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

ANS: As of my knowledge cutoff date in September 2021, Python 3.x does not impose any specific constraints on user-defined exceptions beyond the general requirements for exception classes. However, there are two commonly recommended practices or constraints that developers typically follow when defining custom exceptions:

1. Inherit from Built-in Exception Classes: When defining custom exceptions, it is recommended to inherit from one of the built-in exception classes provided by Python's standard library. The built-in exception classes provide useful attributes and behaviors that are inherited by your custom exception, making it easier to handle and identify specific types of exceptions in your code.

Example of a custom exception inheriting from a built-in exception class:

class CustomError(Exception):

pass

class CustomValueError(ValueError):

pass

1. End Exception Class Names with "Error": Following a common naming convention, it is often suggested to end the names of custom exception classes with the word "Error" to indicate that they represent exceptional conditions or errors.

Example:

class MyCustomError(Exception):

pass

class InvalidDataError(ValueError):

pass

By following these practices, you can create more consistent and easily recognizable custom exceptions that align with Python's exception hierarchy and make it clear that they represent exceptional situations in your code.

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

ANS: When a class-based exception is raised in Python, the matching of the exception to its appropriate handler is determined by the exception's inheritance hierarchy and the order of the `except` blocks in the code. Python searches for the first `except` block that can handle the raised exception, based on the exception's type and the order of inheritance.

The process of matching class-based exceptions to handlers follows these steps:

a). When an exception is raised, Python starts searching for an appropriate `except` block to handle the exception in the current function's scope.

b). Python first checks the `except` blocks from top to bottom. It compares the type of the raised exception with the types specified in the `except` blocks.

c). Python matches the exception type against the types specified in the `except` blocks using the exception's inheritance hierarchy. If the raised exception is an instance of the specified type or a subclass of the specified type, the `except` block is considered a match, and its associated code block is executed.

d). If Python finds a matching `except` block, it executes the code in that block and then continues with the rest of the program's flow after the `try-except` statement.

e). If Python doesn't find any matching `except` block in the current function's scope, it propagates up the call stack, searching for enclosing `try-except` blocks in outer scopes (in the calling functions). The process continues until it finds a suitable `except` block to handle the exception, or the program terminates if no matching handler is found.

Here's an example of how class-based exceptions are matched to handlers:

class MyCustomError(Exception):

pass

class MySpecificError(MyCustomError):

pass

try:

raise MySpecificError("Something went wrong.")

except MySpecificError:

print("Handled MySpecificError.")

except MyCustomError:

print("Handled MyCustomError.")

except Exception:

print("Handled Exception.")

Output:

Handled MySpecificError.

In this example, we have a custom exception hierarchy with `MyCustomError` as the base class and `MySpecificError` as its subclass. When we raise `MySpecificError`, Python first checks for a matching `except` block for `MySpecificError`. Since there is a matching `except MySpecificError`, it executes the associated code block and prints "Handled MySpecificError."

If the order of the `except` blocks were reversed, i.e., `except MyCustomError` before `except MySpecificError`, Python would match the `except MyCustomError` block instead, as `MySpecificError` is a subclass of `MyCustomError`. The order of the `except` blocks matters when dealing with class-based exceptions, as Python will execute the first matching block it encounters during the search process.

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

ANS: When dealing with exceptions, it is often essential to attach context information to the exception artifacts to provide more meaningful error messages, debugging details, and insights into the cause of the exception. Python provides several methods to achieve this. Here are two common methods for attaching context information to exception artifacts:

A). Using the `raise` statement with `from`: When raising an exception, you can use the `from` clause to attach another exception as the context for the current exception. This allows you to create a new exception while preserving the original exception's context. The `from` clause helps to establish a chain of exceptions, providing more context about what caused the exception.

def divide(a, b):

try:

result = a / b

return result

except ZeroDivisionError as e:

raise ValueError("Cannot divide by zero.") from e

try:

result = divide(10, 0)

except ValueError as e:

print(e) # Output: Cannot divide by zero.

print(e.\_\_cause\_\_) # Output: division by zero

In this example, the `divide()` function raises a `ValueError` with the message "Cannot divide by zero" when a zero divisor is encountered. The `from e` clause attaches the original `ZeroDivisionError` as the cause of the new `ValueError`, providing additional context about what caused the error.

