Q1. What are the two latest user-defined exception constraints in Python 3.X?
The two latest user-defined exception constraints introduced in Python 3.X are:
1. Exceptions must be derived from the built-in `Exception` class or one of its subclasses. This constraint ensures that user-defined exceptions are properly categorized and inherit the necessary behavior and attributes from the base `Exception` class.
2. Exceptions can include an optional argument that provides additional information about the exception. This allows developers to pass custom data or context-specific details when raising an exception, providing more descriptive error messages and aiding in the debugging process.

Q2. How are class-based exceptions that have been raised matched to handlers?
In Python, when a class-based exception is raised, the interpreter looks for an appropriate exception handler by following the method resolution order (MRO) of the exception's class hierarchy. It searches for the closest matching exception handler by traversing the inheritance chain of the exception classes. The first matching handler encountered in the inheritance chain, from the raised exception's class down to the base `Exception` class, is executed. If no matching handler is found, the exception propagates to the calling context or the default exception handler.

Q3. Describe two methods for attaching context information to exception artifacts.
Two methods for attaching context information to exception artifacts in Python are:
1. Including additional arguments when raising an exception: When raising an exception, you can pass additional arguments that provide context-specific information related to the exception. These arguments can include relevant data, error codes, or any other details that help in understanding the cause of the exception.
2. Using exception attributes: Exception classes can define attributes that store context information about the exception. These attributes can be set when creating an instance of the exception class or dynamically modified during exception handling. By accessing these attributes in exception handlers, you can retrieve and utilize the contextual information associated with the exception.

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 in Python are:
1. Passing a string argument when raising an exception: When raising an exception, you can pass a string argument that represents the error message. This argument can provide a concise and informative description of the exception, conveying details about the encountered error.
2. Defining a custom `__str__` method in the exception class: By overriding the `__str__` method in the custom exception class, you can define the logic for generating the error message string. This allows you to customize the formatting and content of the error message based on the specific exception class and its attributes.

Q5. Why do you no longer use string-based exceptions?
String-based exceptions have been deprecated and are no longer used in Python because they have several drawbacks. Using string-based exceptions makes it difficult to differentiate between different types of exceptions and handle them appropriately. They lack the structured nature of class-based exceptions, making it harder to organize and categorize exceptions based on their behavior or attributes. String-based exceptions also do not support inheritance or customization, limiting the flexibility and extensibility of exception handling. By using class-based exceptions, Python provides a more robust and structured approach to exception handling, allowing for better organization, customization, and handling of different types of exceptions.