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

Exception processing is a fundamental feature of most programming languages, including Python. It allows you to handle and recover from unexpected or exceptional situations in your code. Here are three applications for exception processing in Python:

Error Handling: One of the most common applications of exception processing is error handling. When a program encounters an error, such as a division by zero or an invalid file name, it can raise an exception to signal that something has gone wrong. By handling these exceptions, you can provide feedback to the user or take corrective action to prevent the program from crashing.

Resource Management: Another application of exception processing is resource management. When a program needs to use a resource such as a file, network connection, or database, it must acquire and release that resource properly to avoid resource leaks or data corruption. By using exception handling to manage resource acquisition and release, you can ensure that resources are released even if an exception occurs.

Program Flow Control: Exception processing can also be used for program flow control. By raising and catching exceptions at strategic points in your code, you can change the flow of execution based on specific conditions or events. This can be useful for implementing retry logic, error recovery, or complex control flow structures.

Overall, exception processing is a powerful tool that can improve the robustness and reliability of your code. By anticipating and handling exceptions effectively, you can make your programs more resilient to errors and unexpected events.

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

If you don't handle an exception in your code, it will propagate up the call stack until it is either caught by a higher-level exception handler or causes the program to terminate.

In Python, if an unhandled exception occurs, the interpreter prints a traceback that shows where the exception occurred and the sequence of function calls that led up to it. The traceback can be useful for debugging, as it provides a detailed snapshot of the program state at the time of the exception.

However, in most cases, an unhandled exception will cause the program to terminate abnormally, which can be undesirable. This is especially true for long-running or critical applications, where crashes can have serious consequences.

To avoid this, it's important to handle exceptions appropriately in your code, either by catching and handling them or by allowing them to propagate to a higher-level exception handler. This can involve logging the exception, displaying a user-friendly error message, retrying the operation, or taking other appropriate corrective actions.

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

When an exception occurs in a Python program, there are several options for recovering from it. Here are some of the most common approaches:

Catch and handle the exception: You can use a try-except block to catch the exception and handle it in a way that allows your program to continue executing. For example, you might log the error, display an error message to the user, or take corrective action to recover from the error.

Reraise the exception: Sometimes it may be appropriate to let the exception propagate up the call stack so that it can be handled by higher-level code. In such cases, you can use the raise statement without an argument to re-raise the current exception.

Retry the operation: In some cases, an exception might be caused by a temporary error or resource contention. In such cases, you can retry the operation that caused the exception after a brief delay, to see if the error resolves itself. You can use a loop with a try-except block to implement retry logic.

Switch to a backup plan: If an operation fails due to an exception, you might have a backup plan that you can switch to, in order to continue executing your program. For example, you might have a secondary server that you can use if the primary server is unavailable.

Terminate gracefully: If an exception cannot be handled or recovered from, you might need to terminate your program gracefully. This involves releasing any resources that your program has acquired, saving any data that needs to be saved, and displaying a message to the user that explains why the program is terminating.

Overall, the best approach to recovering from an exception depends on the specific context of your program and the nature of the exception. By handling exceptions effectively, you can make your program more robust and reliable in the face of unexpected errors.

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

In Python, there are several ways to trigger an exception manually in your code. Here are two common methods:

Using the raise statement: You can use the raise statement to manually trigger an exception in your code. The raise statement takes an exception object as an argument, and raises that exception in the current context. For example, to raise a ValueError exception, you can write raise ValueError("Invalid value"). You can also raise custom exception types by defining your own exception classes that inherit from the built-in Exception class.

Using assertion statements: Assertion statements are used to test for conditions that should always be true, and raise an AssertionError exception if the condition is false. You can use assertion statements to validate your assumptions about the state of your program, and catch potential errors early in the development cycle. For example, to assert that a variable x is non-negative, you can write assert x >= 0, "x must be non-negative".

Both of these methods allow you to trigger exceptions in your code deliberately, for example, to handle error conditions or validate inputs. It's important to use exceptions judiciously, however, and not to rely on them excessively for control flow in your program.

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

In Python, you can use two methods to specify actions to be executed at termination time, regardless of whether or not an exception exists:

1. The finally clause: You can use the finally clause in a try statement to specify a block of code that should be executed regardless of whether an exception was raised or not. The finally block is always executed, even if a return statement or an uncaught exception is encountered in the try or except block. The syntax for using finally is as follows:

try:

# code that may raise an exception

except ExceptionType:

# exception handling code

finally:

# cleanup code that is always executed

1. The atexit module: You can use the atexit module to register functions that should be executed when the Python interpreter is about to exit. These functions are called in reverse order of registration, after the interpreter's standard cleanup process has completed. This module is useful for performing cleanup tasks such as closing open files, shutting down network connections, or releasing system resources..

import atexit

def cleanup():

# Cleanup code that is always executed

atexit.register (cleanup)