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

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

1. \_\_cause\_\_ attribute: The \_\_cause\_\_ attribute is used to establish a causal relationship between exceptions. It allows one exception to reference another exception that caused it. This attribute helps in providing more detailed information about the reason for the exception.
2. \_\_context\_\_ attribute: The \_\_context\_\_ attribute is used to establish a contextual relationship between exceptions. It allows an exception to reference another exception that is being handled in a higher-level exception block. This attribute helps in preserving the original exception context while handling an exception.

These attributes were introduced to enhance the exception handling capabilities in Python, allowing exceptions to carry more information and establish relationships between them. They provide a way to propagate and preserve exception information, making it easier to debug and understand the flow of exceptions in a program.

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

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

When an exception is raised, Python checks the except statements in the current execution context (including the current function or method, and any enclosing try blocks) in a top-down order. It compares the raised exception against the specified exception classes in the except statements.

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

In Python, there are two common methods for attaching context information to exception artifacts: adding custom attributes to the exception object and using exception chaining.

1. Adding Custom Attributes to the Exception Object: You can attach custom attributes to the exception object to provide additional context information about the exception. This can be done by subclassing the built-in exception classes or creating custom exception classes.

Here's an example:

class CustomException(Exception):

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

super().\_\_init\_\_(message)

self.context = context

try:

# Some code that may raise an exception

raise CustomException("An error occurred", {"user\_id": 123, "action": "login"})

except CustomException as e:

print("Caught CustomException")

print("Context:", e.context)

1. Exception Chaining: Exception chaining allows you to associate a new exception with the original exception that caused it. This can be useful when you catch an exception, perform some additional processing or logging, and then raise a new exception to indicate the higher-level failure while preserving the original exception information.

Here's an example:

try:

# Some code that may raise an exception

raise ValueError("Invalid input")

except ValueError as e:

print("Caught ValueError")

# Perform additional processing or logging

raise RuntimeError("Unexpected error occurred") from e

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

In Python, there are two common methods for specifying the text of an exception object's error message:

1. Passing a String Argument to the Exception Class: The most straightforward way to specify the error message of an exception object is to pass a string argument to the exception class when raising the exception. This allows you to provide a custom error message that describes the specific nature of the exception.

Here's an example:

try:

# Some code that may raise an exception

raise ValueError("Invalid input")

except ValueError as e:

print("Caught ValueError")

print("Error message:", str(e))

1. Subclassing Exception Classes and Customizing the \_\_str\_\_ Method: Another method is to create a custom exception class by subclassing one of the built-in exception classes and customizing the \_\_str\_\_ method. The \_\_str\_\_ method is responsible for converting the exception object to a string representation, which is typically used as the error message.

Here's an example

class CustomException(Exception):

def \_\_str\_\_(self):

return "Custom error message"

try:

# Some code that may raise an exception

raise CustomException()

except CustomException as e:

print("Caught CustomException")

print("Error message:", str(e))

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

In Python, string-based exceptions were used in earlier versions (Python 2.x) to represent and handle exceptions by using strings as the exception type. However, this approach has been deprecated and is no longer recommended in Python 3.x. The primary reasons for moving away from string-based exceptions are:

1. Lack of Standardization: String-based exceptions lacked a standardized structure and behavior. Each developer could define their own custom exception strings, leading to inconsistency and difficulty in understanding and handling exceptions across different codebases.
2. Limited Functionality: String-based exceptions did not provide any built-in functionality or attributes for handling exceptions. They were treated as generic objects without any specific behavior or methods associated with them. This made it challenging to perform standard exception handling operations like retrieving error messages, traceback information, or distinguishing between different types of exceptions.
3. Difficulty in Handling and Debugging: String-based exceptions made it harder to catch and handle specific exceptions since there was no explicit exception class to identify and match against. Instead, developers had to rely on string comparisons or regular expressions to determine the type of exception, which could be error-prone and less reliable. Debugging and troubleshooting issues related to string-based exceptions were also more challenging due to the lack of standardized exception objects.