**Q1. Why use the Exception class while creating a Custom Exception?**

The Exception class is the base class for all exceptions in Python, providing essential functionality for defining and handling exceptions. When creating a custom exception, inheriting from the Exception class ensures that:

1. **Standardization**: The custom exception follows Python's built-in exception structure, making it easier to integrate with existing error-handling mechanisms.
2. **Traceback Support**: It automatically includes features like tracebacks, which help developers debug errors by providing the call stack information.
3. **Custom Behavior**: While inheriting from Exception, you can add additional attributes and methods specific to your custom exception.
4. **Compatibility**: Custom exceptions derived from Exception work seamlessly with try-except blocks.

Example:

class CustomError(Exception):

def \_\_init\_\_(self, message):

super().\_\_init\_\_(message)

**Q2. Python program to print Python Exception Hierarchy**

import inspect

def print\_exception\_hierarchy():

for name, obj in inspect.getmembers(\_\_builtins\_\_):

if inspect.isclass(obj) and issubclass(obj, BaseException):

print(name)

print\_exception\_hierarchy()

To see the complete hierarchy in a tree format:

def print\_exception\_tree(cls, indent=0):

print(' ' \* indent + cls.\_\_name\_\_)

for subclass in cls.\_\_subclasses\_\_():

print\_exception\_tree(subclass, indent + 4)

print\_exception\_tree(BaseException)

**Q3. Errors defined in the ArithmeticError class**

The ArithmeticError class is a base class for errors arising from numeric calculations. Its subclasses include:

1. **ZeroDivisionError**: Raised when dividing by zero.
2. **OverflowError**: Raised when a numerical operation exceeds the range of the data type.
3. **FloatingPointError**: Raised for errors in floating-point calculations (rarely used).

**Examples:**

* **ZeroDivisionError**:

try:

result = 10 / 0

except ZeroDivisionError as e:

print(f"Error: {e}")

* **OverflowError**:

import math

try:

result = math.exp(1000)

except OverflowError as e:

print(f"Error: {e}")

**Q4. Why use the LookupError class?**

The LookupError class serves as the base class for exceptions raised when a key or index is invalid for a collection, such as a dictionary or list. It helps catch both KeyError and IndexError in a generic way.

**Examples:**

* **KeyError**: Raised when a dictionary key is not found.

try:

my\_dict = {'a': 1, 'b': 2}

print(my\_dict['c'])

except KeyError as e:

print(f"KeyError: {e}")

* **IndexError**: Raised when accessing an index out of range in a list.

try:

my\_list = [1, 2, 3]

print(my\_list[5])

except IndexError as e:

print(f"IndexError: {e}")

**Q5. Explain ImportError and ModuleNotFoundError**

* **ImportError**: Raised when a module or an attribute within a module cannot be imported. This error typically occurs when the module exists but there is an issue in its import.
* **ModuleNotFoundError**: A subclass of ImportError, specifically raised when the Python interpreter cannot locate the module. It was introduced in Python 3.6 to provide more clarity.

**Example:**

try:

import non\_existent\_module

except ModuleNotFoundError as e:

print(f"ModuleNotFoundError: {e}")

try:

from math import non\_existent\_function

except ImportError as e:

print(f"ImportError: {e}")

**Q6. Best Practices for Exception Handling in Python**

1. **Be Specific with Exceptions**: Catch specific exceptions instead of using a general except clause to avoid masking unexpected errors.
2. try:
3. result = 10 / 0
4. except ZeroDivisionError as e:
5. print(f"Error: {e}")
6. **Use finally for Cleanup**: Ensure resources like files or network connections are closed.
7. try:
8. file = open('example.txt', 'r')
9. finally:
10. file.close()
11. **Avoid Catching All Exceptions**: Using except Exception or except: indiscriminately can hide bugs.
12. try:
13. risky\_code()
14. except ValueError as e:
15. print(f"Specific Error: {e}")
16. **Log Exceptions**: Use logging instead of print statements for better debugging and record-keeping.
17. import logging
18. logging.basicConfig(level=logging.ERROR)
19. try:
20. risky\_code()
21. except Exception as e:
22. logging.error("An error occurred", exc\_info=True)
23. **Use Custom Exceptions**: For application-specific errors, create custom exceptions for clarity.
24. class MyCustomError(Exception):
25. pass
26. **Do Not Suppress Exceptions Silently**: Avoid empty except blocks.
27. try:
28. risky\_code()
29. except Exception as e:
30. print(f"Error occurred: {e}")
31. **Use Context Managers**: For resource management, prefer context managers (with statements).
32. with open('example.txt', 'r') as file:
33. data = file.read()
34. **Validate Inputs**: Prevent exceptions by validating inputs early in the code.
35. if not isinstance(number, int):
36. raise ValueError("Input must be an integer")