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?

Solutions: Q1. As of my knowledge cutoff in September 2021, there were no specific user-defined exception constraints introduced in Python 3.X. However, Python provides a way to define custom exceptions by creating new exception classes derived from the built-in `Exception` class or its subclasses. These custom exceptions can be raised and caught like any other built-in exceptions.

Q2. Class-based exceptions that have been raised are matched to handlers based on the inheritance hierarchy of the exception classes. When an exception is raised, Python searches for an appropriate exception handler by traversing the exception hierarchy from the most specific to the least specific. The search stops at the first matching handler it encounters. If no matching handler is found, the exception propagates up the call stack or results in an unhandled exception error.

Q3. Two methods for attaching context information to exception artifacts are:

- Exception Arguments: When raising an exception, you can pass additional arguments to the exception class to provide context information. These arguments can be accessed within the exception handler using the `args` attribute of the exception object. For example:

```python

class CustomException(Exception):

pass

try:

raise CustomException("Something went wrong", context\_info)

except CustomException as e:

error\_message = e.args[0]

context\_info = e.args[1]

# Handle the exception and context information

```

- Exception Attributes: You can define custom attributes in your exception class to store specific context information. These attributes can be accessed within the exception handler using the attribute access notation (`exception.attribute`). For example:

```python

class CustomException(Exception):

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

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

self.context\_info = context\_info

try:

raise CustomException("Something went wrong", context\_info)

except CustomException as e:

error\_message = str(e)

context\_info = e.context\_info

# Handle the exception and context information

```

Q4. Two methods for specifying the text of an exception object's error message are:

- Exception Class Definition: You can define the error message directly within the exception class definition by overriding the `\_\_str\_\_` method. This method should return a string representation of the exception. For example:

```python

class CustomException(Exception):

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

return "Custom exception occurred"

try:

raise CustomException()

except CustomException as e:

error\_message = str(e)

# Handle the exception and error message

```

- Exception Instance Initialization: You can pass the error message as an argument when initializing an instance of the exception class. This requires defining an appropriate `\_\_init\_\_` method in your exception class. For example:

```python

class CustomException(Exception):

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

self.message = message

try:

raise CustomException("Custom exception occurred")

except CustomException as e:

error\_message = e.message

# Handle the exception and error message

```

Q5. String-based exceptions were commonly used in older versions of Python but are no longer recommended. Instead, Python encourages the use of class-based exceptions derived from the built-in `Exception` class. Using string-based exceptions had several drawbacks:

- Lack of specificity: String-based exceptions only provided a single string message and lacked additional context or information about the type of exception. This made it difficult to handle specific exceptions or differentiate between different error scenarios.

- Limited functionality: String-based exceptions couldn't benefit from the full range of features provided by class-based exceptions, such as inheritance, attribute access, and customization through methods like `\_\_str\_\_`.

- Code clarity and maintainability: Class-based exceptions make the code more readable and maintainable by clearly indicating the nature of the exception and allowing for more specific exception handling. They also provide a consistent interface for handling exceptions across different parts of a program.

Overall, the transition to class-based exceptions in Python enhances code organization, readability, and error handling capabilities, leading to more robust and maintainable code.