Python OOP Assignment

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

Sol)

In the context of programming, "modular" refers to a software design approach where a large system or program is divided into smaller, independent, and self-contained modules or components. Each module performs a specific task or functionality and can be developed, tested, and maintained separately.

The key characteristics of modular programming are:

- Modularity: Modularity is the concept of breaking down a complex problem into smaller, manageable parts. Each module focuses on a specific task or functionality, and modules can be developed and maintained independently. Modularity allows for better organization, readability, and maintainability of code.
- Encapsulation: Each module encapsulates its internal details, hiding them from other modules. This promotes information hiding and allows modules to interact through well-defined interfaces or APIs (Application Programming Interfaces). Encapsulation helps in reducing dependencies between modules and isolating changes within a module.
- Reusability: Modular design promotes code reuse. Modules can be developed as reusable components that can be used in multiple projects or integrated into different parts of the same project. Reusing modules saves development time, reduces errors, and improves overall software quality.
- Abstraction: Modules provide a higher level of abstraction, allowing complex functionality to be encapsulated and presented as a simple interface. This allows other parts of the program to interact with the module without needing to understand the internal implementation details.

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

Sol)

When an attribute is accessed on an object in Python, the inheritance search looks for the attribute in the following order:

- The instance itself: The search starts by checking if the attribute is present directly in the instance of the object being accessed. If the attribute is found, the search stops, and that attribute is used.
- The class: If the attribute is not found in the instance, the search moves to the class of the object. It checks if the attribute is defined in the class. If the attribute is found, it is used, and the search stops.
- Parent classes (inherited classes): If the attribute is not found in the class, the search continues in the parent classes. It follows the inheritance hierarchy, checking each parent class in the order they were defined. The search stops as soon as the attribute is found in one of the parent classes.
- Base object class: If the attribute is not found in the object's class or any of its parent classes, the search continues in the base object class, which is the ultimate parent class for all objects in Python. If the attribute is found in the base object class, it is used.
- AttributeError: If the attribute is not found in any of the above steps, a runtime AttributeError is raised, indicating that the attribute does not exist in the object or its inheritance hierarchy.

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

Sol)

	Class Object	Instance Object
Definition	Created when a class is defined	Created when a class is instantiated (using the class as a constructor)
Purpose	Represents the blueprint or template for creating instances	Represents a specific occurrence or individual of a class
Usage	Defines attributes and methods common to all instances	Has its own unique attributes and can invoke class methods
Access	Accessed directly through the class name	Created by calling the class object as a constructor

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

Sol)

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 is special and holds a reference to the instance of the class that the method is being called on. It is automatically passed as an argument to the method when it is invoked.

Here's why the self parameter is special and its significance in class methods:

- Instance Binding: By convention, the first parameter of a class method is self. It acts as a reference to the instance of the class that the method is being called on. When a method is invoked on an instance, self is automatically bound to that instance. It allows the method to access and manipulate the instance's attributes and invoke other methods on itself.
- Accessing Instance Attributes: The self parameter provides access to the instance attributes within the method. It allows you to read or modify the

state of the instance. By referencing self.attribute_name, you can access and manipulate the instance's attributes.

- Method Invocation: When a method is called on an instance, Python automatically passes the instance as the first argument (self) to the method. This is why you don't explicitly pass the self argument when invoking the method.

Q5. What is the purpose of the init method?

Sol)

