***Exception Handling -2***

**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 Python, the Exception class is the base class for all exceptions. When creating a custom exception, we inherit from the Exception class and define our own exception class.

Using the Exception class as the base class for our custom exception allows us to take advantage of the existing exception handling mechanisms in Python. For example, when we raise an exception that inherits from the Exception class, it can be caught by any except block that catches the base Exception class.

Additionally, the Exception class provides methods such as str() and repr() that can be overridden in our custom exception class to provide a string representation of the exception.

By using the Exception class as the base class for our custom exception, we can ensure that our exception behaves like a standard Python exception and can be handled in the same way as other built-in exceptions. This makes our code more maintainable and easier to understand for other developers who may be working with our code.

**Q2. Write a python program to print Python Exception Hierarchy.**

Python program to print the Python Exception Hierarchy:

def print\_exception\_hierarchy(exception\_class, indent=0):

"""Prints the hierarchy of the given exception class"""

print(" " \* indent + exception\_class.\_\_name\_\_)

for subclass in exception\_class.\_\_subclasses\_\_():

print\_exception\_hierarchy(subclass, indent + 2)

print\_exception\_hierarchy(BaseException)

This program defines a function called print\_exception\_hierarchy that takes an exception class as an argument and prints its name along with the names of all its subclasses. It recursively calls itself on each subclass to print its subclasses, and so on, until it has printed the entire hierarchy.

To print the entire Python Exception Hierarchy, we pass in the BaseException class, which is the base class for all built-in exceptions in Python. This program should output something like this:

BaseException

Exception

ArithmeticError

FloatingPointError

OverflowError

ZeroDivisionError

AssertionError

AttributeError

BufferError

EOFError

ImportError

ModuleNotFoundError

LookupError

IndexError

KeyError

MemoryError

NameError

UnboundLocalError

OSError

BlockingIOError

ChildProcessError

ConnectionError

BrokenPipeError

ConnectionAbortedError

ConnectionRefusedError

ConnectionResetError

FileExistsError

FileNotFoundError

InterruptedError

IsADirectoryError

NotADirectoryError

PermissionError

ProcessLookupError

TimeoutError

ReferenceError

RuntimeError

NotImplementedError

RecursionError

StopAsyncIteration

StopIteration

SyntaxError

IndentationError

TabError

SystemError

TypeError

ValueError

UnicodeError

UnicodeDecodeError

UnicodeEncodeError

UnicodeTranslateError

Warning

DeprecationWarning

PendingDeprecationWarning

RuntimeWarning

SyntaxWarning

UserWarning

FutureWarning

ImportWarning

UnicodeWarning

BytesWarning

ResourceWarning

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

**The** ArithmeticError **class is a built-in exception class in Python that is raised when an arithmetic operation fails. This class is a subclass of the** Exception **class and has several subclasses, which are all related to arithmetic errors. Some of the errors defined in the** ArithmeticError **class include:**

* FloatingPointError**: Raised when a floating-point operation fails, such as dividing a number by zero or taking the square root of a negative number.**
* ZeroDivisionError**: Raised when attempting to divide a number by zero.**

**Here are two examples to demonstrate these errors:**

**# Example 1: FloatingPointError**

**import math**

**# Taking the square root of a negative number raises a FloatingPointError**

**try:**

**x = math.sqrt(-1)**

**except FloatingPointError as e:**

**print("Error:", e)**

**# Output: Error: math domain error**

**# Example 2: ZeroDivisionError**

**# Dividing a number by zero raises a ZeroDivisionError**

**try:**

**x = 1 / 0**

**except ZeroDivisionError as e:**

**print("Error:", e)**

**# Output: Error: division by zero**

**In the first example, we try to take the square root of a negative number using the** math.sqrt() **function. This raises a** FloatingPointError **with the message "math domain error".**

**In the second example, we try to divide the number 1 by zero, which raises a** ZeroDivisionError **with the message "division by zero".**

**These errors are raised when there is an issue with the arithmetic operation being performed, and they can help us identify and fix the problem in our code.**

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

**The** LookupError **class is a built-in exception class in Python that is raised when an index or key lookup fails. It is a subclass of the** Exception **class and has several subclasses, including** IndexError **and** KeyError**.**

IndexError **is raised when an index is out of range for a sequence, such as a list or tuple. For example:**

