Q1. Describe three applications for exception processing.

Sol:-

Error Handling: Exceptions provide a structured way to handle errors and exceptional situations that may arise during the execution of a program. By using try-except blocks, you can catch specific types of exceptions and gracefully handle them. This prevents the program from abruptly terminating and allows you to provide meaningful error messages or perform recovery actions. Error handling is essential for writing resilient code that can gracefully handle unexpected situations, such as input validation errors, file access failures, network communication issues, or unexpected data conditions.

Resource Management: Exceptions are useful for managing resources, such as files, database connections, network sockets, or system locks. By using try-finally or try-except-finally blocks, you can ensure that resources are properly released or cleaned up, even in the event of an exception. The finally block guarantees that certain code is executed regardless of whether an exception occurs or not. This is particularly useful when dealing with resources that need explicit closing or releasing, as it helps prevent resource leaks and ensures proper cleanup.

Program Flow Control: Exceptions can be used to control the flow of a program and handle specific scenarios that require branching based on exceptional conditions. By raising custom exceptions and catching them at appropriate places in the code, you can guide the program's execution based on specific conditions or trigger specific actions. This allows you to implement complex error handling logic or implement alternative paths in your code based on different exceptional situations. Exception-based flow control can help improve code readability, maintainability, and flexibility.

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

Sol:-

Here's what happens when an exception is not explicitly handled:

Propagation: When an exception is raised within a block of code (inside a function, method, or a specific code section within a try block), the normal flow of execution is interrupted, and the exception is propagated upwards in search of an exception handler.

Call Stack Unwinding: As the exception propagates up the call stack, each function or method that is in the call chain is unwound, meaning that the function's local state is cleaned up and control is transferred back to the calling function. This continues until an appropriate exception handler is found.

Exception Handler Search: The exception is propagated up until it encounters a matching except block that can handle the specific exception type. If a matching except block is found, the code within that block is executed, and the program continues running from that point. If no matching except block is found, the exception continues to propagate.

Termination: If the exception reaches the top-level of the program without being caught by any except block, the default behavior is for the program to terminate. An error message is displayed, indicating the type of exception that occurred and a traceback that shows the sequence of function calls leading to the unhandled exception.

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

Sol:-

Here are some common approaches for recovering from exceptions:

Exception Handling: The most common and recommended approach is to use try-except blocks to catch and handle specific exceptions. You can specify the types of exceptions you want to catch and define the actions or code to be executed when those exceptions occur. By handling exceptions, you can gracefully recover from the exceptional situation, perform necessary cleanup or error recovery actions, and continue the execution of your script.

Retry Mechanism: In some cases, it may be appropriate to retry the operation that raised the exception. For example, if you're making a network request and encounter a connection error, you can implement a retry mechanism to attempt the request again after a certain delay or a certain number of retries. This can be useful when the exception is transient and may resolve itself after a certain period.

Alternative Code Paths: Depending on the specific situation, you may have alternative code paths or fallback strategies to handle exceptions. For example, if a file cannot be found, you can provide a default value or use a different file instead. By designing your code to have alternative paths for exceptional scenarios, you can continue the execution with different actions or behavior, ensuring that your script can recover from unexpected situations.

Graceful Termination: In some cases, it may not be possible or desirable to recover from an exception. In such cases, you can gracefully terminate the script by performing necessary cleanup operations and logging the exception details. Graceful termination ensures that any resources used by the script are properly released and any necessary finalization steps are executed.

Logging and Error Reporting: Regardless of the recovery strategy chosen, it is essential to log the details of the exception. Logging allows you to capture relevant information about the exception, including the error message, traceback, and any other relevant context. Logging facilitates debugging and helps in understanding the cause of the exception, enabling you to improve the reliability and stability of your script.

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

Sol:-

Here are two common methods for triggering exceptions:

raise Statement: The raise statement allows you to explicitly raise an exception in your code. You can raise a built-in exception class or create custom exception classes to suit your needs. The raise statement is followed by the type of exception to be raised, optionally accompanied by an error message or additional information. Here's an example:

# Raising a built-in exception

raise ValueError("Invalid input")

# Raising a custom exception

class MyException(Exception):

pass

raise MyException("Something went wrong")

Assertion: Another method to trigger exceptions is by using assertions. Assertions are statements that assert or validate certain conditions to be true. If an assertion fails and the condition is evaluated as false, an AssertionError exception is raised. Assertions are typically used to check for conditions that should always be true during the execution of your script. If the condition is violated, an exception is raised, indicating a logical error in your code. Here's an example:

x = 10

y = 5

assert y != 0, "Divisor cannot be zero"

result = x / y

print(result)

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

Sol:-

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

try:

# Code that may raise an exception

# ...

finally:

# Code that always executes, regardless of exception

# Cleanup or finalization actions

# ...

Context Managers: Context managers provide a more structured way to specify actions to be executed at termination time. They allow you to define setup and teardown code blocks using the with statement. The context manager pattern is implemented using the \_\_enter\_\_() and \_\_exit\_\_() methods. The \_\_enter\_\_() method sets up the context, and the \_\_exit\_\_() method defines the termination actions. Here's an example:

class MyContextManager:

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

# Setup code

# ...

def \_\_exit\_\_(self, exc\_type, exc\_val, exc\_tb):

# Teardown code

# Cleanup or finalization actions

# ...

# Usage

with MyContextManager():

# Code that may raise an exception

# ...