Python User-Defined Exceptions — Questions & Answers (Set 8)

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

1) Must derive from BaseException (usually Exception).
- All user-defined exceptions must be subclasses of BaseException.
- Best practice: subclass Exception.

2) Cannot use old-style string exceptions.
- Unlike Python 2, raising plain strings (raise 'Error') is invalid.
- Only class-based exceptions are supported.

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

Python matches raised exceptions against except clauses using class inheritance checks.
A handler matches if the raised exception is an instance of the listed class or its subclass.
Example:
class MyError(Exception): pass
class SubError(MyError): pass

try:
raise SubError('oops')
except MyError:
print('Caught') # Matches subclass too

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

1) Custom attributes in custom exception classes:
class DataError(Exception):
def __init__(self, record_id, message):
super().__init__(message)
self.record_id = record_id

2) Exception chaining (raise ... from ...):
try:
1 / 0
except ZeroDivisionError as e:
raise RuntimeError('Computation failed') from e

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

1) Pass a string when raising:
raise ValueError('Invalid argument')

2) Override __str__ in a custom exception class:
class CustomError(Exception):
def __str__(self):
return 'Custom failure occurred'

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

String exceptions were a Python 2 feature, removed in Python 3.
Problems:
- Not class-based → no inheritance.
- Cannot attach metadata.
- Fragile string matching.

Class-based exceptions provide structure, extensibility, and clarity.