In [ ]:
```
Q1. Explain why we have to use the Exception class while creating a Custom Exception.
```

```
In Python, the Exception class is the base class for all exceptions. When creating a
custom exception, it is essential to inherit from the Exception class to ensure that
the new exception type is recognized and treated as an exception by the Python
interpreter.
```

In [1]:
```
class CustomError(Exception):
    def __init__(self, message):
        super().__init__(message)

# Using the custom exception
try:
    raise CustomError("This is a custom exception.")
except CustomError as ce:
    print(f"Custom Exception: {ce}")
```

```
Custom Exception: This is a custom exception.
```

In [ ]:
```
Q2. Write a python program to print Python Exception Hierarchy.
```

In [2]:
```
# Python Exception Hierarchy Program
def print_exception_hierarchy(exception_class, indentation=0):
    print("  " * indentation + str(exception_class.__name__))
    for subclass in exception_class.__subclasses__():
        print_exception_hierarchy(subclass, indentation + 1)

# Print Python Exception Hierarchy
print_exception_hierarchy(BaseException)
```

```
    ExpatError
    Error
      ProtocolError
      ResponseError
      Fault
    ParserSyntaxError
    ResolutionError
      VersionConflict
        ContextualVersionConflict
      DistributionNotFound
      UnknownExtra
    _Error
    UnableToResolveVariableException
    InvalidTypeInArgsException
    CustomError
  GeneratorExit
  KeyboardInterrupt
  SystemExit
  CancelledError
  AbortThread
```

In [ ]:
```
Q3. What errors are defined in the ArithmeticError class? Explain any two with an example
```

```
The ArithmeticError class in Python represents errors that occur during arithmetic
operations. Two common errors defined in this class are ZeroDivisionError and
OverflowError.
```

In [3]:
```
try:
    result = 10 / 0
except ZeroDivisionError as zde:
    print(f"ZeroDivisionError: {zde}")
```

```
ZeroDivisionError: division by zero
```

In [4]:
```
import math

try:
    result = math.exp(1000)
except OverflowError as oe:
    print(f"OverflowError: {oe}")
```

```
OverflowError: math range error
```

In [ ]:
```
Q4. Why is the LookupError class used? Explain with an example KeyError and IndexError.
```

```
The LookupError class is a base class for errors that occur when trying to access a
sequence using a key or index that is not present. Two common subclasses are KeyError
and IndexError.
```

In [5]:
```
my_dict = {"name": "John", "age": 25}

try:
    value = my_dict["gender"]
except KeyError as ke:
    print(f"KeyError: {ke}")
```

```
KeyError: 'gender'
```

In [6]:
```
my_list = [1, 2, 3, 4, 5]

try:
    element = my_list[10]
except IndexError as ie:
    print(f"IndexError: {ie}")
```

```
IndexError: list index out of range
```

```
Q5. Explain ImportError. What is ModuleNotFoundError?


ImportError is raised when an import statement fails to find and load a module.
ModuleNotFoundError is a subclass of ImportError and is specifically raised when the
specified module cannot be found.
```

In [7]:
```
try:
    import non_existent_module
except ImportError as ie:
    print(f"ImportError: {ie}")

try:
    from non_existent_package import module
except ModuleNotFoundError as mne:
    print(f"ModuleNotFoundError: {mne}")
```

```
ImportError: No module named 'non_existent_module'
ModuleNotFoundError: No module named 'non_existent_package'
```

In [ ]:
```
Q6. List down some best practices for exception handling in Python.
```

```
Specific Exception Handling: Catch specific exceptions rather than using a broad except
clause.

Use else Block: Use the else block to execute code when no exceptions are raised in the
try block.

Use finally Block: Utilize the finally block to guarantee code execution, whether an
exception occurs or not.

Avoid Bare except: Avoid catching all exceptions using a bare except clause; it may
hide unexpected errors.

Logging: Use the logging module to log exceptions for debugging and monitoring.

Custom Exception Classes: Create and use custom exception classes for application-
specific error handling.

Graceful Degradation: Design the code to gracefully degrade when exceptions occur,
providing a meaningful response to users.

Keep It Simple: Keep exception handling code simple and readable; avoid unnecessary
complexity.

Check Before Operations: Check conditions before performing operations to prevent
exceptions.

Use Context Managers: Utilize context managers (e.g., with statement) for resource
management and cleanup.
```

In [ ]:
```

```