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

Answer :- In Python 3.x, there are two notable constraints related to user-defined exceptions that were introduced to ensure consistency and clarity in exception handling. These constraints affect how custom exceptions are defined and used:

1. **Inheriting from** BaseException: In Python 3.x, all user-defined exceptions must inherit from BaseException or one of its subclasses, typically Exception. This constraint ensures that custom exceptions integrate properly with Python's exception hierarchy and can be caught using standard exception-handling mechanisms.

# Correct way to define a custom exception

class MyCustomError(Exception):

pass

# Using the custom exception

try:

raise MyCustomError("Something went wrong!")

except MyCustomError as e:

print(e)

 If you attempt to define an exception that does not inherit from BaseException or its subclasses, Python will raise a TypeError.

 Defining the \_\_cause\_\_ and \_\_context\_\_ Attributes: When raising exceptions, especially when chaining exceptions (one exception leading to another), Python 3.x has improved mechanisms to link these exceptions through the \_\_cause\_\_ and \_\_context\_\_ attributes. These attributes help in understanding the context in which the exception occurred.

* \_\_cause\_\_: This attribute is explicitly set when an exception is raised from within an except block using the raise from syntax. It indicates that the new exception was directly caused by the original exception.
* \_\_context\_\_: This attribute is automatically set when an exception occurs within an except block but without using raise from. It indicates the exception that was active at the time the new exception was raised.

Example of using raise from:

class CustomError(Exception):

pass

try:

1 / 0

except ZeroDivisionError as e:

raise CustomError("Custom error caused by division by zero") from e

1. In this example:
   * The CustomError is raised from within the except block.
   * The from e syntax links the CustomError to the original ZeroDivisionError, setting the \_\_cause\_\_ attribute.

By adhering to these constraints, user-defined exceptions in Python 3.x are more consistent with the language's overall design, improving error handling and debugging capabilities.

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

Answer :- In Python, class-based exceptions that have been raised are matched to handlers using the inheritance hierarchy of exception classes. When an exception is raised, Python searches for an appropriate exception handler in the sequence of except clauses, starting with the innermost block and moving outward. Here's how the matching process works:

**Exact Match**: Python first looks for an except clause that exactly matches the type of the raised exception.

**Subclass Match**: If no exact match is found, Python looks for handlers that match any of the exception's parent classes (superclasses) in the inheritance hierarchy. This allows a handler to catch exceptions that are subclasses of the specified exception type.

**BaseException Match**: If no match is found among the specific exception types and their superclasses, a generic except BaseException clause can catch any exception, as all exception classes ultimately inherit from BaseException.

Example

Here's a demonstration of how class-based exceptions are matched to handlers:

class MyBaseError(Exception):

pass

class MySubError(MyBaseError):

pass

class AnotherError(Exception):

pass

try:

raise MySubError("This is a subclass error.")

except MySubError:

print("Caught MySubError.")

except MyBaseError:

print("Caught MyBaseError.")

except Exception:

print("Caught a generic exception.")

# Output: Caught MySubError.

In this example:

MySubError is raised.

Python first checks the except MySubError clause and finds a match, so it executes that block.

If the except MySubError clause were not present, Python would match MySubError to the except MyBaseError clause because MySubError is a subclass of MyBaseError.

Handling Multiple Exceptions

Python also allows a single except clause to handle multiple exception types by specifying a tuple of exceptions:

try:

raise MySubError("This is a subclass error.")

except (MySubError, AnotherError):

print("Caught either MySubError or AnotherError.")

except MyBaseError:

print("Caught MyBaseError.")

except Exception:

print("Caught a generic exception.")

# Output: Caught either MySubError or AnotherError.

In this example:

MySubError is raised.

The except (MySubError, AnotherError) clause matches and handles the exception.

Summary

When an exception is raised, Python matches it to handlers based on the inheritance hierarchy of the exception classes. The matching process begins with an exact match, then checks for subclass matches, and finally falls back to a generic handler if no specific match is found. This mechanism allows for flexible and precise exception handling in Python programs.

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

Answer :- Attaching context information to exception artifacts is crucial for debugging and understanding the circumstances under which an error occurred. In Python, there are two primary methods to achieve this: using exception chaining (\_\_cause\_\_ and \_\_context\_\_ attributes) and adding custom attributes or messages to exception objects.

1. Exception Chaining (\_\_cause\_\_ and \_\_context\_\_)

Python's exception chaining mechanism allows you to attach context information by linking exceptions that are related. This is done using the \_\_cause\_\_ and \_\_context\_\_ attributes, which provide a way to trace the sequence of exceptions.

**Using** raise from **to Set** \_\_cause\_\_**:** The raise from syntax is used to explicitly chain exceptions, setting the \_\_cause\_\_ attribute. This indicates that one exception was directly caused by another.

try:

1 / 0

except ZeroDivisionError as e:

raise ValueError("A value error occurred due to a division by zero.") from e

 In this example:

A ZeroDivisionError is raised and caught.

A ValueError is then raised, using from e to link it to the original ZeroDivisionError.

The ValueError's \_\_cause\_\_ attribute is set to the ZeroDivisionError, making the chain explicit.