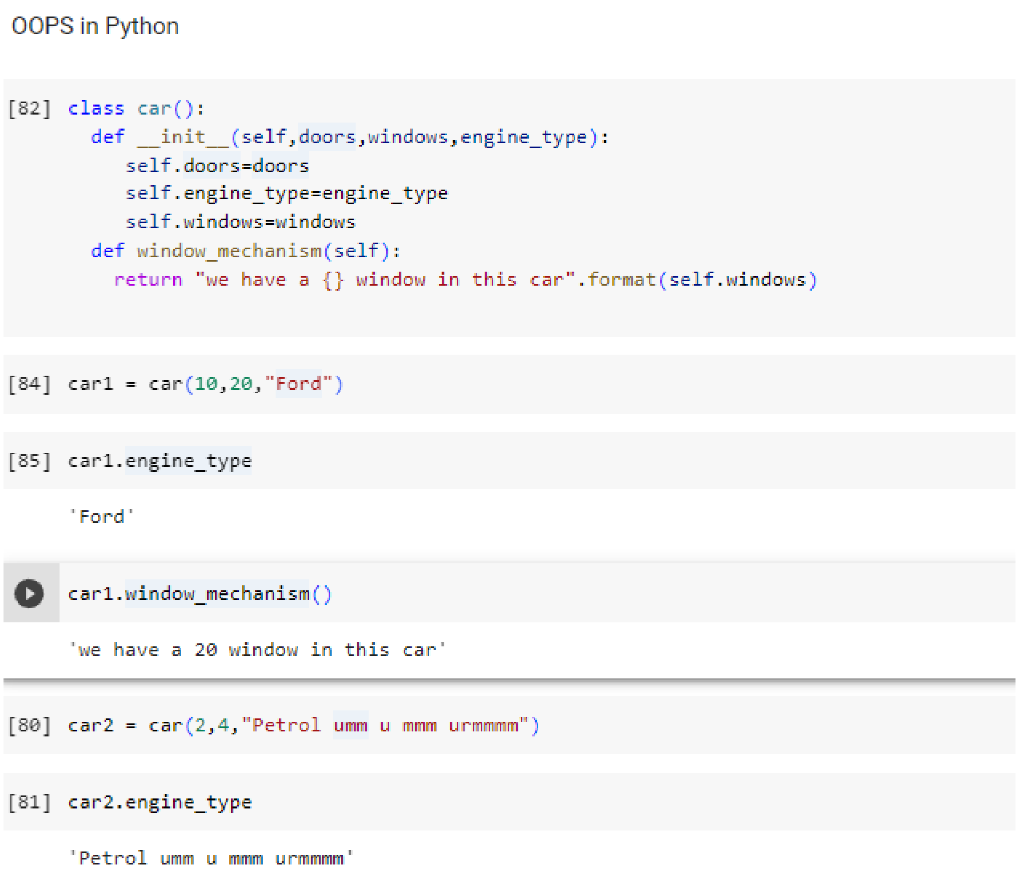
The __init__() method, also known as the constructor, is a special method in Python classes. It is automatically called when an object is created from the class and is used to initialize the object's attributes or perform any other setup tasks required for the object.

Here are the key purposes and characteristics of the __init__() method:

- Object Initialization: The primary purpose of the __init__() method is to initialize the state of an object. It allows you to define and assign initial values to the object's attributes. By defining the __init__() method, you can ensure that certain attributes are always set when creating an instance of the class.
- Automatic Invocation: The __init__() method is automatically called when an object is created from the class. When you create an instance of a class using the class name followed by parentheses (e.g., my_object = MyClass()), Python automatically invokes the __init__() method associated with that class.
- Parameters: The __init__() method takes the self parameter as the first argument, followed by any additional parameters you want to specify. The self parameter refers to the instance being created and allows you to access and modify its attributes. Any additional parameters can be used to pass initial values to the object's attributes.
- Attribute Initialization: Inside the __init__() method, you can use the self parameter to assign values to the object's attributes. These assignments typically involve assigning the parameter values to corresponding instance variables using dot notation (e.g., self.attribute = parameter_value).

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

Sol)



OOPS in Python

```
[82] class car():
       def __init__(self,doors,windows,engine_type):
          self.doors=doors
          self.engine_type=engine_type
          self.windows=windows
       def window_mechanism(self):
         return "we have a {} window in this car".format(self.windows)
```

```
[84] car1 = car(10,20,"Ford")
```

```
[85] car1.engine_type
```

```
'Ford'
```

```
car1.window_mechanism()
```

```
'we have a 20 window in this car'
```

