Q1. Explain why we have to use the Exception class while creating a Custom Exception. Note: Here Exception class refers to the base class for all the exceptions.

In object-oriented programming, exceptions are used to handle and manage exceptional conditions that may occur during the execution of a program. When an exceptional situation arises, such as an error or an unexpected event, it is important to provide a way to communicate and handle that situation effectively.

In many programming languages, including Java and Python, exceptions are represented by classes. These classes are organized in a hierarchy, where the base class for all exceptions is typically called the "Exception" class. This base class provides a foundation for defining and handling exceptions in a uniform manner.

When creating a custom exception, it is considered good practice to derive it from the base "Exception" class. Here are a few reasons why using the "Exception" class as the base for custom exceptions is beneficial:

1. Inheritance and Polymorphism: By inheriting from the "Exception" class, your custom exception can take advantage of the features and behavior provided by the base class. This includes functionalities like stack trace information, message handling, and exception chaining. In addition, using the base class as the parent allows your custom exception to be treated as a general exception type, enabling it to be caught by catch blocks that handle exceptions of the base class or any of its subclasses. This provides flexibility and consistency in exception handling.
2. Standardization and Readability: Deriving from the "Exception" class makes your custom exception adhere to a common convention followed in the programming community. Other developers who encounter your code will easily recognize that your class represents an exception and understand its purpose. It improves code readability and helps maintain a consistent and predictable exception hierarchy throughout your application.
3. Exception Handling: When you use the "Exception" class as the base, you can leverage the existing exception handling mechanisms provided by the programming language or framework. This includes try-catch blocks, exception propagation, and the ability to catch and handle specific types of exceptions. By following the standard exception handling practices, you can write cleaner, more maintainable code that gracefully handles exceptional scenarios.
4. Compatibility with Libraries and Tools: Many libraries, frameworks, and development tools are designed to work with exceptions that inherit from the "Exception" class. By conforming to this convention, your custom exception can seamlessly integrate with these existing tools and libraries, making it easier to handle and manage exceptions across your application.

In summary, using the "Exception" class as the base for custom exceptions provides a consistent and standardized approach to exception handling. It promotes code readability, inheritance, and compatibility with existing tools and libraries, allowing for efficient and effective management of exceptional situations in your programs.

Q2. 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 + 4)

print\_exception\_hierarchy(BaseException)

program defines a recursive function called print\_exception\_hierarchy that takes an exception class as input and prints its name. It then recursively calls itself for each subclass of the given exception class, increasing the indentation by 4 spaces for each level in the hierarchy.

In the main part of the program, we start the hierarchy by passing BaseException as the initial exception class to print\_exception\_hierarchy.

When we run this program, it will output the Python Exception Hierarchy, with each exception class indented to represent its position in the hierarchy.

Note: The program assumes that the subclasses of BaseException are available. In some cases, if the exception classes are defined dynamically or imported from external modules, they might not be discovered by \_\_subclasses\_\_(). In such cases, the hierarchy might not be complete.

Q3. What errors are defined in the ArithmeticError class? Explain any two with an example.

the ArithmeticError class is a base class for all exceptions that occur during arithmetic operations. It is a subclass of the Exception class and provides a common interface for handling arithmetic-related errors. Some of the errors defined in the ArithmeticError class include:

1. ZeroDivisionError: This error occurs when a division or modulo operation is performed with zero as the divisor. It indicates an attempt to divide a number by zero, which is mathematically undefined. Let's consider an example:

num1 = 10

num2 = 0

try:

result = num1 / num2

except ZeroDivisionError as e:

print("Error: Division by zero!")

In this example, the division operation num1 / num2 raises a ZeroDivisionError since num2 has a value of zero. The code within the except block will be executed, and the error message "Error: Division by zero!" will be printed.

1. OverflowError: This error occurs when a calculation exceeds the maximum representable value for a numeric type. It typically happens when performing operations that result in a value outside the range that can be represented by the data type. Here's an example:

import sys

max\_value = sys.maxsize # Maximum value for the current platform

try:

result = max\_value + max\_value

except OverflowError as e:

print("Error: Overflow occurred!")

In this example, we attempt to add max\_value to itself, which results in an arithmetic overflow. The OverflowError is raised because the result exceeds the maximum representable value. The code within the except block will be executed, and the error message "Error: Overflow occurred!" will be printed.

These are just two examples of errors defined in the ArithmeticError class. Other subclasses of ArithmeticError include FloatingPointError, UnderflowError, and OverflowError. Each of these exceptions captures specific types of arithmetic-related errors that can occur during program execution.

