Q1. Describe three applications for exception processing.

ANS: Exception processing is a powerful feature in Python that allows you to handle and manage unexpected or exceptional situations that may arise during program execution. Here are three common applications for exception processing:

a). Error Handling: Exception processing is primarily used for error handling in Python. When certain operations encounter errors or exceptional conditions, they raise exceptions. By using try-except blocks, you can catch and handle these exceptions gracefully, preventing the program from crashing and providing useful error messages to users or developers.

Example - Division by Zero:

def divide(a, b):

try:

result = a / b

return result

except ZeroDivisionError:

return "Error: Cannot divide by zero."

print(divide(10, 0)) # Output: Error: Cannot divide by zero.

b). File Handling: When reading from or writing to files, exceptions can occur due to various reasons, such as file not found, permission issues, or I/O errors. By using try-except blocks, you can handle these exceptions and take appropriate actions, such as creating a new file or informing the user about the issue.

Example - File Not Found:

file\_name = "example.txt"

try:

with open(file\_name, 'r') as file:

contents = file.read()

print(contents)

except FileNotFoundError:

print(f"Error: File '{file\_name}' not found.")

c). Data Validation: When working with user inputs or external data, it is crucial to validate the data before processing it. Exception processing can be used to validate data and handle invalid inputs or data that do not meet certain criteria. This helps ensure the program's stability and correctness.

Example - Integer Input Validation:

def get\_positive\_integer():

try:

num = int(input("Enter a positive integer: "))

if num <= 0:

raise ValueError("Invalid input: Enter a positive integer.")

return num

except ValueError as e:

print(e)

return get\_positive\_integer()

number = get\_positive\_integer()

print(f"You entered: {number}")

Q2. What happens if you don't do something extra to treat an exception?

ANS: If you don't do something extra to treat an exception, i.e., if you do not handle the exception using try-except blocks or any other mechanism, the unhandled exception will propagate up the call stack until it reaches the top-level of the program. When an unhandled exception reaches the top-level, it will cause the program to terminate abruptly, and an error message (traceback) will be displayed, indicating the type of the exception and the line of code where the exception occurred.

The error message includes:

I) The type of the exception (e.g., ZeroDivisionError, FileNotFoundError, ValueError, etc.).

II) A traceback, which is a list of calls made by the program, starting from the location of the exception to the original call that caused the exception.

III) The line number and file name where the exception occurred.

Here's an example of an unhandled exception:

def divide(a, b):

result = a / b

return result

# Unhandled exception - Division by Zero

result = divide(10, 0)

Output:

ZeroDivisionError: division by zero

In this example, the `divide()` function attempts to divide `10` by `0`, which raises a `ZeroDivisionError`. Since there is no try-except block or any other exception handling mechanism, the exception propagates up to the top-level of the program, causing the program to terminate with an error message.

Q3. What are your options for recovering from an exception in your script?

ANS: When an exception occurs in your Python script, you have several options for recovering from it and handling the exceptional situation. These options are provided by exception handling mechanisms, primarily using try-except blocks. Here are some common ways to recover from an exception:

1. Handling the Exception: Use a try-except block to catch the exception and handle it gracefully. Inside the except block, you can perform specific actions, provide meaningful error messages to the user, or take corrective measures to recover from the exceptional situation.

try:

# Code that may raise an exception

except SomeException as e:

# Handle the exception

print(f"An error occurred: {e}")

# Take corrective actions or provide fallback behavior

1. Using Multiple Except Blocks: If you anticipate different types of exceptions, you can use multiple except blocks to handle each type of exception differently.

try:

# Code that may raise different types of exceptions

except ZeroDivisionError as e:

# Handle the ZeroDivisionError

print("Cannot divide by zero.")

except FileNotFoundError as e:

# Handle the FileNotFoundError

print("File not found.")

except SomeOtherException as e:

# Handle other specific exception types

1. Using a Generic Except Block: You can use a generic except block to catch any unanticipated exceptions that are not explicitly handled by other except blocks. However, it is generally recommended to avoid using a completely generic except block, as it may hide potential issues in the code.

try:

# Code that may raise an exception

except Exception as e:

# Handle any unanticipated exceptions

print(f"An unexpected error occurred: {e}")

# Take corrective actions or provide fallback behavior

1. Using Finally Block: A finally block can be used to define cleanup actions that must be executed, whether an exception occurs or not. It allows you to ensure that certain code, like resource release or cleanup, is executed, regardless of whether an exception was raised or not.

try:

# Code that may raise an exception

except SomeException as e:

# Handle the exception

print(f"An error occurred: {e}")

finally:

# Cleanup actions

print("Finally block executed.")