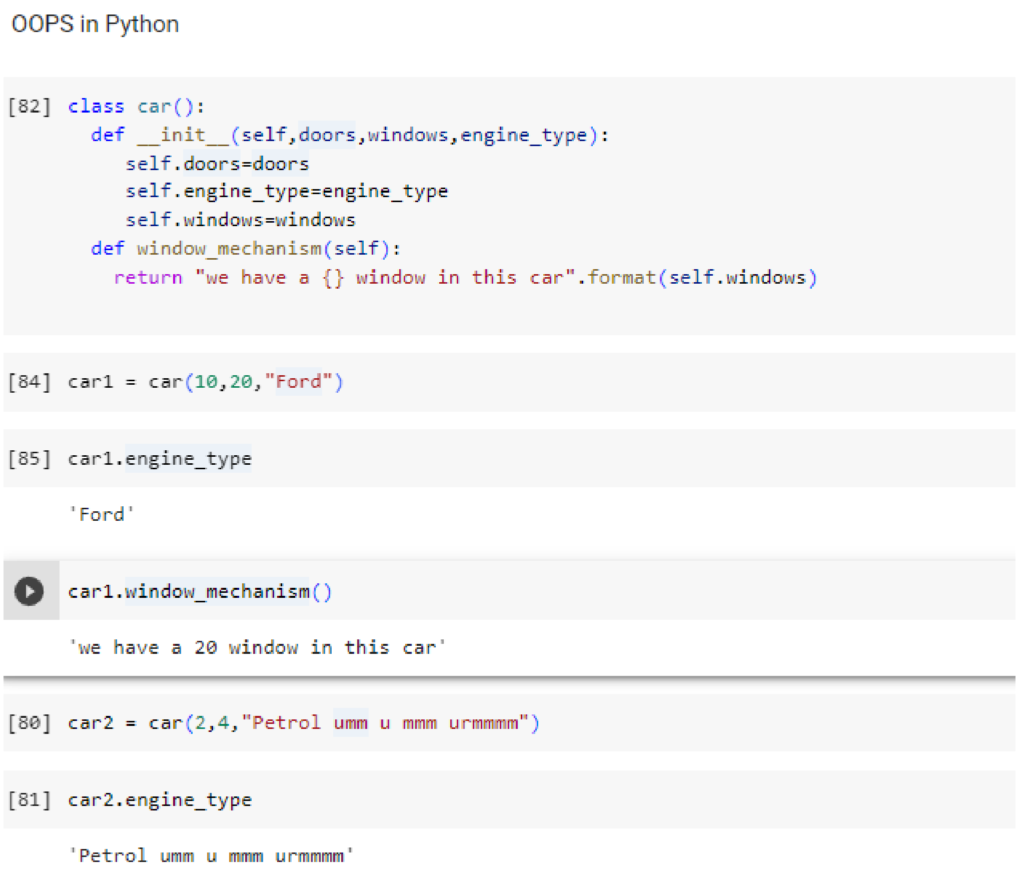
```
[80] car2 = car(2,4,"Petrol umm u mmm urmmmm")
```

```
[81] car2.engine_type
```

```
'Petrol umm u mmm urmmmm'
```

Q7. What is the process for creating a class?

Sol)


OOPS in Python

```
[82] class car():
       def __init__(self,doors,windows,engine_type):
          self.doors=doors
          self.engine_type=engine_type
          self.windows=windows
       def window_mechanism(self):
         return "we have a {} window in this car".format(self.windows)
```

```
[84] car1 = car(10,20,"Ford")
```

```
[85] car1.engine_type

     'Ford'
```

```
     car1.window_mechanism()

     'we have a 20 window in this car'
```

```
[80] car2 = car(2,4,"Petrol umm u mmm urmmmm")
```

```
[81] car2.engine_type

     'Petrol umm u mmm urmmmm'
```

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

Sol)

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

Q9. What is the relationship between classes and modules?