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

The LookupError class is used in Python to handle errors that occur when a lookup or indexing operation fails. It is a base class for specific lookup-related error classes, such as KeyError and IndexError, which are derived from it.

1. KeyError: This error occurs when you try to access a dictionary using a key that does not exist in the dictionary. Here's an example:

my\_dict = {'a': 1, 'b': 2, 'c': 3}

try:

value = my\_dict['d']

except KeyError as e:

print(f"KeyError: {e}")

Output:

KeyError: 'd'

In this example, we are trying to access the value associated with the key 'd', but it does not exist in the dictionary. As a result, a KeyError is raised, indicating that the key 'd' is not found.

1. IndexError: This error occurs when you try to access an element from a sequence (like a list or tuple) using an invalid index that is out of range. Here's an example:

my\_list = [1, 2, 3]

try:

value = my\_list[3]

except IndexError as e:

print(f"IndexError: {e}")

Output:

IndexError: list index out of range

In this example, we are trying to access the element at index 3 in the my\_list, but since the list has only three elements with indices 0, 1, and 2, accessing an element at index 3 is not possible. Therefore, an IndexError is raised, indicating that the index is out of range.

Both KeyError and IndexError are specific types of LookupError. By catching LookupError, you can handle both of these exceptions in a common way, or you can catch the specific exception types individually if you need to handle them differently.

Q5. Explain ImportError. What is ModuleNotFoundError?

Both ImportError and ModuleNotFoundError are exceptions that can occur when working with modules and importing them in Python.

ImportError is a base class for all import-related exceptions. It is raised when an import statement fails to find and load a module. This exception can occur due to various reasons, such as:

1. The module you are trying to import does not exist in the Python standard library.
2. The module is not installed in your Python environment.
3. There is an error in the module code that prevents it from being imported correctly.
4. The module is in a different location or has a different name than what you specified in the import statement

Here's an example of ImportError:

try:

import non\_existent\_module

except ImportError:

print("Module not found!")

ModuleNotFoundError is a subclass of ImportError and is specifically raised when a module is not found during the import process. It was introduced in Python 3.6 as a more specific exception to indicate that the module itself could not be found. Prior to Python 3.6, an ImportError would be raised for both missing modules and other import-related issues.

The difference between ImportError and ModuleNotFoundError lies in their specificity. ModuleNotFoundError helps to differentiate between an import error caused by a missing module and other import errors that may occur due to different reasons.

Here's an example of ModuleNotFoundError:

try:

import non\_existent\_module

except ModuleNotFoundError:

print("Module not found!")

In both cases, it's important to handle these exceptions appropriately to avoid program crashes and provide meaningful error messages to users when a required module is missing or cannot be imported.

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

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

1. Be specific with exception handling: Catch only the exceptions you expect and handle them appropriately. Avoid using a generic except clause, as it can hide errors and make debugging difficult.
2. Use multiple except blocks: When handling multiple exceptions, use separate except blocks for each exception rather than catching them all in a single block. This allows you to handle different exceptions differently and provides better error diagnostics.
3. Handle exceptions gracefully: When an exception occurs, handle it in a way that prevents crashes and provides meaningful feedback to the user. You can log the error, display a user-friendly message, or attempt to recover from the error if possible.
4. Use finally blocks: Use finally blocks to execute cleanup code that should always run, regardless of whether an exception occurred or not. Common use cases include releasing resources like file handles or closing database connections.
5. Avoid bare except blocks: Avoid using bare except blocks without specifying the exception type. This can catch unexpected errors and mask potential bugs in your code. Instead, catch specific exceptions or at least use a bare except block in combination with a logging mechanism to capture and investigate unexpected errors.
6. Handle exceptions at an appropriate level: Handle exceptions at the level where you can take appropriate action or provide useful feedback. Avoid catching exceptions too early and letting them propagate too far without being handled.
7. Use exception chaining: When catching an exception, you can raise a new exception with the original exception as a cause using the raise ... from ... syntax. This preserves the original exception's traceback and provides more information for debugging.
8. Use built-in exception hierarchy: Python has a hierarchy of built-in exceptions. When catching exceptions, try to catch specific exceptions first before more general ones. This helps in providing more targeted error handling and avoids catching and handling exceptions that should be propagated.
9. Avoid unnecessary try-except blocks: Only use try-except blocks when you expect an exception to occur. Don't use them to wrap large blocks of code unnecessarily, as it can make the code harder to read and maintain.
10. Test exception handling: Write tests to verify that your exception handling code works as intended. Include test cases that simulate different exception scenarios to ensure your code handles them correctly.