Assignment

Q1. Explain why we have to use the Exception class while creating a Custom Exception.
Ans:the `Exception` class serves as the base class for all exceptions. When creating a custom exception, it is advisable to inherit from the `Exception` class or one of its subclasses. Here are a few reasons why using the `Exception` class is recommended:

Inheritance from a Common Base Class:

- Inheriting from the `Exception` class ensures that your custom exception is part of the standard exception hierarchy in Python.
- It provides a consistent and standard way to handle exceptions, making your custom exception compatible with existing exception-handling mechanisms.

Compatibility with Exception Handling Mechanisms:

- Python's exception handling mechanism is designed to work with exceptions that derive from the `BaseException` class or its subclasses.
- When your custom exception is derived from the `Exception` class, it can be caught by a generic `except Exception` clause, allowing for a broad catch of exceptions.

Consistency and Best Practices:

- Following conventions and best practices makes your code more understandable for other developers.
- Inheriting from the `Exception` class is a widely accepted practice in Python, making your code more consistent with the broader Python community.

Built-in Methods and Attributes:

- The `Exception` class provides built-in methods and attributes that can be useful for customizing the behavior of your exception.
- For example, you can override the `__str__` method to customize the string representation of your exception.

Here's a simple example:

python

Copy code

```
class CustomError Exception
 def __init__
 super

try
 raise             "This is a custom exception."
except             as
 print f"Caught custom exception: {ce}"
```

```
except           as
 print f"Caught generic exception: {e}"
```

In this example, `CustomError` is derived from the `Exception` class. The `except CustomError` block catches the custom exception, and the `except Exception` block provides a more generic catch for any exception derived from `Exception`.

In summary, using the `Exception` class as the base class for custom exceptions ensures compatibility with the Python exception handling mechanism, promotes consistency in coding practices, and provides access to built-in methods and attributes that enhance the functionality of your custom exceptions.

Q2. Write a python program to print Python Exception Hierarchy.
Ans:can use the `__bases__` attribute to traverse the exception hierarchy in Python. Here's a simple program that prints the Python Exception Hierarchy:

python

Copy code

```
def print_exception_hierarchy                          0
 print ' '
 for          in
                                                  1
```

This program defines a recursive function `print_exception_hierarchy` that takes an exception class and prints its name, then recursively calls itself for each base class. The recursion continues until it reaches the base class `BaseException`.

When you run this program, it will print the hierarchy of built-in exceptions in Python:

php

Copy code

```
Exception
```

```
object
```

This output shows that `Exception` is a subclass of `BaseException`, and `BaseException` is a subclass of `object`. This is the basic hierarchy for exceptions in Python.

Keep in mind that the actual hierarchy may vary based on the version of Python you are using, as Python's exception hierarchy can be extended or modified in different releases. The hierarchy shown here is a simplified version.

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

Ans:The `ArithmeticError` class is a base class for exceptions that arise during arithmetic operations. It serves as a parent class for more specific arithmetic-related exceptions in Python. Two common exceptions derived from `ArithmeticError` are `ZeroDivisionError` and `OverflowError`. Let's explore these two exceptions with examples:

1. `ZeroDivisionError`:

This exception is raised when attempting to divide a number by zero.

Example:

python

Copy code

```
def divide_numbers

 try



 print "Result:"
```

```
except                    as

print "Error:"



            10

              0
```

In this example, the `divide_numbers` function attempts to perform a division operation. However, if the `denominator` is set to 0, a `ZeroDivisionError` will be raised, and the code in the `except` block will be executed.

2. `OverflowError`:

This exception is raised when an arithmetic operation exceeds the limits of the data type, resulting in an overflow.

Example:

python

Copy code

```
def calculate_factorial

 try
```

```
        1

for   in range 1      1



print "Factorial:"

except               as

print "Error:"




try

                   1000

except               as

print "OverflowError:"
```

In this example, the `calculate_factorial` function attempts to calculate the factorial of a number using a loop. However, when trying to calculate the factorial of a large number like 1000, it will result in an `OverflowError` because the result exceeds the limits of the data type.

Both of these examples demonstrate how specific arithmetic-related exceptions, derived from `ArithmeticError`, can be caught and handled to prevent unexpected program termination due to arithmetic errors.

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

Ans:The `LookupError` class is a base class for exceptions that occur when a key or index used to access a mapping or sequence is invalid. It serves as a parent class for more specific lookup-related exceptions in Python. Two common exceptions derived from `LookupError` are `KeyError` and `IndexError`.

1. `KeyError`:

This exception is raised when a dictionary key is not found.

