Q1. Describe three applications for exception processing.

ANSWER.

Exception processing is a versatile tool that enables developers to write robust and reliable software by handling errors gracefully, managing resources efficiently, and validating input effectively. By incorporating exception handling into their code, developers can improve the reliability, usability, and maintainability of their applications.

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

ANSWER.

Failing to handle exceptions in your code can lead to program termination, unpredictable behavior, and error messages being displayed to the user. It's essential to properly handle exceptions to ensure robustness, reliability, and user-friendliness in your applications.

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

ANSWER.

When recovering from an exception in your Python script, you have several options depending on the nature of the exception and the requirements of your application. Here are some common strategies for recovering from exceptions:

1. Try-Except Blocks:

- Use `try`...`except` blocks to catch and handle specific exceptions that may occur within a block of code. You can specify one or more `except` clauses to handle different types of exceptions separately.

2. Logging:

- Use the logging module to log information about exceptions, including error messages, stack traces, and context information. Logging exceptions can help diagnose issues in production environments and provide valuable insights for troubleshooting.

3. Graceful Degradation:

- Implement fallback mechanisms or alternative paths of execution to gracefully handle exceptions and continue program execution. This approach allows the program to recover from errors and continue functioning even if certain operations fail.

4. Retry Logic:

- Implement retry logic to retry failed operations multiple times before giving up. This approach can be useful for dealing with transient errors or network-related issues that may resolve themselves after a short delay.

5. Graceful Exit:

- If an exception cannot be recovered or if continuing execution would lead to unexpected behavior, it may be appropriate to exit the program gracefully and provide appropriate error messages to the user.

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

ANSWER.

In Python, you can trigger exceptions deliberately using various methods to handle exceptional cases or simulate error conditions during testing. Here are two common methods for triggering exceptions in your script:

1. \*\*Raise Statement\*\*:

- The `raise` statement is used to explicitly raise exceptions in Python. You can raise built-in exceptions or create custom exception classes to signal errors or exceptional conditions in your code.

- Syntax:

```python

raise ExceptionType("Optional error message")

```

2. AssertionError:

- The `assert` statement is used to test conditions that should always be true. If the condition evaluates to `False`, an `AssertionError` exception is raised.

- Syntax:

```python

assert condition, "Optional error message"

```

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

ANSWER.

Two methods for specifying actions to be executed at termination time, regardless of whether or not an exception exists, are using the `finally` block and context managers (`with` statement) in Python.

1. Using `finally` Block:

- The `finally` block in a `try`...`except`...`finally` construct allows you to specify code that should be executed regardless of whether an exception occurs or not. The `finally` block is always executed, even if an exception is raised and caught or if control flows out of the `try` block due to a `return`, `break`, or `continue` statement.

- Syntax:

```python

try:

# Code that may raise an exception

except ExceptionType:

# Exception handling code

finally:

# Code to be executed regardless of exceptions

```

2. Using Context Managers (`with` Statement):

- Context managers in Python, implemented using the `with` statement, provide a convenient way to manage resources and ensure cleanup actions are performed at termination time. Context managers automatically call the `\_\_enter\_\_` and `\_\_exit\_\_` methods of an object, allowing you to specify setup and teardown actions.

- Syntax:

```python

with expression as variable:

# Code block

```