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

A1. In Python 3.x, user-defined exceptions can be customized with constraints and attributes to provide more specific and informative error handling. The two latest or notable constraints and enhancements for user-defined exceptions include:

**1. Custom Exception Attributes**

Python allows you to add custom attributes to user-defined exceptions. This enhancement lets you attach additional information to your exceptions, which can be useful for debugging or providing more context about the error.

**Example**:

class CustomError(Exception):

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

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

self.code = code

try:

raise CustomError("Something went wrong", 404)

except CustomError as e:

print(f"Error: {e}, Code: {e.code}")

**Description**: In this example, CustomError has an additional code attribute. This allows you to pass and access more specific information about the exception.

**2. Exception Chaining**

Python 3.x supports exception chaining, which allows you to raise a new exception while preserving the original exception context. This feature is useful for adding context or wrapping exceptions while maintaining the traceback of the original error.

**Example**:

class BaseError(Exception):

pass

class CustomError(BaseError):

pass

try:

raise BaseError("Original error")

except BaseError as e:

raise CustomError("Wrapped error") from e

**Description**: In this example, CustomError is raised in response to the BaseError. Using from e preserves the original exception context, allowing you to see the complete traceback and understand the chain of events that led to the new exception.

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

A2. In Python, class-based exceptions that have been raised are matched to handlers based on the class hierarchy of the exception and the structure of the except blocks. Here's how this matching process works:

**1. Exception Hierarchy**

Exceptions in Python are objects derived from the base class Exception. When handling exceptions, the except blocks check for matches based on the type of the exception raised and the type specified in the except block. The matching is performed using the class hierarchy, where exceptions are matched by their type and by inheritance.

**2. Matching Process**

* **Exact Match**: If an exception is raised as an instance of a specific exception class, it is caught by an except block that specifies that exact class.

**Example**:

class MyError(Exception):

pass

try:

raise MyError("An error occurred")

except MyError:

print("Caught MyError")

**Outcome**: The MyError exception is caught by the except MyError block because it matches the exact type of the raised exception.

* **Inheritance Matching**: If an exception is raised as an instance of a class that inherits from another exception class, it can be caught by an except block that specifies the base class or any superclass in the hierarchy.

**Example**:

class BaseError(Exception):

pass

class DerivedError(BaseError):

pass

try:

raise DerivedError("An error occurred")

except BaseError:

print("Caught BaseError or its subclass")

**Outcome**: The DerivedError exception is caught by the except BaseError block because DerivedError inherits from BaseError.

* **Order of except Blocks**: Python processes except blocks in the order they appear. If multiple except blocks are present, the first block that matches the raised exception is executed. If the exception matches multiple blocks, only the first matching block is handled.

**Example**:

class A(Exception):

pass

class B(A):

pass

try:

raise B("An error occurred")

except B:

print("Caught B")

except A:

print("Caught A")

**Outcome**: The B exception is caught by the except B block because it is the first matching block.

* **Default Handler**: If no matching except block is found, the exception propagates up the call stack until it finds a matching handler or reaches the top level of the program, where it may terminate the program if unhandled.

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

A3. Attaching context information to exceptions helps in diagnosing issues more effectively by providing additional details about the circumstances of the error. Here are two methods for attaching context information to exception artifacts:

**1. Custom Exception Attributes**

You can define custom attributes in user-defined exceptions to attach additional context information. This approach allows you to include relevant details about the error directly in the exception object.

**Example**:

class CustomError(Exception):

def \_\_init\_\_(self, message, error\_code, context\_info):

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

self.error\_code = error\_code

self.context\_info = context\_info

try:

raise CustomError("An error occurred", 404, {"function": "process\_data", "line": 42})

except CustomError as e:

print(f"Error: {e}, Code: {e.error\_code}, Context: {e.context\_info}")

**Description**: In this example, CustomError has additional attributes error\_code and context\_info, which are used to provide more information about the error. When the exception is caught, you can access these attributes to understand the context in which the error occurred.

**2. Exception Chaining**

Exception chaining allows you to raise a new exception while preserving the context of the original exception. This is done using the from keyword, which helps to propagate the original exception’s traceback along with the new exception.

