to calling a function. The parentheses can be empty or contain arguments that are passed to the \_\_init\_\_ method if it is defined in the class.

Instance Initialization: If the class has an \_\_init\_\_ method, it is automatically called during instance creation. The \_\_init\_\_ method is responsible for initializing the attributes and performing any necessary setup tasks. The instance variables can be set to default values or based on the provided arguments.

Instance Usage: Once the instance is created and initialized, you can use it to access its attributes and invoke its methods. The instance variables can be accessed and modified using dot notation (instance.variable) and the methods can be called using parentheses (instance.method()).

Eg:

# Class Definition

class MyClass:

def \_\_init\_\_(self, arg1, arg2):

self.attribute1 = arg1

self.attribute2 = arg2

def some\_method(self):

print("Inside some\_method")

# Instance Creation

my\_instance = MyClass("value1", "value2")

# Accessing instance variables

print(my\_instance.attribute1

print(my\_instance.attribute2)

# Calling a method

my\_instance.some\_method()

Q7. What is the process for creating a class?

The process for creating a class in Python involves the following steps:

Class Declaration: To create a class, you need to declare it using the class keyword followed by the class name. The class name should be written in CamelCase notation, starting with an uppercase letter by convention.

Class Definition: Inside the class block, you define the attributes and methods that belong to the class. Attributes are variables that store data, and methods are functions that define the behavior of the class.

Instantiation: After defining the class, you can create instances of the class, also known as objects. Instantiation is the process of creating an instance of a class. To create an instance, you call the class as if it were a function, with parentheses after the class name. This creates a new object based on the class's definition.

Attribute Assignment: Once an instance is created, you can assign values to its attributes. Attributes are accessed using dot notation (instance.attribute) and can be assigned values specific to that instance.

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

The superclasses of a class, also known as parent classes or base classes, are the classes from which the current class inherits attributes and methods. In Python, class inheritance allows a class to inherit properties and behaviors from one or more superclasses.