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

A)

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

Syntax for defining exceptions: In Python 3.x, you can define custom exceptions by creating a new class that inherits from the built-in Exception class or one of its subclasses. You can define your custom exception classes with a simple class statement, without any additional constraints on the class name or structure.

class MyCustomException(Exception):

pass

This creates a new exception class named MyCustomException that inherits from the built-in Exception class.

Exception chaining: Python 3.x introduced the ability to chain exceptions using the raise statement with the from keyword. This allows you to raise a new exception while preserving the original exception context, making it easier to trace the root cause of an error.

try:

# Some code that may raise an exception

except SomeException as e:

raise MyCustomException("An error occurred") from e

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

A) In Python, class-based exceptions that have been raised are matched to handlers based on their inheritance hierarchy. When an exception is raised, Python searches for an appropriate exception handler by looking at the exception type and then traversing up the inheritance chain until it finds a matching except block.

Here's how the matching process works:

Python starts by looking at the most recently raised exception.

It then searches for an except block that matches the exception type. If an exact match is found, the corresponding except block is executed.

If no exact match is found, Python looks for a handler that matches one of the exception's base classes. It continues searching up the inheritance hierarchy until it finds a matching handler.

If no matching handler is found at any level of the inheritance hierarchy, Python propagates the exception to the caller or the next level in the call stack.

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

A) In Python, there are several methods for attaching context information to exception artifacts to provide additional details when an exception occurs. Two common methods are:

Using Exception Arguments: When raising an exception, you can pass additional context information as arguments to the exception constructor. This allows you to provide specific details about the error that occurred.

try:

# Some code that may raise an exception

raise ValueError("Invalid input value: {}".format(user\_input))

except ValueError as e:

print("Error:", e)

In this example, the ValueError exception is raised with a message that includes the invalid input value. When the exception is caught, the error message provides context information about the nature of the error.

Using Traceback Objects: Python's traceback module provides functions for working with traceback objects, which contain information about the call stack at the point where an exception occurred. You can extract relevant information from the traceback object and include it in exception messages or logs.

import traceback

try:

# Some code that may raise an exception

raise ValueError("Invalid input value")

except ValueError:

tb\_info = traceback.format\_exc()

print("Traceback:", tb\_info)

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

A) In Python, there are multiple methods for specifying the text of an exception object's error message. Two common methods are:

Using Exception Constructor Arguments: When raising an exception, you can pass a string as an argument to the exception constructor to specify the error message. This allows you to provide a custom error message that describes the nature of the error.

try:

# Some code that may raise an exception

raise ValueError("Invalid input value")

except ValueError as e:

print("Error:", e)

In this example, the ValueError exception is raised with the error message "Invalid input value". When the exception is caught, the error message specified in the exception constructor is printed, providing information about the error that occurred.

Using Exception Subclasses: You can create custom exception classes by subclassing built-in exception classes like Exception and overriding the \_\_init\_\_ method to specify the error message. This allows you to define custom exception classes with predefined error messages that provide specific details about the error condition.

class CustomError(Exception):

def \_\_init\_\_(self, message):

self.message = message

try:

# Some code that may raise an exception

raise CustomError("Custom error message")

except CustomError as e:

print("Error:", e.message)

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

A) String-based exceptions were used in older versions of Python, but they have been deprecated in favor of using exception classes for several reasons:

Lack of Structured Information: String-based exceptions only provide a text message describing the error, without any structured information about the type or context of the error. This makes it harder to programmatically handle different types of errors and extract meaningful information from exceptions.

Limited Extensibility: With string-based exceptions, it's difficult to create custom exception types with additional attributes or behavior. Exception classes allow for more flexibility and extensibility by allowing you to define custom exception hierarchies with specialized behavior.

Type Safety: Using exception classes provides type safety, ensuring that exceptions are consistently raised and caught using the correct types. This helps prevent accidental errors and makes code easier to understand and maintain.

Better Error Handling: Exception classes allow for more precise error handling by enabling you to catch specific types of exceptions and handle them appropriately. This makes it easier to write robust error-handling code that can gracefully handle different types of errors.