Questions

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

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

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?

Q5. What is the purpose of the init method?

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

Q7. What is the process for creating a class?

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

Q9. What is the relationship between classes and modules?

Q10. How do you make instances and classes?

Q11. Where and how should be class attributes created?

Q12. Where and how are instance attributes created?

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

Q14. How does a Python class handle operator overloading?

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

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

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

Q18. Describe three applications for exception processing.

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

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

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

Q22. Identify two methods for specifying actions to be executed at termination time, regardless of

whether or not an exception exists.

Q23. What is the purpose of the try statement?

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

Q25. What is the purpose of the raise statement?

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

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

Q28. What are *args, **kwargs?

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

Q30. What are Lambda Functions?

Q31. Explain Inheritance in Python with an example?

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?

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

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

Q35. What is the global keyword?

Answers

Q1. The purpose of Python's object-oriented programming (OOP) is to organize and structure code into reusable and modular components called objects. OOP provides a way to model real-world entities and their relationships using classes and objects. It promotes concepts such as encapsulation, inheritance, and polymorphism, which help in writing more maintainable and extensible code.

Q2. Inheritance search in Python looks for an attribute in the following order:

1. The instance itself (object).
2. The class that the instance belongs to.
3. The superclasses of the class (in the order specified during class definition).

Q3. In Python, a class object is an instance of its metaclass, usually the built-in `type` class. It represents the blueprint or template for creating instances. On the other hand, an instance object is a specific realization of a class. It is created by calling the class as a function and represents a unique instance of that class.

Q4. The first argument in a class's method function is conventionally named `self` and refers to the instance object that the method is being called on. This argument allows the method to access and operate on the instance's attributes and methods. It is automatically passed by Python when invoking the method on an instance.

Q5. The `__init__` method, also known as the initializer or constructor, is a special method in Python classes. It is called automatically when a new instance of the class is created. The purpose of the `__init__` method is to initialize the instance's attributes and perform any setup required for the object to be in a valid state.

Q6. To create a class instance in Python, you typically follow these steps:

1. Define a class with the necessary attributes and methods.
2. Instantiate the class by calling it as a function, optionally passing any required arguments.
3. The instantiation creates a new instance object.
4. You can then access and manipulate the instance's attributes and call its methods using dot notation.

Q7. The process of creating a class in Python involves:

1. Define the class using the `class` keyword followed by the class name.
2. Optionally, specify one or more superclasses inside parentheses to inherit from them.
3. Inside the class, define attributes and methods that describe the behavior and properties of objects created from the class.

Q8. The superclasses of a class are the classes from which the current class inherits. In Python, you can define the superclasses of a class by specifying them inside parentheses after the class name during class definition. For example: `class MyClass(SuperClass1, SuperClass2)`.

Q9. Classes and modules are two different concepts in Python, but they can be related. A class is a blueprint for creating objects, while a module is a file containing Python code. Modules can contain class definitions, and classes can be defined inside modules. Classes provide a way to organize and encapsulate code, whereas modules provide a way to organize and package related code files.

Q10. To create instances, you instantiate a class by calling it as a function. For example, if you have a class named `MyClass`, you can create an instance like this: `my_object = MyClass()`. To create a class, you define it using the `class` keyword followed by the class name and its body.

Q11. Class attributes are created within the class definition, outside of any method. They are typically defined directly below the class declaration line. Class attributes are shared among all instances of a class and can be accessed using the class name or any instance of the class.

Q12. Instance attributes are created inside the

methods of a class. They are typically created and assigned to using the `self` parameter, which refers to the instance object. Each instance of a class can have its own set of instance attributes, which are independent of other instances.

Q13. In a Python class, the term "self" is a conventional name for the first parameter of instance methods. It refers to the instance object itself on which the method is being called. By using `self`, you can access the instance's attributes and methods within the method definition.

Q14. Python classes can handle operator overloading by defining special methods with double underscores (e.g., `__add__`, `__sub__`) that correspond to specific operators. These methods allow objects of the class to behave like built-in types and define their own behavior for arithmetic, comparison, and other operators.

Q15. Operator overloading is considered when you want to define custom behavior for operators in your class. If you want instances of your class to support arithmetic operations, comparisons, or other operators, you can define the corresponding special methods to specify the desired behavior.

Q16. The most popular form of operator overloading is implementing arithmetic operators such as addition, subtraction, multiplication, and division using the special methods `__add__`, `__sub__`, `__mul__`, and `__div__`, respectively. These methods allow instances of a class to participate in arithmetic operations using the corresponding operators.

Q17. The two most important concepts to grasp in order to comprehend Python OOP code are:

1. Classes and Objects: Understanding how to define and use classes to create objects with attributes and behaviors.

2. Inheritance and Polymorphism: Understanding how classes can inherit attributes and methods from other classes, and how objects can exhibit different behaviors based on their specific class or superclass.

Q18. Three applications for exception processing in Python are:

1. Error handling: Catching and handling runtime errors and exceptions to prevent program crashes and provide graceful error recovery.

2. Resource management: Using exception handling to ensure proper cleanup and release of resources, such as closing files or network connections.

3. Flow control: Employing exceptions to control the flow of program execution by raising and catching exceptions at appropriate points.