Sol)

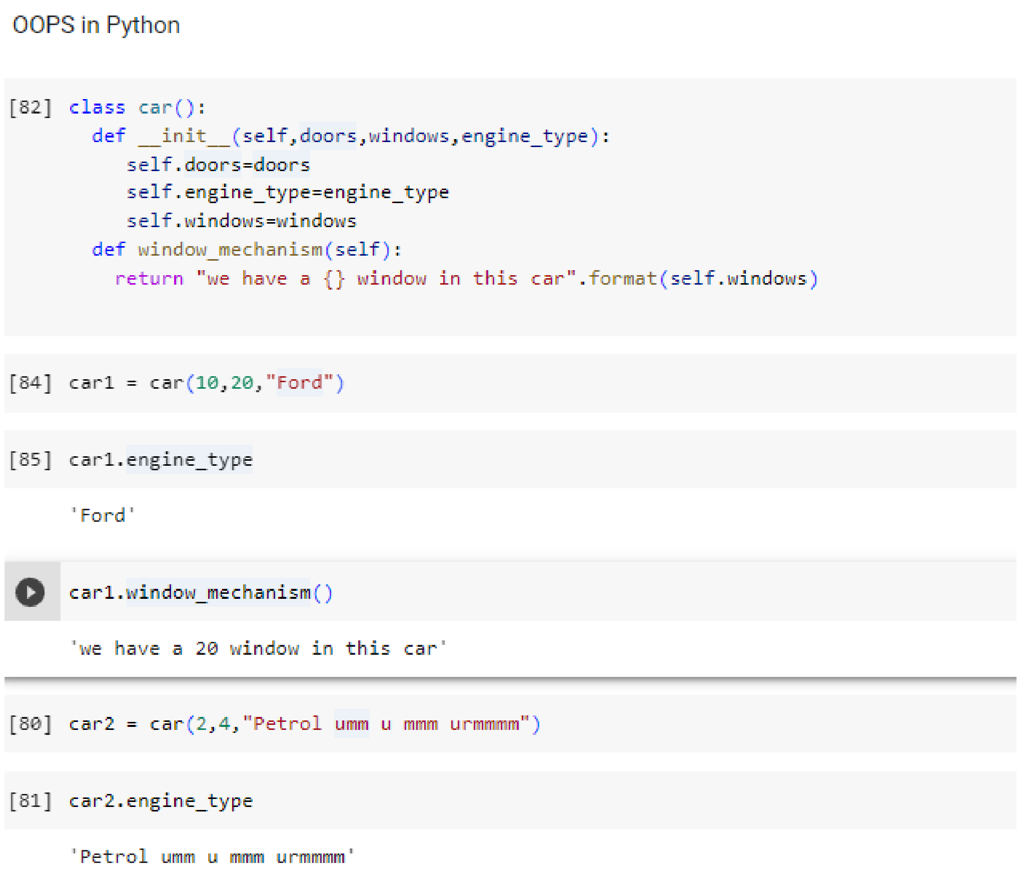
Relationship:

- Classes can be defined within modules: You can define classes within a module to encapsulate related functionality. By defining classes within a module, you can organize your code in a structured manner and group related behavior together.

- Modules can contain classes: A module can define one or more classes along with other code elements. This allows you to package related classes and other code together in a single module for easy reuse and import.
- Classes can be imported from modules: To use a class defined in a module, you can import the class into another script or module. Importing a class allows you to create instances of the class and utilize its attributes and methods in the importing code.
- Modules can import other modules: Similarly, modules can import other modules to access and use their code. This allows for code reuse, modular design, and organization of functionality across multiple modules.

Q10. How do you make instances and classes?

Sol)



OOPS in Python

```
[82] class car():
       def __init__(self,doors,windows,engine_type):
          self.doors=doors
          self.engine_type=engine_type
          self.windows=windows
       def window_mechanism(self):
         return "we have a {} window in this car".format(self.windows)
```

```
[84] car1 = car(10,20,"Ford")
```

```
[85] car1.engine_type

     'Ford'
```

```
     car1.window_mechanism()

     'we have a 20 window in this car'
```

```
[80] car2 = car(2,4,"Petrol umm u mmm urmmmm")
```

```
[81] car2.engine_type

     'Petrol umm u mmm urmmmm'
```

Q11. Where and how should be class attributes created?

