Q1. Describe three applications for exception processing.

A1. Exception processing in programming is used to handle errors or unexpected situations that arise during the execution of a program. It allows a program to respond to errors gracefully without crashing. Here are three common applications for exception processing:

**1. Error Handling in User Input**

**Application**: Validating and handling user input to ensure that the program continues to function even if the user provides invalid or unexpected data.

**Example**: In a financial application where users enter amounts of money, exceptions can be used to handle cases where the input is not a valid number.

def get\_user\_input():

while True:

try:

amount = float(input("Enter an amount: "))

if amount < 0:

raise ValueError("Amount cannot be negative.")

return amount

except ValueError as e:

print(f"Invalid input: {e}. Please enter a valid number.")

amount = get\_user\_input()

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

**Purpose**: Ensures that the program handles invalid or incorrect input from users without crashing and prompts the user to enter valid data.

**2. File Operations and Resource Management**

**Application**: Handling errors related to file operations, such as reading from or writing to files, and managing system resources such as network connections.

**Example**: Opening a file for reading and handling cases where the file might not exist or might be inaccessible.

def read\_file(filename):

try:

with open(filename, 'r') as file:

content = file.read()

return content

except FileNotFoundError:

print(f"File {filename} not found.")

except IOError as e:

print(f"An error occurred while reading the file: {e}")

content = read\_file('example.txt')

if content:

print(content)

**Purpose**: Ensures that file operations are managed properly, providing informative messages when errors occur and preventing the program from crashing due to file-related issues.

**3. Network Communication**

**Application**: Handling errors that occur during network communication, such as connecting to a server, sending data, or receiving responses.

**Example**: Connecting to a remote server and handling exceptions related to network connectivity issues.

import requests

def fetch\_data(url):

try:

response = requests.get(url)

response.raise\_for\_status() # Raises HTTPError for bad responses (4xx or 5xx)

return response.json()

except requests.ConnectionError:

print("Network connection error. Please check your internet connection.")

except requests.HTTPError as e:

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

except requests.RequestException as e:

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

data = fetch\_data('https://api.example.com/data')

if data:

print(data)

**Purpose**: Ensures that network-related errors are handled gracefully, such as connection problems or server errors, and provides informative feedback to the user or developer.

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

A2. If you don't handle an exception, it will propagate up the call stack, potentially causing the program to terminate abruptly and resulting in an error message or traceback. This can lead to program crashes, unpredictable behavior, poor user experience, and difficulty in debugging.

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

A3. To recover from an exception in your script, you have several options:

1. **Exception Handling with try and except**: Use a try block to execute code that may raise an exception, and except blocks to handle specific exceptions and take corrective actions.

try:

# Code that may raise an exception

except SomeException as e:

# Handle the exception

1. **Fallback Logic**: Implement fallback logic in the except block to provide alternative actions or default values when an exception occurs.

try:

result = risky\_operation()

except SomeException:

result = default\_value

1. **Retry Mechanism**: Retry the operation that caused the exception, possibly with a delay or after a certain number of retries.

import time

for \_ in range(max\_retries):

try:

result = risky\_operation()

break # Exit loop if successful

except SomeException:

time.sleep(delay)

1. **Logging and Reporting**: Log the exception details to a file or monitoring system for later analysis, and optionally notify users or administrators.

import logging

try:

# Code that may raise an exception

except SomeException as e:

logging.error(f"An error occurred: {e}")

1. **Graceful Degradation**: Modify the behavior of the script to continue operating with reduced functionality when an exception occurs.

try:

perform\_critical\_operation()

except SomeException:

perform\_alternative\_operation()

Each option allows you to handle exceptions in a way that minimizes disruption and maintains program stability.

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

A4. You can trigger exceptions in your script using the following methods:

**1. Using the raise Statement**

The raise statement allows you to manually trigger an exception by specifying the exception type. You can also provide an optional error message or arguments to the exception.

**Example**:

def divide(x, y):

if y == 0:

raise ValueError("Cannot divide by zero")

return x / y

try:

result = divide(10, 0)

except ValueError as e:

print(f"Error: {e}")

**Description**: In this example, the raise statement triggers a ValueError if the divisor (y) is zero, with a message indicating the problem. The exception is then caught and handled in the except block.

**2. Using the assert Statement**

The assert statement is used to test conditions and trigger an AssertionError if the condition evaluates to False. It is often used for debugging purposes to catch conditions that should never occur.

**Example**:

def positive\_divide(x, y):

assert y > 0, "The divisor must be positive"

return x / y

try:

result = positive\_divide(10, -2)

except AssertionError as e:

print(f"Assertion failed: {e}")

**Description**: In this example, the assert statement checks if the divisor (y) is positive. If y is not positive, it raises an AssertionError with the provided message. The exception is then caught and handled in the except block.

Both methods allow you to control and manage error conditions in your script, providing a way to handle unexpected or invalid states.

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

A5. To specify actions that should be executed at termination time, regardless of whether an exception occurs, you can use the following methods:

**1. Using the finally Block**

The finally block is used in conjunction with try and except blocks to ensure that specific code is executed no matter what—whether an exception is raised or not. This is useful for cleanup actions such as closing files or releasing resources.

**Example**:

try:

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

# Perform file operations

finally:

file.close() # This will execute whether an exception occurred or not

**Description**: In this example, the finally block ensures that the file is closed even if an exception occurs during file operations.

**2. Using Context Managers (with Statement)**

Context managers are used to manage resources such as files, network connections, or locks. They ensure that resources are properly acquired and released, using the \_\_enter\_\_ and \_\_exit\_\_ methods. The with statement automatically handles the setup and teardown of the context.

**Example**:

class Resource:

def \_\_enter\_\_(self):

# Setup code

return self

def \_\_exit\_\_(self, exc\_type, exc\_value, traceback):

# Cleanup code

pass

with Resource() as resource:

# Use the resource

pass

**Description**: In this example, the \_\_exit\_\_ method of the Resource class is called when the with block is exited, regardless of whether an exception was raised. This ensures that cleanup actions are performed properly.

Both methods provide mechanisms for executing code at termination time, ensuring that resources are properly managed and cleanup actions are performed even in the presence of exceptions.