Q19. If an exception is not handled or caught, it propagates up the call stack until it reaches the top-level of the program. If it remains unhandled, it will result in the program terminating and displaying an error message.

Q20. When an exception occurs in a script, you have several options for recovering from it:

1. Handle the exception using a try-except block to catch the exception and execute specific code to handle the error gracefully.

2. Propagate the exception up the call stack by not catching it, allowing it to be handled by an enclosing try-except block.

3. Terminate the program by not handling the exception, resulting in an error message and program termination.

Q21. Two methods for triggering exceptions in your script are:

1. Using the `raise` statement: You can explicitly raise an exception using the `raise` keyword followed by an exception type or instance. This allows you to indicate a specific error condition or situation.

2. Invoking built-in functions: Certain built-in functions, such as `assert` or specific operations like dividing by zero (`ZeroDivisionError`), can trigger exceptions if certain conditions are not met.

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

1. Using the `finally` block: It is part of a try-except statement and allows you to specify code that will be executed regardless of whether

an exception was raised or not. The `finally` block is useful for cleanup tasks or releasing resources.

2. Defining an exit handler with the `atexit` module: By registering functions using the `atexit.register()` function, you can ensure that these functions will be executed when the script terminates, regardless of exceptions.

Q23. The purpose of the try statement in Python is to define a block of code where exceptions can occur. It is followed by one or more except blocks, which specify how to handle specific exceptions that may occur within the try block. The try statement allows for the implementation of error handling and exception processing.

Q24. The two most popular try statement variations are:

1. try-except: This variation allows you to catch and handle specific exceptions. The code within the `try` block is executed, and if an exception of the specified type occurs, the corresponding `except` block is executed to handle the exception.

2. try-except-else: In addition to the try-except block, this variation includes an `else` block. The code within the `else` block is executed only if no exceptions occur in the `try` block. It allows you to specify code that should run only when the try block is successful.

Q25. The `raise` statement in Python is used to explicitly raise an exception. It allows you to indicate that a specific error condition or situation has occurred. The `raise` statement is followed by an exception type or instance that represents the type of exception being raised.

Q26. The `assert` statement in Python is used to test a condition. If the condition evaluates to False, the `assert` statement raises an `AssertionError` exception. It is often used for debugging and to enforce certain conditions that must be true at specific points in the program. The `assert` statement is similar to an if statement, but it has a specific purpose of raising an exception when the condition is not met.

Q27. The purpose of the `with/as` argument, also known as the "context manager," is to ensure the proper acquisition and release of resources. It is used in combination with file objects, network connections, and other resources that need to be managed. The `with` statement sets up the context for the resource, and the `as` keyword assigns the resource to a variable. It is similar to using a try-finally block but provides a more concise and readable syntax.

Q28. `*args` and `**kwargs` are used in Python function definitions to allow for variable-length argument lists. Here's what they represent:

- `*args`: It allows a function to accept any number of positional arguments. The arguments passed are collected into a tuple.

- `**kwargs`: It allows a function to accept any number of keyword arguments. The arguments passed are collected into a dictionary.

Q29. To pass optional or keyword parameters from one function to another, you can use `*args` and `**kwargs` in the function definition and pass them along when calling another function. For example:

```python
def func1(*args, **kwargs):
    func2(*args, **kwargs)


def func2(*args, **kwargs):
    # Use the passed arguments and keyword arguments
    ...
```

Q30. Lambda functions, also known as anonymous functions, are small, one-line functions without a name. They are defined using the `lambda` keyword and can take any number of arguments but can only have a single expression as their body. Lambda functions are often used when a small function is needed for a short period of time and don't require a named function.

Q31. Inheritance in Python allows a class to inherit attributes and methods from another class

. It enables code reuse and promotes the concept of hierarchical relationships between classes. Here's an example:

```python
class Animal:
    def eat(self):
        print("Animal is eating.")


class Dog(Animal):
    def bark(self):
        print("Dog is barking.")


dog = Dog()
dog.eat()   # Inherits eat() from Animal
dog.bark()
```

In this example, the `Dog` class inherits the `eat()` method from the `Animal` class. The `Dog` class also defines its own `bark()` method.

Q32. When class `C` inherits from classes `A` and `B` as `class C(A, B)`, the version of the method `func()` that gets invoked depends on the method resolution order (MRO) of the classes. By default, Python follows the C3 linearization algorithm to determine the MRO. The MRO defines the order in which the base classes are searched for a method. In this case, the MRO will determine whether the version of `func()` from class `A` or class `B` is invoked.

Q33. The methods/functions used to determine the type of an instance and its inheritance are:

- `type(obj)`: Returns the type of the object `obj`. It can be used to determine the class from which the object is instantiated.

- `isinstance(obj, class)`: Returns `True` if `obj` is an instance of `class` or any of its derived classes. It can be used to check inheritance relationships.

Q34. The `nonlocal` keyword is used in Python to indicate that a variable is not local to the current function scope but also not a global variable. It is used inside a nested function to assign a value to a variable defined in the nearest enclosing scope that is not global. The `nonlocal` keyword allows you to modify variables in an outer, but non-global, scope.

Q35. The `global` keyword in Python is used to indicate that a variable defined within a function should be treated as a global variable, i.e., it should be accessible and modifiable from both inside and outside the function. It allows a function to access and modify global variables defined outside its local scope.