Example:

python

Copy code

```
           "apple"  3  "banana"  5  "orange"  2



try



               "grape"

 print "Value:"

except          as

 print "KeyError:"
```

In this example, the dictionary `my_dict` does not have a key named "grape." When attempting to access this nonexistent key, a `KeyError` is raised, and the code in the `except` block is executed.

Handling `KeyError` allows the program to gracefully respond to situations where a requested key is not present in the dictionary.

2. `IndexError`:

This exception is raised when trying to access an index that is outside the bounds of a sequence (e.g., list, tuple).

Example:

python

Copy code

```
           10  20  30  40  50



try


                10

 print "Value:"

except           as

 print "IndexError:"
```

In this example, the list `my_list` has only five elements, and we attempt to access the element at index 10. Since the index is beyond the bounds of the list, an `IndexError` is raised, and the code in the `except` block is executed. Handling `IndexError` allows the program to manage situations where an invalid index is used to access a sequence.

In both cases, catching exceptions derived from `LookupError` (either `KeyError` or `IndexError`) allows for more specific and targeted error handling based on the type of lookup-related issue that occurred.

Q5. Explain ImportError. What is ModuleNotFoundError?

Ans:`ImportError` is a base class for exceptions that occur when the import statement fails to locate and load a module. It is a broad exception that can encompass various issues related to importing modules.

`ImportError` Example:

python

Copy code

```
try

 import
except             as
 print "ImportError:"
```

In this example, an attempt is made to import a module named `non_existent_module`, which does not exist. As a result, an `ImportError` is raised, and the code in the `except` block is executed.

`ModuleNotFoundError`:

`ModuleNotFoundError` is a more specific subclass of `ImportError`. It is raised when the specified module cannot be found during the import process.

`ModuleNotFoundError` Example:

python

Copy code

```
try

 import
except                    as
 print "ModuleNotFoundError:"
```

In this example, the same attempt to import the non-existent module is made, but the exception caught is `ModuleNotFoundError` instead of `ImportError`. `ModuleNotFoundError` provides more specific information about the nature of the import failure, indicating that the module could not be found.

While both `ImportError` and `ModuleNotFoundError` can be used to handle import-related issues, using `ModuleNotFoundError` is more precise and allows for more targeted exception handling when the primary concern is the absence of a specific module. It is important to note that `ModuleNotFoundError` was introduced in Python 3.6, and if you are working with an earlier version, you might encounter `ImportError` for module-not-found situations.

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

Ans:Exception handling is a crucial aspect of writing robust and maintainable Python code. Here are some best practices for effective exception handling in Python:

Be Specific with Exception Types:

- Catch specific exceptions rather than using broad `except` clauses. This helps in providing more targeted and appropriate handling for different types of errors.

python

Copy code

```
try

except                    as

except                          as
```

Avoid Using a Blanket `except` Clause:

- Refrain from using a generic `except` clause without specifying the exception type. This can make it challenging to identify and troubleshoot issues.

python

Copy code

```
try

except          as
```

Use `else` Block Sparingly:

- The `else` block after a `try` block should contain code that runs only if no exceptions were raised. Use it sparingly to avoid introducing unnecessary complexity.

python

Copy code

```
try

except                as

else
```

Use `finally` for Cleanup:

- The `finally` block is executed whether an exception occurs or not. It's useful for cleanup operations like closing files or releasing resources.

python

Copy code

```
try

except                as

finally
```

Log Exceptions:

- Use logging to record information about exceptions. Logging allows for better debugging and understanding of issues in production environments.

python

Copy code

```
import
```

```
try

except               as
               f"An exception occurred: {se}"
```

Raise Exceptions at the Right Level:

- Raise exceptions at the level where they can be appropriately handled. Avoid catching exceptions at a level where meaningful action cannot be taken.

python

Copy code

```
def process_data
 if not
 raise            "Data cannot be empty"
```

Handle Exceptions Locally:

- Handle exceptions as close to the source of the error as possible. This improves code readability and helps in isolating and addressing issues.

python

Copy code

```
def read_file
 try
 with open            'r'   as

 except
 print f"File not found: {file_path}"
 except         as
 print f"IO error: {ioe}"
```

Use Context Managers (`with` statement):

- Utilize context managers, especially when working with resources like files or database connections. They ensure proper cleanup even if exceptions occur.

python

Copy code

```
with open "example.txt"  "r"  as
```

Document Exception Handling:

- Provide comments or docstrings to document the rationale behind specific exception-handling decisions. This helps other developers understand the intent.

python

Copy code

```
try

except               as
```

Keep Exception Handling Concise:

- Avoid overly complex exception-handling logic. Keep it concise and focused on addressing the specific error condition.

python

Copy code

```
try

except               as

except                  as
```

By following these best practices, you can enhance the robustness, readability, and maintainability of your Python code, making it easier to identify and address issues when they arise.