B). Adding Custom Attributes to Exceptions: You can subclass built-in exception classes or create custom exception classes to add custom attributes that carry additional context information about the exception. By adding custom attributes, you can include relevant data that helps understand the exceptional situation better.

class MyCustomError(Exception):

def \_\_init\_\_(self, message, context\_info=None):

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

self.context\_info = context\_info

try:

raise MyCustomError("An error occurred.", context\_info="Additional details.")

except MyCustomError as e:

print(e) # Output: An error occurred.

print(e.context\_info) # Output: Additional details.

In this example, we've created a custom exception class `MyCustomError`, which includes an additional `context\_info` attribute. When raising the exception, we provide the context information as an argument, and it becomes accessible via the `context\_info` attribute of the exception instance.

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

ANS: In Python, you can specify the text of an exception object's error message using two methods: by passing an argument to the exception class during exception creation and by overriding the `\_\_str\_\_()` or `\_\_repr\_\_()` method in the custom exception class.

1. Passing an Argument to the Exception Class:

When raising a built-in or custom exception, you can pass a string argument to the exception class constructor, which will be used as the error message for the exception object. The constructor of the exception class can accept additional arguments, and you can use them to provide contextual information about the exceptional situation.

Example using a built-in exception:

try:

age = -5

if age < 0:

raise ValueError("Age cannot be negative.")

except ValueError as e:

print(e) # Output: Age cannot be negative.

Example using a custom exception:

class MyCustomError(Exception):

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

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

try:

raise MyCustomError("An error occurred.")

except MyCustomError as e:

print(e) # Output: An error occurred.

b) Overriding the `\_\_str\_\_()` or `\_\_repr\_\_()` Method in Custom Exception Class:

When creating a custom exception class, you can override the `\_\_str\_\_()` or `\_\_repr\_\_()` method to customize the error message returned by the exception when it is printed or converted to a string. These methods allow you to control the string representation of the exception object.

Example using `\_\_str\_\_()`:

class MyCustomError(Exception):

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

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

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

return f"MyCustomError: {self.args[0]}"

try:

raise MyCustomError("An error occurred.")

except MyCustomError as e:

print(e) # Output: MyCustomError: An error occurred.

Example using `\_\_repr\_\_()`:

class MyCustomError(Exception):

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

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

def \_\_repr\_\_(self):

return f"MyCustomError('{self.args[0]}')"

try:

raise MyCustomError("An error occurred.")

except MyCustomError as e:

print(repr(e)) # Output: MyCustomError('An error occurred.')

By using these methods, you can customize the error messages associated with your exception objects. Providing clear and informative error messages helps in understanding the exceptional situations that occur in your code and makes debugging and error handling more effective.

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

ANS: As an AI language model, I don't have personal preferences or behaviors, so I don't "use" or "not use" string-based exceptions. However, I can provide insights into the reasons why string-based exceptions are discouraged and less common in modern Python programming practices.

This approach was not recommended for several reasons:

1. Less Descriptive and Error-Prone: Using string-based exceptions provides less descriptive error messages, making it harder to understand the cause of the exception. Moreover, since exceptions are represented as simple strings, there is no clear distinction between different types of exceptions, and it becomes error-prone to handle specific exception cases.
2. No Inheritance Hierarchy: With string-based exceptions, there is no hierarchy or inheritance structure, which makes it difficult to organize and handle exceptions in a structured manner. Exception hierarchy helps in organizing related exceptions and allows more specific exception handling.
3. Lack of Compatibility and Future-Proofing: Python developers have been moving towards using class-based exceptions since Python 2.5. As Python evolved, string-based exceptions were deprecated and eventually removed in Python 3.x. Relying on string-based exceptions could result in compatibility issues and may not be future-proof.

Instead of using string-based exceptions, Python encourages using class-based exceptions, where you define custom exception classes that inherit from the built-in `BaseException` or its subclasses (`Exception`, `StandardError`, etc.). Class-based exceptions provide several advantages, including:

- Clear distinction between different types of exceptions.

- Exception hierarchy for organizing and handling exceptions in a structured way.

- Custom attributes to store additional information about the exception.

- Better support for exception handling and debugging.

Here's an example of using class-based exceptions in Python:

class MyCustomError(Exception):

pass

try:

raise MyCustomError("Something went wrong.")

except MyCustomError as e:

print("Caught the error:", e)

In conclusion, class-based exceptions are the recommended and standard approach in modern Python programming due to their advantages over string-based exceptions. They provide more structured and informative exception handling, leading to more robust and maintainable code.