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

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

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

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

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

Answers

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

In Python 3.X, there are no specific constraints on user-defined exceptions. Exceptions can be defined by creating a new class that directly or indirectly inherits from the `Exception` class or one of its subclasses[1][2][4].

The two main guidelines for defining user-defined exceptions are:

1. \*\*Exception classes should be named with names that end in "Error"\*\*, similar to the naming of standard exceptions in Python[1].

2. \*\*Exception classes should be derived from the `Exception` class or one of its subclasses\*\*, either directly or indirectly[1][2][4].

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

In Python, class-based exceptions that have been raised are matched to handlers using the `try-except` statement[5]. The `except` clause specifies the exception class or classes that the handler is designed to catch.

When an exception is raised, Python searches for an `except` clause that matches the exception class. The search starts in the `try` block and travels up the call stack, checking the `except` clauses in successive enclosing blocks, until a matching exception is found or there are no more blocks to check[5].

If a matching `except` clause is found, the exception is considered handled, and the code block associated with the `except` clause is executed. If no matching `except` clause is found, the exception is considered unhandled, and the program terminates with an error message[5].

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

In Python, you can attach context information to exception artifacts in two main ways:

1. \*\*By passing additional arguments to the exception class constructor\*\*[4]:

```python

class CustomException(Exception):

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

self.message = message

self.context = context

```

When raising the exception, you can provide the context information as an argument:

```python

raise CustomException("Something went wrong", {"key": "value"})

```

2. \*\*By adding attributes to the exception instance after it has been created\*\*[5]:

```python

try:

raise Exception("Error occurred")

except Exception as e:

e.context = {"key": "value"}

raise

```

In this case, you can add the context information to the exception instance using a custom attribute, and then re-raise the exception with the added context.

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

There are two main methods for specifying the text of an exception object's error message in Python:

1. \*\*By passing the error message as an argument to the exception class constructor\*\*[4]:

```python

class CustomException(Exception):

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

self.message = message

```

When raising the exception, you can provide the error message as an argument:

```python

raise CustomException("Something went wrong")

```

2. \*\*By overriding the `\_\_str\_\_` method of the exception class\*\*[1][3]:

```python

class CustomException(Exception):

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

self.value = value

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

return repr(self.value)

```

In this case, the `\_\_str\_\_` method is responsible for returning the string representation of the exception object, which will be used as the error message.

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

In Python 2.X, it was possible to use string-based exceptions, where you could raise a string instead of an exception object. However, this practice has been deprecated since Python 2.6 and is no longer used in Python 3.X[3].

There are several reasons why string-based exceptions are no longer recommended:

1. \*\*String-based exceptions lack the ability to carry additional information\*\*, such as context or attributes, which can be useful for handling and debugging exceptions[3].

2. \*\*Using strings for exceptions makes it difficult to catch and handle specific exceptions\*\*, as you would need to match the exact string value in the `except` clause[3].

3. \*\*String-based exceptions are less flexible and extensible than class-based exceptions\*\*, as you cannot create a hierarchy of exceptions or define custom behavior for exception handling[3].

4. \*\*The `BaseException.message` attribute, which was used to access the exception message, has been deprecated since Python 2.6 and removed in Python 3.0\*\*[3].

For these reasons, it is recommended to use class-based exceptions, which provide more flexibility, extensibility, and the ability to carry additional information, making exception handling more robust and maintainable in modern Python code.