Q4. Describe two methods for triggering exceptions in your script.

ANS: In Python, you can trigger exceptions intentionally in your script using two main methods: raising exceptions with the `raise` statement and using the `assert` statement. These methods allows us to explicitly signal exceptional conditions or errors based on certain conditions or criteria defined in code.

A} Raising Exceptions with `raise`: The `raise` statement is used to raise exceptions explicitly in your script. It allows you to indicate that an exceptional condition has occurred and provides information about the error. You can raise built-in exceptions or create custom exceptions to handle specific cases.

Example - Raising a Custom Exception:

def calculate\_square\_root(number):

if number < 0:

raise ValueError("Cannot calculate square root of a negative number.")

return number \*\* 0.5

try:

result = calculate\_square\_root(-4)

except ValueError as e:

print(e) # Output: Cannot calculate square root of a negative number.

In this example, the `calculate\_square\_root()` function raises a `ValueError` if the input `number` is negative. The `raise` statement is used to trigger the exception explicitly, and the error message is provided for informative purposes.

B} Using `assert` Statement: The `assert` statement is used to perform simple sanity checks or to validate assumptions in your code. It raises an `AssertionError` exception if the expression it evaluates is False. It is typically used during development and testing to detect and fix issues early.

Example - Using `assert` for Data Validation:

def divide(a, b):

assert b != 0, "Cannot divide by zero."

return a / b

try:

result = divide(10, 0)

except AssertionError as e:

print(e) # Output: Cannot divide by zero.

In this example, the `divide()` function uses `assert` to check if the divisor `b` is not zero. If `b` is zero, it raises an `AssertionError` with the message "Cannot divide by zero."

Both of these methods allow you to intentionally raise exceptions based on specific conditions, errors, or assumptions in your code. Raising explicit exceptions enables you to handle exceptional situations gracefully and provide meaningful feedback to users or developers about what went wrong. When raising custom exceptions, you can define exception classes that carry additional information, making error handling more informative and flexible.

Q5. Identify two methods for specifying actions to be executed at termination time, regardless of whether or not an exception exists.

ANS: In Python, we can specify actions to be executed at termination time, regardless of whether or not an exception exists, using two methods: the `finally` block and the `atexit` module.

1. Using the `finally` Block: The `finally` block is used to define a section of code that will always be executed, regardless of whether an exception occurred or not. It is typically used for cleanup operations or to ensure that certain code must be executed, regardless of the program's flow.

def divide(a, b):

try:

result = a / b

print("Division successful!")

return result

except ZeroDivisionError:

print("Cannot divide by zero.")

return None

finally:

print("Finally block executed.")

result1 = divide(10, 2)

result2 = divide(10, 0)

Output:

Division successful!

Finally block executed.

Cannot divide by zero.

Finally block executed.

In this example, the `finally` block contains the code that prints "Finally block executed." This code is executed after either the try block (if no exception occurs) or the except block (if an exception occurs) finishes its execution.

1. Using the `atexit` Module: The `atexit` module allows you to register functions to be called automatically at program termination, regardless of whether the termination is normal or due to an unhandled exception. This module is helpful for performing cleanup actions or saving state before the program exits.

import atexit

def cleanup():

print("Performing cleanup before exit.")

atexit.register(cleanup)

# Rest of the program...

Output (when the program exits):

Performing cleanup before exit.

In this example, the `cleanup()` function is registered using `atexit.register(cleanup)`. When the program exits, whether it exits normally or due to an unhandled exception, the `cleanup()` function will be automatically called.

Both the `finally` block and the `atexit` module provide mechanisms to ensure that specific code is executed at program termination, regardless of exceptional conditions. The `finally` block is more suitable for handling cleanup actions within a specific code block, while the `atexit` module is ideal for registering functions that need to be executed at the very end of the program's execution.