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

Ans1

There are no specific new user-defined exception constraints.

We can create own custom exceptions.

Example

class CustomException(Exception):

pass

This creates a new exception class called CustomException that inherits from the built-in Exception class.

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

Ans2

When a class-based exception is raised, it is matched to handlers based on its type hierarchy. The interpreter checks if the exception is an instance of the handler's exception class or a subclass of it. If it is, then the handler's code is executed.

class MyException(Exception):

pass

try:

raise MyException("Something went wrong!")

except MyException:

print("Caught a MyException!")

except Exception:

print("Caught some other exception.")

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

Ans3

Using the args attribute of the exception object: The args attribute of an exception object is a tuple containing any arguments passed to the exception constructor.

class MyException(Exception):

pass

try:

raise MyException("Something went wrong!", 42)

except MyException as ex:

print("Caught a MyException with additional context:", ex.args[0], ex.args[1])

Defining custom attributes on the exception object

class MyException(Exception):

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

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

self.context = context

try:

raise MyException("Something went wrong!", {"foo": 42, "bar": "baz"})

except MyException as ex:

print("Caught a MyException with additional context:", ex.context)

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

Ans4

Passing a string argument to the exception constructor.

class MyException(Exception):

pass

try:

raise MyException("Something went wrong!")

except MyException as ex:

print("Caught a MyException with message:", str(ex))

Overriding the \_\_str\_\_ method of the exception class.

class MyException(Exception):

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

return "Custom error message"

try:

raise MyException()

except MyException as ex:

print("Caught a MyException with message:", str(ex))

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

Ans5

Lack of clarity: Using string literals to raise and catch exceptions makes the code less clear and more difficult to understand. It is not immediately clear what kind of exception is being raised or caught, and it is not possible to add any additional information to the exception.

Harder to maintain: Because string-based exceptions are not associated with a specific class, it is difficult to maintain them. There is no way to add methods or attributes to string-based exceptions, and it is not possible to create hierarchies of exceptions.

Limitations: String-based exceptions have limited functionality and cannot provide as much information as class-based exceptions.

For these reasons class based exceptions are used instead of string based exceptions.