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

Inherit from BaseException or its Subclasses: When defining custom exceptions, it's generally recommended to have them inherit from the built-in BaseException class or one of its subclasses (Exception, TypeError, etc.). This ensures that your custom exceptions follow the same exception hierarchy as built-in exceptions.

```
class CustomError(Exception):
    pass

raise CustomError("Example of Custom Exceptions in Python")

Output: CustomError: Example of Custom Exceptions in Python
```

Provide Informative Error Messages: Exception messages should be clear, concise, and informative. They should help developers understand what went wrong and potentially guide them in troubleshooting the issue. When raising your custom exception, consider providing a meaningful error message that describes the exceptional condition.

```
class CustomError(Exception):
    def __init__(self, message):
        self.message = message
    def __str__(self):
        return self.message
```

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

When a class-based exception is raised in Python, it is matched to handlers in the order in which the except clauses appear in the code. The exception hierarchy is traversed from the raised exception upwards, looking for a matching exception type in the except clauses. The first matching except block encountered will be executed, and the program will then proceed from there.

Here's how the process works:

Exception Hierarchy Traversal: When an exception is raised, Python starts searching for matching except blocks from the raised exception's class and works its way up the exception hierarchy towards the base BaseException class. This means that more specific exceptions are checked before broader ones.

Matching the except Block: When a matching exception type is found in an except clause, the code within that block is executed. If multiple except clauses match the raised exception type, only the first one encountered is executed. Subsequent except blocks that match the same exception type or its parent types are ignored.

No Match or Unhandled Exceptions: If no matching except block is found in the code, or if the exception hierarchy traversal reaches the BaseException class without finding a match, the exception becomes an unhandled exception. This typically results in the program terminating and displaying an error message along with a traceback.

Example:

```
try:
    # Code that might raise an exception
except SpecificException:
    # Code to handle SpecificException
except ParentException:
    # Code to handle ParentException
except BaseException:
    # Code to handle BaseException (not recommended)
```

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

Attaching context information to exception artifacts is important for providing additional details about the circumstances that led to an exception. This information can aid in debugging, troubleshooting, and identifying the root cause of issues. Here are two methods for attaching context information to exception artifacts:

Custom Exception Classes with Attributes: When defining our own custom exception classes, we can include attributes that store relevant context information. These attributes can be set when an exception is raised, providing valuable information about the state of the program when the exception occurred.

In this example, the context_info dictionary is attached to the CustomError exception instance, providing additional context about the error.

```
class CustomError(Exception):
    def __init__(self, message, context_info):
        super().__init__(message)
        self.context_info = context_info

try:
    # Code that might raise an exception
except SomeException:
    context = {"variable": value, "user": "John"}
    raise CustomError("An error occurred.", context)
```

Using Exception Chaining and from Keyword: Python's exception chaining feature allows us to associate a new exception with the context of an existing exception using the from keyword. This can be helpful when a higher-level exception is raised, and we want to include the lower-level exception as context.

```
try:
    # Code that might raise an exception
except LowerLevelException as e:
    raise HigherLevelException("Higher-level error occurred") from e
```

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

When raising exceptions in Python, we can specify the text of an exception object's error message using various methods. The error message provides important information about the exception and helps developers understand what went wrong. Here are two common methods for specifying the text of an exception object's error message:

Using the Exception's Constructor: When defining custom exception classes, we can pass the error message as an argument to the constructor of the base exception class (Exception or its subclasses). This is a straightforward way to provide a custom error message when an exception is raised.

Example: the CustomError class's constructor accepts a message argument, which is then stored as an instance attribute. The __str__ method is overridden to provide a string representation of the exception, which includes the custom error message.

```
class CustomError(Exception):
    def __init__(self, message):
        self.message = message
    def __str__(self):
        return self.message

try:
    # Code that might raise an exception
    if some_condition:
        raise CustomError("An error occurred due to some condition.")
except CustomError as e:
    print("Error:", e)
```

Concatenating Strings: Another method to specify an exception's error message is to concatenate strings directly within the raise statement. This is a simpler approach for basic scenarios where you don't need to create a custom exception class.
Example:
The error message is constructed using string concatenation and provided directly when raising the exception.

```
try:
    # Code that might raise an exception
    if some_condition:
        raise Exception("An error occurred due to some condition: " + additional_info)
except Exception as e:
    print("Error:", e)
```

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

string-based exceptions have been largely deprecated and discouraged in Python. This means that while we can still raise and catch string-based exceptions, it's recommended to use class-based exceptions instead. Here's why string-based exceptions are no longer favored:

1. **Lack of Structure and Information:** String-based exceptions provide only the error message as a string without any additional structured information. Class-based exceptions, on the other hand, allow us to include attributes and methods that provide context, additional data, and behavior to the exception instance.
2. **Limited Customization:** With string-based exceptions, we're limited to the string message itself. Class-based exceptions offer greater customization, allowing us to define our own attributes and methods to convey more information about the error, making them more powerful and flexible.
3. **Consistency and Hierarchy:** Class-based exceptions follow a hierarchy that mirrors the exception hierarchy in Python. This makes it easier to catch and handle specific types of exceptions, and it provides a consistent approach for working with exceptions.
4. **Clear Intent:** Using class-based exceptions makes the intent of the code clearer. When you raise a class-based exception, you are explicitly indicating that we are dealing with an exceptional condition. This enhances the code's readability and maintainability.
5. **Better Exception Handling:** Class-based exceptions allow us to catch specific types of exceptions using except clauses. This selective handling improves error management and allows for more effective and fine-grained error handling strategies.