Q1. What are the two latest user-defined exception constraints in Python 3.X?

**Answer:** In Python, user-defined exceptions are typically derived from the built-in Exception class or its subclasses. By subclassing Exception, you can create custom exceptions that suit your specific needs. You can add custom attributes, methods, and behavior to these exceptions to handle specific error conditions in your code.

However, it's important to note that while there are no inherent constraints on user-defined exceptions, it is generally recommended to follow certain conventions and best practices when defining custom exceptions. Some of these practices include:

Inherit from Exception or its subclasses: It is recommended to derive your custom exceptions from Exception or one of its existing subclasses (e.g., ValueError, TypeError, etc.). This ensures that your exceptions are compatible with the existing exception hierarchy and can be caught and handled appropriately.

Provide informative error messages: It's good practice to include informative error messages in your custom exceptions. This helps users of your code understand the cause of the exception and provides helpful information for debugging and troubleshooting.

Consider naming conventions: Follow Python naming conventions when naming your custom exceptions. Typically, exception class names are in CamelCase, ending with the word "Error" (e.g., CustomError, InvalidInputError, etc.).

Document your exceptions: Provide documentation for your custom exceptions, including information about when and why they might be raised, the expected behavior when catching them, and any additional considerations or requirements.

Q2. How are class-based exceptions that have been raised matched to handlers?

**Answer:** In Python, when an exception is raised, the runtime system searches for an appropriate exception handler to handle the exception. The matching process is based on the exception class hierarchy and the order in which exception handlers are defined.

When an exception is raised, Python starts by looking for an exception handler within the current scope. If no suitable handler is found, it propagates the exception to the next outer scope and continues the search. This process continues until either an appropriate handler is found or the exception reaches the outermost scope, causing the program to terminate and display a traceback.

Q3. Describe two methods for attaching context information to exception artefacts.

**Answers:**

1.Exception Arguments:

One straightforward way to attach context information to an exception is by providing relevant details as arguments when raising the exception. When raising a built-in or custom exception, you can pass additional information as arguments to the exception constructor. These arguments can be accessed later when handling the exception, allowing you to retrieve and utilize the context information.

2.Custom Exception Classes:

Another approach is to define custom exception classes that inherit from the built-in exception classes or their subclasses. By creating your own exception classes, you can add custom attributes or properties to store the desired context information. These attributes can then be accessed when handling the exception.

Q4. Describe two methods for specifying the text of an exception object's error message.

**Answer:**

1.Passing a Message String:

The most straightforward method is to pass a message string as an argument when creating the exception object. You can provide a descriptive error message that explains the nature of the exception. This message string can be accessed later when handling the exception to display or log the error information.

2.Formatting Error Message:

Another method is to format the error message dynamically using the str.format() or f-string formatting techniques. This allows you to incorporate dynamic values, variables, or other context information into the error message. By utilizing the string formatting capabilities, you can create more informative and contextual error messages.

Q5. Why do you no longer use string-based exceptions?

**Answer:**

The main reason for moving away from string-based exceptions is to improve the clarity, consistency, and maintainability of the exception handling mechanism in Python.

1.Lack of Standardization

2.Limited Flexibility and Information

3.Reduced Error Identification

4.Improved Exception Hierarchy