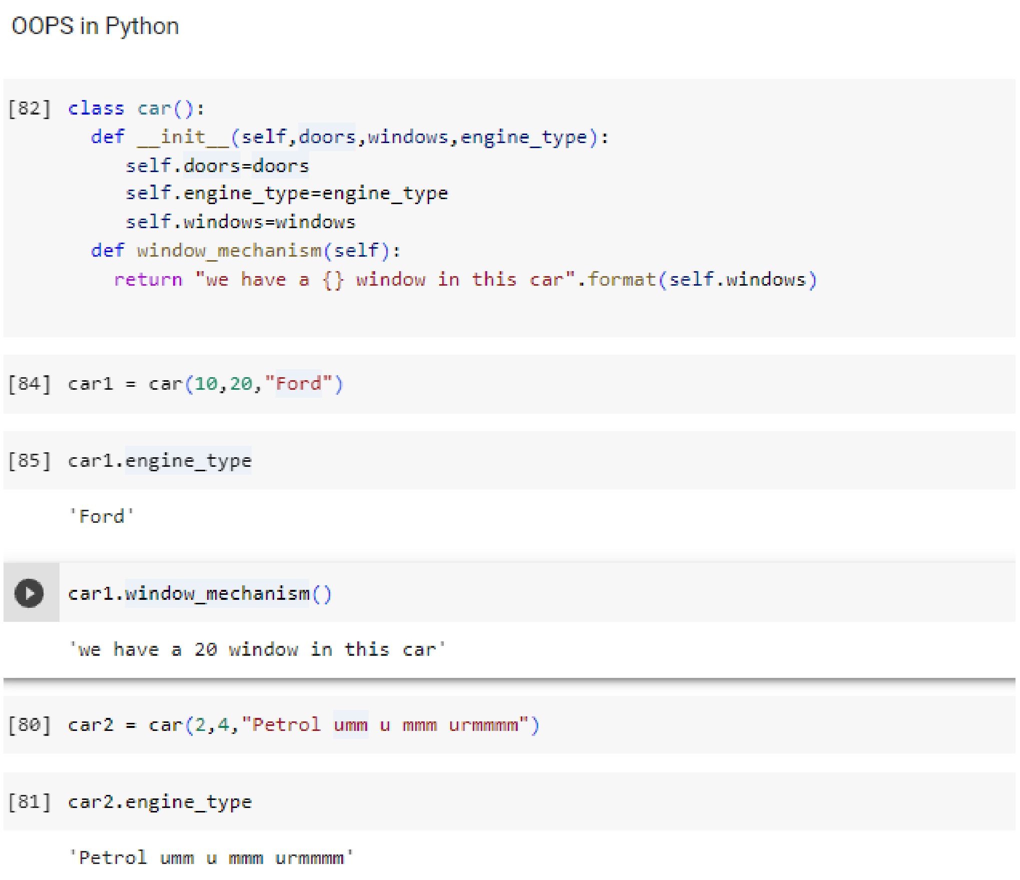
Sol)

Class attributes in Python are created within the class definition, outside of any method. They are defined directly beneath the class declaration but outside of any methods or constructor.

```
class MyClass:

    class_attribute = "Hello"  # Class attribute defined within the class

    def __init__(self, instance_attribute):

        self.instance_attribute = instance_attribute
```

Q12. Where and how are instance attributes created?

Sol)



OOPS in Python

```
[82] class car():
       def __init__(self,doors,windows,engine_type):
          self.doors=doors
          self.engine_type=engine_type
          self.windows=windows
       def window_mechanism(self):
         return "we have a {} window in this car".format(self.windows)
```

```
[84] car1 = car(10,20,"Ford")
```

```
[85] car1.engine_type
```

```
'Ford'
```

```
car1.window_mechanism()
```

```
'we have a 20 window in this car'
```

```
[80] car2 = car(2,4,"Petrol umm u mmm urmmmm")
```

```
[81] car2.engine_type
```

```
'Petrol umm u mmm urmmmm'
```

Q13. What does the term "self" in a Python class mean?

