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

**Answer:-**

1. Exception Chaining: This constraint allows user-defined exceptions to capture and retain the original exception information. When raising a new exception within an exception handler, you can include the original exception as the cause of the new exception using the from keyword. This preserves the traceback and context of the original exception, providing more detailed information for debugging and error analysis. For example:

**try:**

**# Code that may raise an exception**

**# ...**

**except Exception as e:**

**raise NewException("Something went wrong") from e**

In this example, the NewException is raised with the original exception e as the cause. This allows for chaining exceptions and preserving the original exception information.

1. Exception Context Variables: This constraint allows user-defined exceptions to store additional context information as attributes. You can define custom attributes within your exception class to provide relevant details about the exception. These context variables can be accessed when handling the exception, providing more specific information about the error condition. For example:

class MyException(Exception):

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

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

self.error\_code = error\_code

try:

# Code that may raise an exception

# ...

except MyException as e:

print("Error:", e)

print("Error Code:", e.error\_code)

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

**Answer:-**

Here's the process of matching a class-based exception to its handler:

1. The exception is raised using the raise statement or by a built-in function/method.
2. The interpreter searches for an appropriate except block that can handle the exception. It starts with the first except block defined in the code and proceeds sequentially.
3. The MRO of the exception class is followed to determine the inheritance order. The MRO defines the order in which the base classes are checked for method or attribute resolution.
4. The interpreter compares the raised exception with each except block in the order they are defined. If the raised exception is an instance of the exception type specified in the except block or is a subclass thereof, the corresponding block is considered a match.
5. Once a matching except block is found, the code within that block is executed, and the program continues from the next statement after the except block. If no matching except block is found, the exception propagates up the call stack to the next higher-level except block or, if not caught, results in program termination with an unhandled exception.

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

**Answer:-**

1. Exception attributes: Additional information can be attached to exception objects as attributes to provide context.
2. Example: exception\_object.context = "Additional context"
3. Custom exception classes: Define your own exception classes with custom attributes and methods to store and retrieve context information.
4. Example: class MyException(Exception): def \_\_init\_\_(self, message, context): self.message = message self.context = context
5. Attach context during exception creation: Pass context information as arguments when creating an exception instance.
6. Example: raise MyException("Error occurred", "Context information")
7. Accessing context: When handling the exception, access the attached context information from the exception object.
8. Example: except MyException as e: print("Exception:", str(e)) print("Context:", e.context)
9. Custom methods: Define methods within custom exception classes to retrieve specific context information.
10. Example: class MyException(Exception): def get\_context(self): return self.context

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

**Answer:-**

1. Custom Exception Classes: Define a custom exception class and override the \_\_init\_\_ method to set the error message.
2. Example: class MyException(Exception): def \_\_init\_\_(self, message): self.message = message
3. Raise the custom exception: raise MyException("Custom error message").
4. Exception Arguments: Pass the error message as an argument when raising a built-in or custom exception.
5. Example: raise ValueError("Invalid value provided").
6. Accessing the error message: Retrieve the error message from the exception object's args attribute.
7. Example: except MyException as e: print("Error:", e.args[0]).
8. String Interpolation: Format the error message using string interpolation techniques.
9. Example: raise ValueError(f"Invalid value {value} provided").
10. Concatenation: Concatenate strings to form the error message.

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

**Answer:-**

In Python, it is no longer recommended to use string-based exceptions as they have been deprecated since Python 2.6 and removed in Python 3. The use of string-based exceptions has several drawbacks:

1. Limited Information: String-based exceptions provide limited information about the exception. They only convey a textual description of the error without any structured data or context.
2. Lack of Exception Hierarchy: String-based exceptions do not allow for organizing exceptions in a hierarchical structure. This makes it harder to catch and handle specific types of exceptions and provide appropriate error handling based on the exception type.
3. Ambiguity and Fragility: String-based exceptions rely on string matching for exception handling, which can be error-prone and less reliable. String comparisons can be case-sensitive, leading to potential mismatches or bugs.
4. Reduced Maintainability: With string-based exceptions, it becomes difficult to refactor or change the exception names without manually updating every occurrence of the exception string in the codebase.

Instead of using string-based exceptions, it is recommended to use custom exception classes or built-in exception classes provided by Python. Custom exception classes allow for more structured and meaningful exception handling, enabling better error understanding, and facilitating appropriate error handling based on the exception type.