Q1. Describe three applications for exception processing.

A) Exception processing, which involves handling unexpected or exceptional situations in software, is crucial for ensuring robustness and reliability in applications. Here are three common applications:

Error Handling: Exception processing is extensively used for error handling in software systems. When a program encounters an error condition, such as invalid input, file not found, or database connection failure, it raises an exception. Exception handlers then catch these exceptions and take appropriate actions, such as logging the error, displaying a user-friendly error message, or gracefully terminating the program.

Resource Management: Exception processing is also vital for managing resources efficiently, especially in systems where resources like files, database connections, or network sockets need to be acquired and released properly. Exceptions ensure that resources are released even in the presence of errors or exceptions. For example, if an exception occurs while writing to a file, an exception handler can ensure that the file is closed to prevent resource leaks.

Transaction Management: In applications that involve transactions, such as banking systems or e-commerce platforms, exception processing is essential for ensuring data consistency and integrity. If an error occurs during a transaction, such as a database operation failing or a network error, the transaction needs to be rolled back to maintain data consistency. Exception handling mechanisms are used to catch these errors and initiate rollback procedures to restore the system to a consistent state.

These are just a few examples of how exception processing is applied in software development to enhance reliability, maintainability, and robustness of applications.

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

A) If an exception is not properly handled or treated in a software application, several undesirable consequences can occur:

Program Crashes: If an exception is not caught and handled by an exception handler, it will propagate up the call stack until it reaches the top-level of the program. In many programming languages, an unhandled exception at the top-level of the program will cause the program to terminate abruptly. This results in a crash, and the application may display an error message to the user or simply exit without warning.

Resource Leaks: If exceptions occur during the acquisition or manipulation of resources (such as file handles, database connections, or memory allocations) and these exceptions are not properly handled, the associated resources may not be released or cleaned up. This can lead to resource leaks, where system resources are consumed and not returned to the system, eventually depleting available resources and potentially causing system instability or failure.

Data Corruption: In systems where transactions are used to maintain data consistency and integrity, failing to properly handle exceptions can lead to data corruption. For example, if an exception occurs during a database transaction and the transaction is not rolled back, the database may be left in an inconsistent state where some operations are applied while others are not. This can lead to data integrity issues and compromise the reliability of the system.

Security Vulnerabilities: Unhandled exceptions can also pose security risks to an application. If sensitive information, such as stack traces or error messages containing system details, is exposed to users or attackers, it may provide valuable insights into the inner workings of the application and potential vulnerabilities that can be exploited.

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

A) When encountering an exception in a script, you have several options for recovering from it and continuing the execution of the script:

Exception Handling with Try-Except Blocks: This is the most common approach to handle exceptions in scripts. You enclose the code that might raise an exception within a try block, and then specify one or more except blocks to catch and handle specific exceptions. Within the except block, you can perform recovery actions or provide alternative logic to handle the exception gracefully.

try:

# Code that may raise an exception

result = 10 / 0

except ZeroDivisionError:

# Handle the specific exception (ZeroDivisionError in this case)

print("Error: Division by zero occurred.")

# Perform recovery actions or alternative logic here

Exception Handling with Try-Except-Else Blocks: You can use the else block along with try-except to execute code that should run only if no exceptions occur within the try block. This is useful for separating the normal execution path from the exception handling logic.

try:

# Code that may raise an exception

result = 10 / 5

except ZeroDivisionError:

# Handle the specific exception (ZeroDivisionError in this case)

print("Error: Division by zero occurred.")

# Perform recovery actions or alternative logic here

else:

# Code to execute if no exceptions occur

print("Division successful. Result:", result)

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

A) In Python, you can trigger exceptions in your script using various methods to handle error conditions or exceptional situations. Here are two common methods for triggering exceptions:

Explicitly Raise an Exception: You can explicitly raise exceptions in your script using the raise statement. This allows you to indicate that an error condition has occurred and halt the normal flow of execution.

def divide(x, y):

if y == 0:

# Raise a ZeroDivisionError if the denominator is zero

raise ZeroDivisionError("Division by zero is not allowed.")

return x / y

try:

result = divide(10, 0)

print("Result:", result)

except ZeroDivisionError as e:

print("Error:", e)

# Handle the exception or perform recovery actions here

In this example, the divide function raises a ZeroDivisionError if the denominator y is zero. This exception is then caught and handled in the except block.

Use Built-in Functions or Operations that Raise Exceptions: Many built-in functions or operations in Python can raise exceptions in certain error conditions. For example, attempting to open a non-existent file with the open() function will raise a FileNotFoundError.

try:

file = open("nonexistent\_file.txt", "r")

# Attempt to open a non-existent file

# This will raise a FileNotFoundError

except FileNotFoundError as e:

print("Error:", e)

# Handle the exception or perform recovery actions here

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

A) In Python, you can specify actions to be executed at termination time, regardless of whether or not an exception exists, using the following methods:

Using a Finally Block: The finally block is used in conjunction with a try-except block to define cleanup actions that should be executed regardless of whether an exception occurs or not. Code within the finally block will always run, even if an exception is raised within the try block or if an exception is caught and handled in the except block.

try:

# Code that may raise an exception

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

# Perform operations on the file

except FileNotFoundError:

print("Error: File not found.")

# Handle the exception

finally:

# Cleanup actions to be executed regardless of exceptions

file.close() # Close the file to release resources

print("Cleanup complete.")

In this example, the finally block ensures that the file is closed, releasing system resources, even if an exception occurs while trying to open the file or during the execution of code within the try block.

Using the atexit Module: The atexit module provides a way to register functions that should be executed when the Python interpreter exits. These functions are executed regardless of whether the program exits normally or due to an unhandled exception. You can register cleanup functions using the atexit.register() function.

import atexit

def cleanup():

print("Performing cleanup actions...")

atexit.register(cleanup)