Sol)

In Python, the term "self" is used as a convention to refer to the instance of a class within its own methods. It is a special parameter that must be included as the first parameter in the method definition of a class.

The purpose of "self" is to allow the instance (object) of a class to refer to its own attributes and methods. It acts as a reference to the instance itself, allowing access to instance variables and methods within the class.

Q14. How does a Python class handle operator overloading?

Sol)

In Python, operator overloading allows you to define or redefine the behavior of built-in operators (+, -, *, /, etc.) for objects of a class. It enables objects to respond to operator symbols in a custom way, providing more flexibility and expressiveness.

To handle operator overloading in a Python class, you can define special methods, also known as magic methods or dunder methods (double underscore methods).

Q15. When do you consider allowing operator overloading of your classes?

Sol)

Operator overloading should be considered when it enhances the clarity and expressiveness of your class and provides a natural and intuitive way to perform operations on objects of that class. Here are some scenarios where allowing operator overloading can be beneficial:

- Mimicking mathematical operations: If your class represents a mathematical concept or data structure, overloading operators can make the code more readable and intuitive. For example, overloading the + operator for a Vector class to perform vector addition or overloading the * operator for a Matrix class to perform matrix multiplication.
- Customizing comparison and equality checks: By overloading comparison operators (<, <=, >, >=) and equality operators (==, !=), you can define specific criteria for comparing objects of your class. This can be useful when you want to compare objects based on certain attributes or properties.
- Simplifying syntax and improving readability:

Q16. What is the most popular form of operator overloading?

Sol)

In Python, one of the most popular forms of operator overloading is the implementation of the arithmetic operators (+, -, *, /, //, %, **) for custom classes. This allows objects of a class to behave like numeric types and perform arithmetic operations in a natural and intuitive manner.

By overloading these arithmetic operators, you can define how objects of your class should behave when involved in addition, subtraction, multiplication, division, and other arithmetic operations. This enables you to create custom numeric types or data structures that can be used seamlessly with the built-in arithmetic operators.

Q17. What are the two most important concepts to grasp in order to comprehend Python OOP code?

Sol)

The two most important concepts to grasp in order to comprehend Python object-oriented programming (OOP) code are:

- Classes and Objects: Understanding the concept of classes and objects is fundamental to OOP in Python. A class is a blueprint or template that defines the attributes (variables) and methods (functions) that objects of that class will have. An object, on the other hand, is an instance of a class. It represents a specific entity or thing created based on the class definition. Objects can have their own unique state and behavior while also inheriting attributes and methods from their class.
- Inheritance and Polymorphism: Inheritance is a powerful concept in OOP that allows you to create new classes (derived or child classes) based on existing classes (base or parent classes). The derived classes inherit attributes and methods from their parent classes and can also add new ones or modify the existing ones. This promotes code reusability and allows for creating more specialized classes based on general ones. Polymorphism, on the other hand, refers to the ability of objects of different classes to respond to the same

method calls in different ways. It allows you to treat objects of different classes as if they belong to a common superclass, simplifying code and promoting flexibility.

Q18. Describe three applications for exception processing.

Sol)

Exception processing, also known as exception handling, is a crucial aspect of programming that helps handle and manage errors or exceptional conditions that may occur during program execution. Here are three applications for exception processing:

- Error Handling:
- Input Validation:
- Resource Management:

Q19. What happens if you don't do something extra to treat an exception?

Sol)

If you don't do something extra to treat an exception, it will propagate up the call stack until it is caught and handled by an appropriate exception handler. If no exception handler is found, the program will terminate and an error message, including the traceback, will be displayed.

Q20. What are your options for recovering from an exception in your script?

Sol)

- Handle the Exception:
- Retry the Operation
- Provide Default Values or Fallback Behavior
- Graceful Termination:

Q21. Describe two methods for triggering exceptions in your script.

