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

Solution – In Python 3.x, the two latest user-defined exception constraints are:

Exception chaining with the raise ... from ... syntax:
Python 3.x introduced the ability to chain exceptions using the raise ... from ... syntax. This allows you to associate a new exception with an existing exception, preserving the original exception's traceback. It provides more informative error messages and helps in debugging.

Context-specific exception handling with the except ... as ... syntax:
Python 3.x introduced the except ... as ... syntax, which allows you to specify an exception variable that captures the raised exception instance. This provides more flexibility in handling different types of exceptions and accessing their attributes.

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

Solution - In Python, when a class-based exception is raised, it is matched to the appropriate exception handler based on the inheritance hierarchy of the exception classes. The exception handlers are defined using the except statement.

When an exception is raised, Python starts searching for an exception handler from the innermost try block and moves upwards through the call stack until it finds a matching handler. The matching is determined by checking if the raised exception class is a subclass of the exception class specified in the except statement or if it is the exact same class.

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

Solution - Here are two common methods for attaching context information to exception artifacts:

Exception Arguments:
When raising an exception, you can pass additional arguments to the exception constructor to provide context-specific information. These arguments can be accessed through the args attribute of the exception object.

Custom Exception Classes:
Another approach is to create custom exception classes that inherit from built-in exception classes or base exception classes. Custom exception classes allow you to define your own attributes and methods specific to the type of exception. You can include context information as attributes in the custom exception class and set them when raising the exception.

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

Solution - When raising an exception in Python, you can specify the text of the error message in different ways. Here are two common methods for specifying the text of an exception object's error message:

Using a String Argument:
The simplest method is to pass a string argument to the exception constructor when raising the exception. This string argument represents the error message associated with the exception.

Defining Custom Exception Classes:
Another approach is to define custom exception classes that inherit from built-in exception classes or base exception classes. By creating custom exception classes, you can provide specific error messages tailored to different types of exceptions.

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

Solution - Here are a few reasons why string-based exceptions have fallen out of favor:

Lack of Type Information: String-based exceptions do not provide any type information about the exception**.**

Limited Customization: String-based exceptions offer limited customization options compared to class-based exceptions.

Code Maintainability: Using class-based exceptions enhances code readability and maintainability.

Compatibility with Exception Hierarchy: Python's exception hierarchy is based on class inheritance, where built-in exceptions are organized into a hierarchy. By using class-based exceptions, your custom exceptions can easily integrate into this hierarchy, allowing for consistent exception handling and compatibility with existing exception-related features and libraries.