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

## Answer :

**<u>The two key concepts related to user-defined exceptions include</u>**:

**1.Exception Inheritance**

When creating a user-defined exception in Python, it must inherit from the built-in Exception class (or one of its subclasses). This is a general constraint to ensure that custom exceptions fit into the exception hierarchy and are treated appropriately by the Python interpreter.

**<u>Code Below :</u>**

```
class MyCustomError(Exception):
    """Custom exception that inherits from Exception"""
    Pass
```

**2. __str__ and __repr__ Customization**

When defining custom exceptions, it’s common to override the __str__ or __repr__ methods to provide more informative error messages. While this is not mandatory, it is considered good practice and a recommended constraint for better debugging and error reporting.

**<u>Code Below :</u>**

```
class MyCustomError(Exception):
    def __init__(self, message):
        self.message = message

    def __str__(self):
        return f"MyCustomError occurred: {self.message}"
```

These two aspects – inheritance from Exception and overriding special methods for better error messages – are critical when defining exceptions in Python 3.x.

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

## Answer :

The class-based exceptions are matched to handlers using the **exception hierarchy** and **try-except blocks**. When an exception is raised, Python looks for the first matching except clause by checking the type (class) of the raised exception and comparing it to the exception classes specified in the except clauses.

<u>**Here's how it works**</u>:

**1. Exact Class Match**

If the raised exception exactly matches the class specified in an except block, that block is executed.

<u>**Code Below :**</u>

```
class CustomError(Exception):
   pass

try:
   raise CustomError("An error occurred")
except CustomError:
   print("Caught CustomError")
```

<u>**Output:**</u>

```
Caught CustomError
```

**2. Subclass Match (Inheritance Hierarchy)**

If the raised exception is a subclass of the exception specified in an except block, it will also be caught. This is because Python matches exceptions based on the inheritance hierarchy.

<u>**Code Below :**</u>

```
class CustomError(Exception):
   pass

class SpecificError(CustomError):
   pass

try:
   raise SpecificError("A specific error occurred")
except CustomError:
    print("Caught a subclass of CustomError")
```

**Output:**

```
Caught a subclass of CustomError
```

Even though SpecificError was raised, it is a subclass of CustomError, so the CustomError handler catches it.

**3. Generic Exception Match**

If no more specific handlers are found, the general Exception class can be used as a "catch-all" for any user-defined or built-in exceptions.

**Code Below :**

```
try:
    raise ValueError("A value error occurred")
except Exception:
    print("Caught a general exception")
```

**4. Order of Exception Handlers**

Python matches handlers in the order they appear. It’s important to place more specific exceptions (subclasses) before general ones, like Exception. Otherwise, more general handlers will catch exceptions before the specific ones.

**Code Below :**

```
try:
    raise ValueError("A value error occurred")
except Exception:
    print("Caught by general Exception")
except ValueError:
    print("Caught ValueError")
```

**Output:**

```
Caught by general Exception
```

In this case, Exception is caught before ValueError even though ValueError is more specific, so it’s important to order them correctly:

**Code Below :**

```
try:
    raise ValueError("A value error occurred")
except ValueError:
    print("Caught ValueError")
except Exception:
    print("Caught by general Exception")
```

**Summary of Matching Process:**

1. Python checks the raised exception class.
2. It compares it with the exception types in the except clauses.
3. It starts with the most specific and moves to more general types, matching by inheritance.
4. The first match is executed, and any later handlers are ignored.

This process ensures that exceptions are handled in an organized manner, depending on their class and position in the inheritance hierarchy.

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

## Answer :

We can attach context information to exceptions to provide more useful details when errors occur. This can be done through the following two primary methods:

**1.Custom Attributes in User-Defined Exceptions**

- We can define custom attributes within your user-defined exception class to store relevant context information (e.g., error codes, file names, or other details).
- When the exception is raised, you can pass these details, making them accessible when the exception is caught.

**How it works:**

- Define a custom exception class that extends the built-in Exception class.
- Add an __init__ method to accept and store additional context information.

**Code Below:**

```
class FileProcessingError(Exception):
   def __init__(self, filename, error_code, message="Error processing file"):
      self.filename = filename
      self.error_code = error_code
      super().__init__(message)

try:
   raise FileProcessingError("data.csv", 404, "File not found")
except FileProcessingError as e:
   print(f"Error: {e.message}, Filename: {e.filename}, Error Code: {e.error_code}")
```

**Output:**

```
Error: File not found, Filename: data.csv, Error Code: 404
```

In this example, the FileProcessingError exception carries additional context about the filename and error code, which can be used in error handling.

## 2. Using the raise ... from Syntax for Exception Chaining

- The raise ... from syntax in Python allows you to raise a new exception while preserving the context of an original exception. This is useful for **exception chaining**, where a lower-level error leads to a higher-level exception being raised.

- The original exception is attached to the new one as its __cause__ attribute, making it easy to trace the root cause of the error.

**<u>How it works</u>:**

We catch an exception and raise a new, more meaningful exception while using raise ... from to maintain the original context.