Sol)

In Python, there are various methods for triggering exceptions in your script. Here are two common methods:

- Raise an Exception Explicitly: You can raise an exception explicitly using the raise statement. This allows you to raise built-in exceptions or create custom exceptions to indicate specific error conditions. To raise an exception, you specify the type of the exception and an optional error message or arguments
- Invoke Functions or Methods that Raise Exceptions: Another method for triggering exceptions is by invoking functions or methods that are designed to raise exceptions. Many built-in Python functions and methods raise exceptions in response to certain conditions. For example, the open() function raises a FileNotFoundError if the specified file cannot be found, and the int() function raises a ValueError if the input argument cannot be converted to an integer.

Q22. Identify two methods for specifying actions to be executed at termination time, regardless of whether or not an exception exists.

Sol)

- Finally Block
- Context Managers

Q23. What is the purpose of the try statement?

Sol)

The purpose of the try statement in Python is to define a block of code where exceptional conditions or errors might occur. It allows you to handle exceptions and gracefully recover from errors, preventing the program from terminating abruptly.

Q24. What are the two most popular try statement variations?

Sol)

- try-except
- try-except-else-finally

Q25. What is the purpose of the raise statement?

Sol)

The raise statement in Python is used to explicitly raise exceptions in your code. It allows you to signal exceptional conditions or errors programmatically.

Q26. What does the assert statement do, and what other statement is it like?

Sol)

The assert statement in Python is used to assert or validate that a given condition is true. It acts as a debugging aid and helps in detecting logical errors or inconsistencies in the code. If the condition specified in the assert statement evaluates to False, it raises an AssertionError exception.

Q27. What is the purpose of the with/as argument, and what other statement is it like?

Sol)

The with/as statement in Python is used for context management, providing a convenient and concise way to handle resources that need to be properly managed, such as files, network connections, and locks. It ensures that the resources are automatically acquired and released in a controlled manner, even in the presence of exceptions or errors.

Q28. What are *args, **kwargs?

Sol)

In Python, *args and **kwargs are special syntaxes used in function definitions to handle a variable number of arguments.

Q29. How can I pass optional or keyword parameters from one function to another?

Sol)

To pass optional or keyword parameters from one function to another in Python, you can use the *args and **kwargs syntax in the function call. Here's how you can do it:

Q30. What are Lambda Functions?

Sol)

Lambda functions, also known as anonymous functions, are small and anonymous functions in Python that are defined without a function name. They are created using the lambda keyword and can take any number of arguments but can only have a single expression as their body. Lambda functions are typically used when you need a small function for a short period of time and don't want to define a separate named function.

Q31. Explain Inheritance in Python with an example?

Sol)

Inheritance is a fundamental concept in object-oriented programming that allows a class to inherit attributes and methods from another class. In Python, a class can inherit from another class, called the superclass or parent class, to create a new class, called the subclass or child class. The child class inherits all the attributes and behaviors of the parent class, and can also add its own unique attributes and behaviors

Q32. Suppose class C inherits from classes A and B as class C(A,B).Classes A and B both have their own versions of method func(). If we call func() from an object of class C, which version gets invoked?

Sol)

In Python, when a class inherits from multiple parent classes and both parent classes have a method with the same name, the method resolution order (MRO) determines which version of the method gets invoked.

The MRO defines the order in which Python searches for methods and attributes in a class hierarchy. It ensures that methods are inherited and overridden in a predictable and consistent manner.

In the case of class C inheriting from classes A and B as class C(A, B), the MRO is determined by the C3 linearization algorithm, which follows a depth-first left-to-right traversal of the inheritance hierarchy.

Q33. Which methods/functions do we use to determine the type of instance and inheritance?

Sol)

```
class MyClass:

    pass

obj = MyClass()

print(type(obj))  # Output: <class '__main__.MyClass'>
```

Q34.Explain the use of the 'nonlocal' keyword in Python.

Sol)

In Python, the nonlocal keyword is used to declare a variable in a nested function that is not local to that function but is defined in an outer (enclosing) function. It allows you to access and modify a variable in the nearest enclosing scope that is not global.

Q35. What is the global keyword?

Sol)

In Python, the global keyword is used to declare that a variable inside a function should be treated as a global variable, rather than a local variable. It allows you to modify global variables from within a function's scope.