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

As of my knowledge cutoff date of September 2021, I believe there were no specific "latest" user-defined exception constraints in Python 3.x. Python allows users to define custom exceptions by creating a new class that inherits from the built-in Exception class or one of its subclasses. The only constraints are that the custom exception class should have a descriptive name and can include additional methods or attributes as needed.

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

When a class-based exception is raised in Python, the interpreter looks for an appropriate exception handler to handle it. The search for a handler proceeds as follows:

1. The interpreter looks for an exception handler that directly matches the raised exception class. If a handler is found, it is executed.
2. If no direct match is found, the interpreter looks for handlers that match the exception class in the exception hierarchy. The exception hierarchy is determined by the class's inheritance relationships. Handlers for base classes will also match exceptions raised by subclasses.
3. If no matching handler is found in the current function or method, the interpreter unwinds the call stack and searches the calling function, then its caller, and so on until a matching handler is found or the exception propagates to the top level of the program.

If no handler is found, the interpreter prints a traceback and exits the program.

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

Attaching context information to exception artifacts in Python can be helpful for debugging and understanding the circumstances that led to the exception being raised. Here are two common methods for doing this:

1. Adding a custom attribute to the exception object: You can add custom attributes to an exception object to provide context information. For example, if an exception is raised because of an invalid argument passed to a function, you can attach the invalid argument value to the exception object using a custom attribute. Then, when you catch the exception, you can inspect the attribute to determine the cause of the exception.
2. Using the raise ... from ... syntax: The raise ... from ... syntax in Python allows you to raise one exception while indicating that it was caused by another. The original exception can be attached as the "cause" of the new exception, which can provide context information about what triggered the error. For example, if an exception is raised while reading a file, you can attach the original file-related exception as the cause of the new exception raised due to the higher-level processing of the file's content.

For both of these methods, it's important to use context information judiciously and only include relevant details. Including too much information or irrelevant data can make the exception message difficult to read and understand.

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

When raising an exception object in Python, you can provide an error message that describes the reason for the exception. Here are two common methods for specifying the text of an exception object's error message:

1. Passing a string argument to the exception constructor: Most built-in exceptions in Python accept a string argument that specifies the error message. For example, to raise a ValueError with a custom error message, you can do:

raise ValueError("The value is not valid")

This will create a new ValueError object with the error message "The value is not valid".

1. Overriding the \_\_str\_\_() method of a custom exception class: If you define your own custom exception class, you can override its \_\_str\_\_() method to return a custom error message. For example, you might define a CustomException class as follows:

class CustomException(Exception):

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

return "A custom error occurred"

Then, when you raise an instance of this class, it will display the error message returned by the \_\_str\_\_() method:

raise CustomException()

This will create a new CustomException object with the error message "A custom error occurred".

Note that it's important to provide clear and informative error messages that describe the reason for the exception. The error message should help the user or developer understand what went wrong and how to fix the problem.

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

As a language model, I do not personally use exceptions or write code. However, I can tell you that using string-based exceptions in Python is discouraged because it has several drawbacks:

1. String-based exceptions are less descriptive and less specific than class-based exceptions. Using a string to represent an error message can make it difficult to identify the exact cause of an exception and can make it harder to debug and fix the issue.

2. String-based exceptions do not support inheritance. In Python, it's common to define custom exception classes that inherit from the built-in Exception class or one of its subclasses. This allows you to create a hierarchy of exception classes that provide more detailed and specific error messages. This is not possible with string-based exceptions.

3. String-based exceptions do not provide access to the exception object. When an exception is raised, an exception object is created that contains information about the error. If you use a string-based exception, you cannot access this object to extract additional information about the error.

For these reasons, it is generally recommended to use class-based exceptions in Python rather than string-based exceptions.