Q1. Explain why we have to use the Exception class while creating a Custom Exception.

In Python, exceptions are used to handle errors and unexpected events that may occur during program execution. While Python provides many built-in exceptions, sometimes you may need to define your own custom exception to handle specific errors in your program.

When creating a custom exception in Python, it is recommended to use the Exception class as the base class for your custom exception. This is because the Exception class is the root of the exception hierarchy in Python, and it provides a number of useful methods and attributes that you can use in your custom exception.

By inheriting from the Exception class, your custom exception will automatically inherit all of the methods and attributes of the Exception class. This includes the ability to define a custom message for your exception, as well as the ability to specify an error code, traceback information, and other useful information.

Using the Exception class also ensures that your custom exception will be compatible with the standard Python exception handling mechanisms. This means that you can catch your custom exception using a try-except block, just like you would with any other exception in Python.

Overall, using the Exception class as the base class for your custom exception provides a number of benefits and ensures that your custom exception will be well-integrated with the rest of your Python code.

In [5]:
```
#Q2. Write a python program to print Python Exception Hierarchy.

class PrintExceptionHierarchy:
    def __init__(self, exc_class=Exception, level=0):
        self.exc_class = exc_class
        self.level = level

    def __str__(self):
        indent = ' ' * self.level
        exception_hierarchy = indent + str(self.exc_class.__name__)
        for sub_class in self.exc_class.__subclasses__():
            exception_hierarchy += '\n' + PrintExceptionHierarchy(sub_class, self.
        return exception_hierarchy

print(PrintExceptionHierarchy())
```

In [12]:
```
#Q3. What errors are defined in the ArithmeticError class? Explain any two with an
x = 10
y = 0
try:
    z = x / y
except ZeroDivisionError:
    print("Error: Attempted to divide by zero")
```

```
Error: Attempted to divide by zero
```

In [11]:
```
#Q3. What errors are defined in the ArithmeticError class? Explain any two with an
import sys
x = sys.maxsize
try:
```

```
    y = x * x
except OverflowError:
    print("Error: Result too large to be represented")
```

Q4. Why LookupError class is used? Explain with an example KeyError and IndexError.

In Python, the LookupError class is a base class for errors that occur when trying to access an item in a sequence or mapping that does not exist. It is a subclass of the built-in Exception class and is used as a base class for more specific lookup-related exceptions. Here are two examples of errors that are defined in the LookupError class:

In [15]:
```
#Q4. Why LookupError class is used? Explain with an example KeyError and IndexError
my_dict = {'a': 1, 'b': 2, 'c': 3}
try:
    value = my_dict['d']
except KeyError:
    print("Error: Key not found")
```

```
Error: Key not found
```

In [16]:
```
#Q4. Why LookupError class is used? Explain with an example KeyError and IndexError
my_list = [1, 2, 3]
try:
    value = my_list[3]
except IndexError:
    print("Error: Index out of bounds")
```

```
Error: Index out of bounds
```

Q5. Explain ImportError. What is ModuleNotFoundError?

In Python, ImportError is an exception that is raised when an import statement fails to import a module. This can occur for several reasons, such as a misspelled module name, an invalid or inaccessible file path, or a missing dependency. Here's an example of how

In [17]:
```
#Q5. Explain ImportError. What is ModuleNotFoundError?
try:
    import non_existent_module
except ImportError:
    print("Error: Failed to import module")
```

```
Error: Failed to import module
```

In [18]:
```
#Q5. Explain ImportError. What is ModuleNotFoundError?
try:
    import non_existent_module
except ModuleNotFoundError:
    print("Error: Module not found")
```

```
Error: Module not found
```

Q6. List down some best practices for exception handling in python.

Here are some best practices for exception handling in Python:

Use specific exception types: Instead of using a generic Exception class, use specific exception types to catch and handle different types of errors. This helps to identify the specific cause of the error and handle it accordingly.

Handle exceptions as close to the source as possible: Catch exceptions as close to the source of the error as possible, and handle them in the same block. This helps to keep the code organized and maintainable.

Use try-except-finally blocks: Use try-except-finally blocks to handle exceptions. The try block contains the code that may raise an exception, the except block contains the code to handle the exception, and the finally block contains the code that should be executed regardless of whether an exception was raised or not.

Avoid catching all exceptions: Avoid catching all exceptions using a bare except block. This can mask other errors and make it difficult to diagnose the cause of the error.

Use exception chaining: Use exception chaining to provide more information about the cause of the error. This involves raising a new exception with the original exception as the cause.

Use logging: Use logging to log errors and exceptions. This can help in debugging and identifying the cause of the error.

Raise exceptions when appropriate: Raise exceptions when appropriate to signal an error condition. This can help to prevent unexpected behavior and provide better error messages.

Use context managers: Use context managers to handle resources, such as files, sockets, and database connections. This ensures that the resources are properly cleaned up, even if an exception is raised.

In [ ]: