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

In Python 3.x, there are two new user-defined exception constraints:

The exception's \_\_cause\_\_ attribute: This attribute is used to specify a previous exception that caused the current exception to occur. This can be useful for debugging and tracing error chains.

The exception's \_\_context\_\_ attribute: This attribute is used to specify the context of the current exception. This is useful when a new exception is raised in response to a caught exception, allowing the new exception to retain information about the original exception.

Both of these attributes provide additional information about exceptions, allowing for better debugging and error handling in Python 3.x.

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

When a class-based exception is raised, Python searches for an appropriate exception handler to handle the exception. This search follows a hierarchical order and stops when an appropriate handler is found.

The search starts with the innermost block, where the exception is raised, and moves outward through the call stack until the outermost block is reached. Along the way, Python looks for handlers that match the exception type in the order in which they are defined.

If an appropriate handler is found, Python executes the code inside the handler block and then continues with the next statement after the try-except block. If no handler is found, Python terminates the program and prints a traceback that shows the exception information and the line numbers where the exception occurred.

It is important to note that the search for an appropriate exception handler only considers the class hierarchy of the exception, not its instance attributes. Therefore, it is possible for an exception to be caught by a handler that is not designed to handle it properly. To avoid this, it is important to design exception hierarchies carefully and to be specific in the exception types that are raised and handled.

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

Attaching context information to exception artifacts can help in debugging and understanding the cause of an exception. Here are two methods for attaching context information to exception artifacts:

1. Using the args attribute: The args attribute of an exception object is a tuple of arguments passed to the exception constructor. You can include additional context information by passing it as a separate argument in the args tuple. For example, consider the following code:

try:

# some code that may raise an exception

except ValueError as e:

Here, we catch the ValueError exception and re-raise it with an additional context message.

1. Subclassing exceptions: Another way to attach context information is to subclass an exception and add additional attributes to the subclass. For example:

class MyException(Exception):

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

self.message = message

self.context = context

try:

# some code that may raise an exception

except ValueError as e:

raise MyException("ValueError occurred", e.args[0])

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

In Python, there are several ways to specify the text of an exception object's error message:

1. Using the constructor of the exception class: When you create an instance of an exception class, you can pass a string as an argument to the constructor to provide a custom error message. For example:

class MyException(Exception):

pass

try:

raise MyException("This is a custom error message")

except MyException as e:

print(str(e)) # Output: This is a custom error message

1. Using string formatting: You can use string formatting to create a dynamic error message that includes the values of variables or other context information. For example:

try:

x = 42

y = 0

z = x / y

except ZeroDivisionError as e:

raise ValueError(f"Invalid value for y: {y}") from e

In this example, we catch a ZeroDivisionError exception and raise a ValueError exception with a custom error message that includes the value of y.

Both of these methods allow you to provide a custom error message that provides more information about the cause of the exception, which can be helpful when debugging your code.

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

In earlier versions of Python (2.x), it was possible to raise exceptions as strings using the raise statement. However, this approach has been deprecated and is no longer recommended.

The main reason for deprecating string-based exceptions is that they don't allow for the creation of a hierarchy of exception classes, which makes it difficult to catch and handle exceptions in a structured way. In contrast, the use of exception classes enables developers to define and organize exceptions in a more meaningful way, making it easier to catch and handle them appropriately.

Additionally, the use of string-based exceptions can lead to errors that are difficult to diagnose, as it is not always clear what type of exception is being raised and what information should be provided to handle it properly. Using exception classes provides a clear and consistent way to represent and handle errors, making code more reliable and easier to maintain.