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

As of my knowledge cutoff in September 2021, there are no specific predefined "user-defined exception constraints" in Python 3.x. However, Python allows users to define their own custom exceptions by creating a new class that inherits from the built-in `Exception` class or any of its subclasses. The design and constraints of user-defined exceptions are entirely up to the programmer's discretion and specific use case.

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 a mechanism called exception handling. When an exception is raised, the Python interpreter looks for an appropriate exception handler to handle it.

Exception handlers are defined using the `try-except` statement. Inside the `try` block, code that may raise an exception is placed. After the `try` block, one or more `except` blocks can be defined to handle specific types of exceptions.

Each `except` block specifies the type of exception it can handle. If the raised exception matches the type specified in an `except` block, that block is executed. If the exception doesn't match any of the `except` blocks, it propagates up the call stack until it finds a suitable handler or results in a program termination.

Here's an example:

```python

try:

# Code that may raise an exception

raise ValueError("Something went wrong")

except ValueError:

# Handler for ValueError

print("Handling ValueError")

except Exception:

# Generic exception handler

print("Handling generic exception")

```

In this example, if a `ValueError` is raised, the first `except` block is executed. If a different type of exception is raised, the second `except` block is executed. If no matching handler is found, the exception will propagate further up.

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

Two methods for attaching context information to exception artifacts are:

1. Adding additional attributes to the exception object: You can define custom attributes on the exception object and populate them with relevant context information before raising the exception. These attributes can then be accessed within the exception handler.

2. Wrapping the exception: Instead of raising the exception directly, you can create a custom exception class that inherits from the base exception and includes additional attributes or methods to capture and provide context information. This custom exception can then be raised with the desired context information encapsulated within it.

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

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

1. Passing a string argument to the exception constructor: When raising an exception, you can provide a string argument to the exception class constructor, specifying the error message. This allows you to customize the error message based on the specific context or information relevant to the exception.

2. Overriding the `\_\_str\_\_` method: By overriding the `\_\_str\_\_` method in a custom exception class, you can define the logic for generating the error message. This method is called when the exception object needs to be converted to a string, such as when printing or logging the exception, allowing you to control the error message format and content.

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

String-based exceptions are no longer used in modern Python because they have several drawbacks:

1. Lack of clarity and specificity: String-based exceptions provide limited information about the type and nature of the exception, making it difficult to determine the cause of the error.

2. Inability to catch specific exceptions: Since all string-based exceptions are essentially the same type (`Exception`), it becomes challenging to catch and handle specific exceptions separately.

3. Reduced functionality: String-based exceptions lack the additional functionality and attributes provided by class-based exceptions, such as custom methods or properties. Class-based exceptions offer more flexibility and extensibility in exception handling.