Python Advanced Assignment-8

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

Ans.

1.Inheritance from `BaseException`: All user-defined exceptions must inherit from `BaseException`, typically from its subclass `Exception`. This ensures that user-defined exceptions are consistent with Python's exception hierarchy and can be caught by generic `except Exception` clauses.

2.New-style class definition: All user-defined exceptions must be new-style classes, meaning they should use the `class` keyword and inherit from `Exception` or any subclass of it. Old-style classes (from Python 2.x) are not allowed.

```
class CustomError(Exception):

    pass
```

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

Ans.

Class-based exceptions are matched to handlers using the class hierarchy. When an exception is raised, Python looks for an `except` clause that matches the exception's class. If an exact match is not found, it looks for handlers that match any superclass of the exception, moving up the inheritance hierarchy until a match is found or until the `BaseException` class is reached. If no match is found, the exception propagates, and the program may terminate.

```
class CustomError(Exception):

    pass


try:

    raise CustomError("An error occurred.")

except CustomError:

    print("Caught a CustomError.")

except Exception:

    print("Caught a generic exception.")
```

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

Ans.

1.Custom Attributes: You can define custom attributes in a user-defined exception class to store additional context information.

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


try:
    raise CustomError("An error occurred.", 42)
except CustomError as e:
    print(f"Error: {e}, Value: {e.value}")
```

2. Using the `__cause__` or `__context__` attributes: Python's built-in exception objects have `__cause__` and `__context__` attributes that automatically attach information about the originating exception when one exception causes another. This helps track the chain of errors.

```
try:
    1 / 0
except ZeroDivisionError as e:
    raise ValueError("A value error occurred.") from e
```

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

Ans.

1. Passing the Message as an Argument: When defining a custom exception class, you can pass a message string as an argument to the constructor and call the `super()` method to set it.

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

2. Overriding the `__str__` Method: You can override the `__str__` method in your custom exception class to provide a custom representation of the exception's error message when it is printed.

```
class CustomError(Exception):
    def __init__(self, value):
        self.value = value

    def __str__(self):
        return f"CustomError occurred with value: {self.value}"
```

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

Ans.

String-based exceptions are not used in Python 3.X because they lack structure, hierarchy, and flexibility. They do not provide the benefits of class-based exceptions, such as inheritance and the ability to store context information and attributes. Class-based exceptions are more organized, making it easier to categorize and handle errors based on the type of the exception rather than just matching strings. This approach allows for better maintainability and readability of the code.