**# Example 1: IndexError**

**lst = [1, 2, 3]**

**try:**

**x = lst[3] # accessing an out of range index**

**except IndexError as e:**

**print("Error:", e)**

**# Output: Error: list index out of range**

**In this example, we create a list with three elements and try to access the element at index 3, which is out of range. This raises an** IndexError **with the message "list index out of range".**

KeyError **is raised when a dictionary key is not found. For example:**

**# Example 2: KeyError**

**dict = {"a": 1, "b": 2, "c": 3}**

**try:**

**x = dict["d"] # accessing a non-existent key**

**except KeyError as e:**

**print("Error:", e)**

**# Output: Error: 'd'**

**In this example, we create a dictionary with three key-value pairs and try to access the value for the key "d", which does not exist in the dictionary. This raises a** KeyError **with the message "'d'".**

**These errors are subclasses of** LookupError **because they both occur when there is a failure to look up a value by index or key. By catching these specific exceptions, we can handle these errors in our code and provide more informative error messages to users.**

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

ImportError **is a built-in exception class in Python that is raised when an import statement fails to find a module or fails to find a name in a module. It is a subclass of the** Exception **class.**

**An** ImportError **can occur for several reasons, such as:**

* **The module or package does not exist or is not installed.**
* **The module or package is not located in a directory on the Python path.**
* **The module or package contains errors that prevent it from being imported.**
* **The name being imported is not defined in the module or package.**

**Here is an example to demonstrate** ImportError**:**

**# Example: ImportError**

**try:**

**import non\_existent\_module**

**except ImportError as e:**

**print("Error:", e)**

**# Output: Error: No module named 'non\_existent\_module'**

**In this example, we try to import a module that does not exist using the** import **statement. This raises an** ImportError **with the message "No module named 'non\_existent\_module'".**

ModuleNotFoundError **is a subclass of** ImportError **that was added in Python 3.6 to provide a more informative error message when a module cannot be found. It is raised when an import statement fails to find a module.** ModuleNotFoundError **is more specific than** ImportError **and provides additional information about the module that was not found.**

**Here is an example to demonstrate** ModuleNotFoundError**:**

**# Example: ModuleNotFoundError**

**try:**

**import non\_existent\_module**

**except ModuleNotFoundError as e:**

**print("Error:", e)**

**# Output: Error: No module named 'non\_existent\_module'**

**In this example, we try to import a module that does not exist using the** import **statement. This raises a** ModuleNotFoundError **with the same message as the** ImportError**: "No module named 'non\_existent\_module'".**

**In summary,** ImportError **is a built-in exception class in Python that is raised when an import statement fails to find a module or fails to find a name in a module, while** ModuleNotFoundError **is a more specific subclass of** ImportError **that provides a more informative error message when a module cannot be found.**

**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.Use specific exception types: Instead of catching a generic** Exception **type, use more specific exception types such as** ValueError**,** TypeError**,** IOError**, etc. This helps in writing cleaner code and also provides better error messages.**

**2.Use the try-except-else-finally block: The** try-except-else-finally **block is a powerful construct that allows you to handle exceptions gracefully. The** try **block contains the code that might raise an exception, the** except **block contains the code that handles the exception, the** else **block contains the code that executes when there are no exceptions, and the** finally **block contains the code that executes after the try block, regardless of whether an exception was raised or not.**

**3.Keep the try block minimal: Only include the code that might raise an exception in the** try **block. This makes it easier to identify where the exception occurred and also helps in writing cleaner code.**

**4.Avoid bare except clauses: Avoid using a bare** except **clause to catch all exceptions. This can lead to unexpected behavior and also makes it harder to identify and handle specific exceptions. Instead, catch specific exceptions or use** Exception **as a catch-all clause.**

**5.Use custom exception classes: Use custom exception classes to define your own exceptions for your application. This helps in writing more descriptive error messages and also makes it easier to identify and handle specific exceptions.**

**6.Handle exceptions where they occur: Handle exceptions as close to where they occur as possible. This makes it easier to identify where the exception occurred and also makes the code easier to read.**

**7.Log exceptions: Use a logging library to log exceptions. This helps in debugging and also provides valuable information for diagnosing issues.**

**8.Reraise exceptions: If you catch an exception and cannot handle it, reraise it using the** raise **statement. This allows the exception to propagate up the call stack, where it can be handled appropriately.**

**By following these best practices, you can write more robust and reliable code that is easier to maintain and debug.**