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

Sol:-

In Python 2.X, the syntax for catching exceptions was **except ExceptionType:**, or except ExceptionType, target: when the exception object is desired

In Python 3.X the syntax changed to **except ExceptionType as target:**

Python 2’s raise statement was designed at a time when exceptions weren’t classes, and an exception’s type, value, and traceback components were three separate objects:

raise ValueError, 'invalid input'

raise ValueError, 'invalid input', some\_traceback

In Python 3, one single object includes all information about an exception:

raise ValueError('invalid input')

e = ValueError('invalid input')

e.\_\_traceback\_\_ = some\_traceback

raise e

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

Sol:-

When a class-based exception is raised, it is matched to exception handlers based on the exception's class hierarchy. The process involves searching for an appropriate exception handler in the order of the exception's inheritance hierarchy until a matching handler is found or until the exception propagates to the top-level of the program.

Here's how the matching process works:

Exception Class Hierarchy: Each exception class in Python forms a hierarchy based on inheritance. The base class for all exceptions is BaseException, and all built-in and user-defined exceptions ultimately inherit from this class. This hierarchy allows for organizing and categorizing exceptions based on their relationships.

Searching for Matching Handlers: When an exception is raised, Python starts searching for exception handlers in the current scope and then continues to the enclosing scopes until it finds a matching handler or reaches the top-level scope (the global scope). The search follows the exception's class hierarchy.

Matching Exception Handlers: Python compares the raised exception's class (or any of its superclasses) with the exception classes specified in the except clauses. If there is a match between the raised exception's class and the exception class specified in an except clause, that handler is considered a match, and the corresponding code block is executed.

Exception Propagation: If a matching exception handler is not found within the current scope or its enclosing scopes, the exception propagates to the next enclosing scope. If the exception reaches the top-level scope without being caught by an appropriate handler, it becomes an unhandled exception, and the program terminates with an error message.

class CustomException(Exception):

pass

try:

raise CustomException("Something went wrong")

except CustomException:

print("CustomException handler")

except Exception:

print("Exception handler")

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

Sol:-

When it comes to attaching context information to exception artifacts in Python, there are several methods available. Two commonly used methods are:

Exception Arguments: When raising an exception, you can provide additional context information by passing arguments to the exception class constructor. These arguments can be used to store relevant data or details related to the exception. The exception class can define attributes to hold this information, which can then be accessed in exception handlers.

class CustomException(Exception):

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

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

self.context = context

try:

raise CustomException("Something went wrong", {"user\_id": 123, "action": "delete"})

except CustomException as e:

print(e.context)

Custom Exception Attributes: Another method is to define custom attributes in the exception class itself. By creating specific attributes in the exception class, you can directly attach context information to the exception instance, making it easily accessible within exception handlers.

class CustomException(Exception):

pass

try:

raise CustomException("Something went wrong")

except CustomException as e:

e.user\_id = 123

e.action = "delete"

print(e.user\_id, e.action)

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

Sol:-

Exception Class Constructor: One method is to specify the error message when creating an instance of the exception class. This can be done by overriding the \_\_init\_\_ method of the exception class and accepting a message parameter that represents the error message. Inside the \_\_init\_\_ method, you can store the message in an instance variable or pass it to the superclass constructor.

class CustomException(Exception):

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

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

try:

raise CustomException("Something went wrong")

except CustomException as e:

print(str(e))

Exception Arguments: Another method is to utilize the arguments of the exception class. By default, the base Exception class accepts an arbitrary number of arguments that can be used to pass information related to the exception. You can include the error message as one of the arguments and access it using the args attribute of the exception instance.

class CustomException(Exception):

pass

try:

raise CustomException("Something went wrong")

except CustomException as e:

print(e.args[0])

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

Sol:-

Lack of Type Safety: String-based exceptions do not provide any type safety. It becomes difficult to distinguish between different types of exceptions based on their strings alone. This can lead to ambiguity and make exception handling more error-prone.

Limited Information: String-based exceptions do not provide a standardized way to attach additional information or context to the exception. They lack the structure and capabilities of custom exception classes, which can have attributes, methods, and inheritance relationships. With custom exception classes, you can include relevant data and methods specific to the exceptional situation, enhancing error handling and diagnostics.

Reduced Expressiveness: Custom exception classes allow you to define more expressive exception hierarchies. By creating a hierarchy of exception classes, you can categorize and handle different types of exceptions more effectively. String-based exceptions lack this expressiveness and make it challenging to organize and manage exceptions in a structured manner.

Interoperability and Compatibility: Using string-based exceptions can cause compatibility issues when working with other libraries or code that expects exceptions to be objects. Modern Python code and libraries are designed to work with exception objects that inherit from the BaseException class or its subclasses. String-based exceptions do not adhere to this convention, which can lead to compatibility problems and make code integration more difficult.