**Example**:

class BaseError(Exception):

pass

class DetailedError(BaseError):

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

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

self.details = details

try:

try:

# Code that raises an exception

raise ValueError("Original issue")

except ValueError as e:

raise DetailedError("Enhanced error with additional context") from e

except DetailedError as e:

print(f"Error: {e}, Details: {e.details}")

print(f"Original exception: {e.\_\_cause\_\_}")

**Description**: In this example, DetailedError is raised with additional context, and it chains the original ValueError using from e. The \_\_cause\_\_ attribute of the DetailedError object provides access to the original exception, helping to understand the sequence of events that led to the new exception.

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

A4. Specifying the text of an exception object's error message helps to provide clear and informative error reporting. Here are two common methods for specifying the text of an exception object's error message:

**1. Passing a Message String to the Exception Constructor**

When you create a custom exception class, you can pass a message string to the base Exception class or directly to your custom exception's constructor. This message string becomes the error message associated with the exception.

**Example**:

class CustomError(Exception):

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

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

try:

raise CustomError("An error occurred due to invalid input.")

except CustomError as e:

print(f"Error: {e}")

**Description**: In this example, the CustomError class takes a message parameter in its constructor. When raising the exception, the message string is passed to the base Exception class. The print(f"Error: {e}") statement outputs the message associated with the exception.

**2. Formatting the Error Message in the Exception's \_\_init\_\_ Method**

You can customize the exception's \_\_init\_\_ method to format the error message based on the input parameters or additional context. This allows you to construct a more detailed or context-specific error message.

**Example**:

class DetailedError(Exception):

def \_\_init\_\_(self, error\_code, context\_info):

self.error\_code = error\_code

self.context\_info = context\_info

self.message = f"Error code {error\_code}: {context\_info}"

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

try:

raise DetailedError(404, "Resource not found")

except DetailedError as e:

print(f"Error: {e.message}")

**Description**: In this example, DetailedError constructs a formatted message within its \_\_init\_\_ method based on error\_code and context\_info. The formatted message is then passed to the base Exception class. When the exception is caught, the message attribute provides the formatted error text.

**Summary**

1. **Passing a Message String to the Exception Constructor**: Provide a simple error message string to the exception's constructor. This message is used as the exception's error message and is accessible via the Exception class's default behavior.
2. **Formatting the Error Message in the Exception's \_\_init\_\_ Method**: Customize the \_\_init\_\_ method of the exception to construct a detailed or context-specific error message based on the parameters or additional context. This allows for more informative and tailored error messages.

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

A5. String-based exceptions (i.e., raising exceptions using string messages rather than instance objects) were used in older versions of Python but are no longer recommended for several reasons:

**1. Lack of Extensibility and Information**

String-based exceptions provide only a simple message without any additional context or structured information. This limits the ability to add extra data or attributes that could be useful for debugging and error handling.

**Example**:

raise "An error occurred"

**Issue**: This raises a TypeError because the raise statement expects an exception instance, not a string. Using strings in this way was never standard and has been replaced by instance-based exceptions.

**2. Improved Exception Handling**

Using class-based exceptions allows you to create custom exception types that can carry additional information and context. This approach is more flexible and provides better error handling by allowing for richer exception objects that can be extended with custom attributes and methods.

**Example**:

class CustomError(Exception):

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

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

self.code = code

raise CustomError("An error occurred", 404)

**Benefit**: This allows the exception to carry not just a message but also additional data (e.g., error codes, context information), which can be accessed when handling the exception.

**3. Consistent Exception Hierarchy**

Python’s exception hierarchy is based on class inheritance from the base Exception class. Using class-based exceptions ensures consistency with Python’s exception handling mechanisms and allows for better integration with existing exception handling practices.

**Example**:

try:

# Code that raises an exception

pass

except CustomError as e:

print(f"Caught exception: {e.message}, Code: {e.code}")

**Benefit**: This fits well within Python's standard practices for exception handling, enabling more granular control and better integration with the language’s exception handling system.

**4. Better Debugging and Traceback Information**

Class-based exceptions provide richer traceback information and allow for more detailed inspection of what went wrong. This helps in debugging and understanding the context in which the exception occurred.

**Example**:

try:

# Code that might raise an exception

pass

except CustomError as e:

print(f"Exception occurred: {e}")

print(f"Traceback: {e.\_\_traceback\_\_}")

**Benefit**: Provides a full traceback of where the exception was raised and propagated, aiding in debugging and error analysis.