**Q1. Describe three applications for exception processing.**

1. Input Validation: Exception handling can be used to validate user input and ensure that the program only processes valid data. For example, if a program asks the user to enter a number, it can catch the exception if the user enters something other than a number, and prompt them to enter a valid input.
2. File Handling: Exception handling can be used in file operations to handle potential errors such as file not found or insufficient permissions to access the file. This way, the program can catch these exceptions and provide an appropriate error message or take an alternative course of action.
3. Network Operations: Exception handling can be used in network operations to handle potential errors such as a broken connection or a server not responding. This way, the program can catch these exceptions and handle them gracefully, for example, by retrying the operation or providing a suitable error message to the user.

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

If you don't handle an exception in Python, the program will terminate and an error message will be displayed to the user. This is known as an unhandled exception, and it will cause the program to crash. The error message will contain information about the type of exception that was raised and its location in the code. This information can be useful for debugging the issue, but it can also be confusing for the end user.

For example, if you try to open a file that doesn't exist, the following error message will be displayed:

FileNotFoundError: [Errno 2] No such file or directory: 'nonexistent\_file.txt'

In order to prevent the program from crashing, it is best to handle exceptions using a try-except block. This way, you can catch the exception and handle it gracefully, for example by printing an error message to the user or taking an alternative course of action.

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

When an exception occurs in a Python script, you have several options for recovering from the exception:

1. Graceful Error Handling: You can catch the exception using a try-except block and handle it in a way that allows the rest of the script to continue executing. For example, you can display an error message to the user or take an alternative course of action.
2. Debugging: You can use a try-except-else-finally block to get more information about the exception. The else clause is executed if no exception is raised, while the finally clause is executed regardless of whether an exception was raised or not. You can use the finally clause to clean up resources or to log the error for debugging purposes.
3. Ignoring the Exception: You can choose to ignore the exception and continue with the script. This is not recommended, as the root cause of the exception may cause more problems later in the script.
4. Terminating the Script: If you do not handle the exception, the Python interpreter will terminate the script and display an error message to the user. This can be useful for debugging purposes, but it is not ideal for production-level scripts.

It's important to choose the appropriate recovery strategy for each exception, as different exceptions may require different responses. The goal should be to handle the exception in a way that allows the script to continue running and to minimize the impact of the exception on the user experience.

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

In Python, there are two main methods for triggering exceptions in your script:

1. Raising Exceptions: You can raise an exception manually using the raise keyword. This is useful when you want to signal an error condition and interrupt the normal flow of the script. For example, you can raise an exception if a function receives an invalid argument:

def divide(a, b):

if b == 0:

raise ZeroDivisionError("division by zero")

return a / b

1. Triggering Built-in Exceptions: Certain actions in Python can trigger built-in exceptions automatically. For example, trying to access an index that doesn't exist in a list will raise an IndexError, while trying to open a file that doesn't exist will raise a FileNotFoundError. These exceptions are raised automatically by the Python interpreter and can be caught and handled using a try-except block.

By using these two methods, you can trigger exceptions in your script and handle them appropriately to ensure that your script runs smoothly and provides the desired results to the user.

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

In Python, there are two main methods for specifying actions to be executed at termination time, regardless of whether or not an exception exists:

1. The finally clause: The finally clause can be used in conjunction with the try-except block to specify a block of code that will be executed no matter what. This is useful for cleaning up resources or for ensuring that critical operations are completed before the script exits. For example:

try:

# Code that may raise an exception

except Exception as e:

# Code to handle the exception

finally:

# Code to be executed no matter what

2. The atexit module: The atexit module provides a simple interface for registering functions to be executed when the script exits, regardless of whether it exits normally or due to an exception. For example:

import atexit

def cleanup():

# Cleanup code to be executed at exit

atexit.register(cleanup)

By using these two methods, you can specify actions to be executed at termination time, regardless of whether or not an exception exists, and ensure that your script cleans up resources and performs other critical operations before it exits.