**<u>Code Below</u>:**

```
try:
    data = open('missing_file.txt')
except FileNotFoundError as e:
    raise RuntimeError("Failed to process file") from e
```

**<u>Output:</u>**

```
Traceback (most recent call last):
  File "example.py", line 2, in <module>
    data = open('missing_file.txt')
FileNotFoundError: [Errno 2] No such file or directory: 'missing_file.txt'

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "example.py", line 4, in <module>
    raise RuntimeError("Failed to process file") from e
RuntimeError: Failed to process file
```

In this example, the FileNotFoundError is caught and a new RuntimeError is raised, but the original exception is preserved, providing full context.

**Summary of Methods:**

1. **Custom Attributes:** Attach custom information (e.g., error codes, file names) to user-defined exceptions by defining custom attributes.

2. **Exception Chaining (raise ... from):** Attach the original exception to a new one, preserving context and making it easier to trace the root cause of an error.

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

## Answer :

We can specify the error message for an exception object in the following two common ways:

**1.Passing the Error Message to the Exception Constructor**

When raising an exception, you can pass an error message as an argument to the exception's constructor. This message is stored as part of the exception object and can be accessed later (for example, when the exception is caught and handled).

**How it works:**

- The error message is passed as a string argument when raising the exception.
- The default __str__ method of the exception class returns the message when the exception is printed.

**Code Below:**

```
try:
   raise ValueError("Invalid input provided")
except ValueError as e:
   print(e)  # Outputs the error message: "Invalid input provided"
```

**Output:**

```
Invalid input provided
```

In this case, the error message "Invalid input provided" is passed to the ValueError exception when it is raised.

**2.Customizing the __str__ Method in User-Defined Exceptions**

- When defining a user-defined exception, you can override the __str__ method to customize how the exception's message is displayed. This allows you to dynamically format or generate the error message based on specific attributes of the exception object.
- The __str__ method controls how the exception is converted to a string when printed or logged.

**How it works:**

- Define a custom exception class that inherits from Exception (or any built-in exception class).
- Override the __str__ method to format the error message using attributes of the exception object.

**Code Below:**

```
class MyCustomError(Exception):
   def __init__(self, value):
      self.value = value

   def __str__(self):
      return f"MyCustomError: An error occurred with value {self.value}"

try:
   raise MyCustomError(42)
except MyCustomError as e:
   print(e)  # Outputs: "MyCustomError: An error occurred with value 42"
```

**Output:**

```
MyCustomError: An error occurred with value 42
```

In this example, the custom __str__ method returns a formatted message that incorporates the value attribute of the exception object.

**Summary of Methods:**

1. **Passing a Message to the Constructor:** Provide the error message when raising the exception, which can later be accessed through the default __str__ method.
2. **Customizing the __str__ Method:** Define a custom exception class and override the __str__ method to dynamically generate a custom error message based on the exception's attributes.

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

## Answer :

String-based exceptions were deprecated and eventually removed in Python because they introduced several limitations and problems that class-based exceptions addressed.

The key reasons for no longer using string-based exceptions are:

**1.Lack of Flexibility and Consistency**

- String-based exceptions rely solely on matching the exception message, which is prone to errors. For example, slight differences in strings (such as typos or variations in formatting) could result in exceptions not being correctly identified or handled.
- With class-based exceptions, the hierarchy of exception types allows for more flexible and structured handling. You can catch exceptions based on their type, which is more robust and predictable.

**Example of string-based exception issue:**

**Code Below :**

```
try:
   raise "File not found"
except "File not found":  # This could fail due to slight variations in the string
   print("Caught exception")
```

With class-based exceptions:

**Code Below :**

```
try:
   raise FileNotFoundError("File not found")
except FileNotFoundError:
   print("Caught FileNotFoundError")
```

**2.No Inheritance and Categorization**

- String-based exceptions do not support inheritance, meaning you cannot create a hierarchy of related exceptions (e.g., catching all I/O-related errors under a single IOError class).
- Class-based exceptions allow you to categorize exceptions into more general or specific types, making it easier to handle related errors in a more structured way.

**Example of class-based hierarchy:**

**Code Below :**

```
class CustomError(Exception):
   pass
```

```
class FileNotFoundError(CustomError):
    pass

try:
    raise FileNotFoundError("File not found")
except CustomError:  # Can catch both specific and general exceptions
    print("Caught a custom error")
```

**3.Type Safety and Attribute Support**

- String-based exceptions are just strings, which provide no type information. As a result, they cannot be differentiated beyond their content.

- Class-based exceptions allow you to define attributes and methods that store additional context about the error (e.g., file names, error codes), providing much more detailed error information.

**Code Below:**

```
class CustomError(Exception):
    def __init__(self, filename):
        self.filename = filename
        super().__init__(f"Error with file: {filename}")

try:
    raise CustomError("data.csv")
except CustomError as e:
    print(f"Error occurred with file: {e.filename}")
```

**4.Better Integration with Python’s Exception Handling Mechanism**

- Python's exception handling mechanism (e.g., try, except, finally) is built around class-based exceptions, which integrate smoothly with the language’s features like inheritance, exception chaining (raise ... from), and custom attributes.

- String-based exceptions do not have this level of integration and flexibility, leading to inconsistent and less maintainable code.

## Conclusion:

Class-based exceptions provide more flexibility, consistency, and robustness compared to string-based exceptions. They support inheritance, can carry additional context, and integrate seamlessly with Python's exception handling system, which is why string-based exceptions are no longer used in modern Python.