 Automatic Setting of \_\_context\_\_: When an exception is raised inside an except block without using raise from, the new exception's \_\_context\_\_ attribute is automatically set to the current exception.

try:

try:

1 / 0

except ZeroDivisionError:

int('invalid')

except ValueError as e:

print(f"Caught a ValueError: {e}")

print(f"Original context: {e.\_\_context\_\_}")

In this example:

A ZeroDivisionError is raised and caught.

While handling the ZeroDivisionError, a ValueError is raised.

The ValueError's \_\_context\_\_ attribute is automatically set to the ZeroDivisionError.

2. Adding Custom Attributes or Messages

Another method to attach context information is by adding custom attributes or messages to exception objects. This can be done by extending exception classes and defining additional attributes, or by including detailed messages when raising exceptions.

**Extending Exception Classes:**

class CustomError(Exception):

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

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

self.custom\_data = custom\_data

try:

raise CustomError("An error occurred", custom\_data={"key": "value"})

except CustomError as e:

print(f"Error: {e}")

print(f"Custom Data: {e.custom\_data}")

 In this example:

A CustomError class is defined with an additional custom\_data attribute.

When CustomError is raised, it includes both a message and custom data.

The custom data can be accessed when handling the exception.

 Including Detailed Messages:

try:

raise ValueError("An error occurred. Additional context: the input value was invalid.")

except ValueError as e:

print(e)

In this example:

A ValueError is raised with a detailed message that includes context information.

The message provides additional context for understanding the error.

By using exception chaining and adding custom attributes or messages, you can provide rich context information in your exception artifacts, making it easier to debug and understand the root causes of errors

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

Answer :- In Python, you can specify the text of an exception object's error message in two primary ways:

**Using the Exception's Constructor**: When you raise an exception, you can pass the error message as an argument to the exception's constructor. This is the most straightforward and common way to specify an error message.

try:

raise ValueError("This is a custom error message.")

except ValueError as e:

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

 In this example:

The ValueError is raised with a custom error message.

The message is printed when the exception is caught.

 Defining a Custom Exception Class with Custom Attributes: You can define a custom exception class that includes additional attributes for storing context-specific information. This approach allows you to pass more detailed information and format the error message as needed.

class CustomError(Exception):

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

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

self.error\_code = error\_code

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

return f"{self.args[0]} (Error Code: {self.error\_code})"

try:

raise CustomError("A custom error occurred", 404)

except CustomError as e:

print(e) # Output: A custom error occurred (Error Code: 404)

In this example:

A CustomError class is defined with an additional error\_code attribute.

The \_\_str\_\_ method is overridden to include the error code in the error message.

When the exception is raised, the custom message along with the error code is printed.

Summary

**Using the Exception's Constructor**: Pass the error message directly when raising the exception. This method is simple and effective for most cases.

**Defining a Custom Exception Class with Custom Attributes**: Define additional attributes and override the \_\_str\_\_ method to format the error message, allowing for more detailed and context-specific information.

These methods provide flexibility in how you present and handle error messages, making it easier to understand and debug exceptions in your code.

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

Answer :- String-based exceptions, where exceptions are raised using strings as their primary form of error reporting, are no longer recommended for several reasons:

**Lack of Type Information**: String-based exceptions do not provide any type information, which makes it difficult to handle different kinds of errors appropriately. Exception handling relies on the ability to differentiate between different types of exceptions, and string-based exceptions do not support this.

try:

raise "An error occurred"

except "An error occurred":

print("Caught an error")

 In this example, it's impossible to handle different types of errors or differentiate between them because all exceptions are treated as generic strings.

 **No Stack Trace**: Exception objects in Python carry a stack trace that helps developers understand where the error occurred. String-based exceptions do not include stack trace information, making debugging more challenging.

try:

raise ValueError("An error occurred")

except ValueError as e:

print(e) # Output: An error occurred

print(e.\_\_traceback\_\_) # Shows traceback information

 This capability is lost with string-based exceptions, as they do not provide traceback details.

 **No Customization**: Exception classes allow for customization, such as defining custom attributes or methods. This flexibility helps in attaching additional context or information to exceptions, which is essential for effective error handling.

class CustomError(Exception):

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

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

self.error\_code = error\_code

try:

raise CustomError("An error occurred", 404)

except CustomError as e:

print(e) # Output: An error occurred

print(e.error\_code) # Output: 404

 String-based exceptions lack this level of customization and extensibility.

 **Consistency with Python's Exception Hierarchy**: Python's exception handling system is designed around class-based exceptions. The standard library and many third-party libraries rely on this design. Using string-based exceptions is inconsistent with the language's design and best practices.

 **Error Handling and Specificity**: Using class-based exceptions allows for more specific error handling. You can catch specific exceptions or groups of exceptions and handle them accordingly, which is not possible with strings.

try:

raise ValueError("A specific error message")

except ValueError:

print("Caught a ValueError")

In this example, catching a specific exception type is straightforward and precise.

In summary, class-based exceptions provide a more robust, informative, and flexible mechanism for error handling compared to string-based exceptions. They integrate better with Python's exception hierarchy and offer the ability to include additional context, which is crucial for effective debugging and error management.