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

When creating a custom exception in Python, it is essential to use the Exception class (or one of its subclasses) as the base class for your custom exception. Here are the main reasons for this:

1. **Consistency with Python's Exception Hierarchy**:
   - Python's exception hierarchy is structured with BaseException at the top, followed by Exception, and then other built-in exceptions. By inheriting from the Exception class, your custom exception becomes part of this hierarchy, ensuring that it integrates seamlessly with Python's exception handling mechanisms.
2. **Catchability**:
   - Custom exceptions derived from the Exception class can be caught using a generic except Exception block. If your custom exception does not inherit from Exception, it may not be caught by such a block, leading to unhandled exceptions.
3. **Standard Exception Features**:
   - The Exception class provides standard methods and attributes that are useful for handling exceptions, such as the ability to store and display error messages. By inheriting from Exception, your custom exception gains these features automatically.
4. **Best Practices**:
   - Following best practices in Python programming, custom exceptions should inherit from Exception or its subclasses to ensure that they conform to the expected behavior of exceptions in Python. This makes your code more readable and maintainable for other developers.
5. **Clarity and Specificity**:
   - Custom exceptions can provide more clarity and specificity when handling errors in your code. By creating a custom exception that inherits from Exception, you can define specific error types that make it easier to identify and respond to different error conditions.

2. Write a python program to print Python Exception Hierarchy

```
def print_exception_hierarchy(exception_class, indent=0):

    print(' ' * indent + exception_class.__name__)

    for subclass in exception_class.__subclasses__():

        print_exception_hierarchy(subclass, indent + 2)

print_exception_hierarchy(BaseException)
```

3. What errors are defined in the ArithmeticError class? Explain any two with an example

The ArithmeticError class in Python is a built-in exception class that serves as the base class for all errors that occur for numeric calculations. Three main exceptions are defined under ArithmeticError:

1. FloatingPointError
2. OverflowError
3. ZeroDivisionError

Let's explain OverflowError and ZeroDivisionError with examples.

OverflowError

An OverflowError is raised when the result of an arithmetic operation is too large to be represented within the available range of numerical values.

Example:

```
import math

try:

    result = math.exp(1000)

except OverflowError as e:

    print(f"OverflowError: {e}")
```

ZeroDivisionError

A ZeroDivisionError is raised when an attempt is made to divide a number by zero

Example:

```
try:

    result = 10 / 0

except ZeroDivisionError as e:

    print(f"ZeroDivisionError: {e}")
```

These exceptions help in handling specific error conditions that can occur during arithmetic operations, allowing the program to respond appropriately to such situations.

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

The LookupError class in Python is a built-in exception class that serves as the base class for all exceptions that occur when a key or index used for lookup in a sequence or mapping is invalid or not found. It provides a common base class for exceptions that involve lookup failures, making it easier to catch and handle these types of errors.

Two main exceptions defined under LookupError are KeyError and IndexError.

KeyError

A KeyError is raised when a dictionary (or other mapping) does not contain the specified key.

Example:

```
try:

    my_dict = {'a': 1, 'b': 2, 'c': 3}

    value = my_dict['d']

except KeyError as e:

    print(f"KeyError: {e}")
```

IndexError

An IndexError is raised when an index used to access an element in a list (or other sequence) is out of the valid range.

Example:

```
try:

    my_list = [1, 2, 3]

    value = my_list[5]

except IndexError as e:

    print(f"IndexError: {e}")
```

5. Explain ImportError. What is ModuleNotFoundError?

ImportError

ImportError is a built-in exception in Python that is raised when an import statement fails to import a module or when a from ... import ... statement fails to find the name being imported in the module. This can happen for various reasons, such as:

1. The module does not exist.
2. The module name is misspelled.
3. The module is not installed.
4. The module is not in the Python path.

Example:

```
try:

    import non_existent_module

except ImportError as e:

    print(f"ImportError: {e}")
```

ModuleNotFoundError

ModuleNotFoundError is a subclass of ImportError introduced in Python 3.6. It specifically indicates that a module could not be found. This distinction helps in making the error handling more precise and clear, especially when debugging import issues.

Example:

```
try:

    import non_existent_module

except ModuleNotFoundError as e:

    print(f"ModuleNotFoundError: {e}")
```

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

Exception handling is an essential aspect of writing robust and maintainable code in Python. Here are some best practices for exception handling:

1. Catch Specific Exceptions

Why: Catching specific exceptions helps to handle different error conditions appropriately and avoids catching unintended exceptions.

Example:

```
try:

    result = 10 / 0

except ZeroDivisionError as e:

    print(f"ZeroDivisionError: {e}")

except ValueError as e:

    print(f"ValueError: {e}")
```

2. Use finally Block for Cleanup

Why: The finally block ensures that cleanup code is executed regardless of whether an exception was raised.

Example:

```
try:

    file = open('example.txt', 'r')

except IOError as e:

    print(f"IOError: {e}")

finally:

    file.close()
```