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

Exception processing is a powerful mechanism in programming languages that allows you to handle and manage exceptional or error conditions that may occur during the execution of a program. Here are three common applications for exception processing:

1. Error Handling: Exceptions are commonly used to handle and manage errors that occur during program execution. When an unexpected situation or error occurs, an exception can be raised to indicate the error and provide information about the cause. By catching and handling these exceptions, you can gracefully recover from errors, provide error messages to users, and prevent the program from crashing. Error handling using exceptions helps improve the robustness and reliability of the program.

2. Resource Management: Exception processing is often used for managing resources, such as file handling or network connections. When working with resources, exceptions can be used to handle situations where resources are not available, fail to open or close properly, or encounter other issues. By catching exceptions related to resource management, you can ensure that resources are properly released and avoid resource leaks, ensuring efficient and safe resource usage.

3. Input Validation: Exceptions can be used to validate user input or input from external sources. When validating input, exceptions can be raised if the input does not meet certain criteria or violates constraints. By catching these exceptions, you can provide appropriate feedback to the user, enforce data integrity, and prevent the program from proceeding with invalid input. Exception-based input validation helps ensure that the program operates with valid and expected data, improving the overall reliability and correctness of the system.

In summary, exception processing is commonly used for error handling, resource management, and input validation, among other applications. It helps in managing exceptional conditions, ensuring program reliability, and providing better user experiences by gracefully handling errors and exceptions that may occur during program execution.

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

If an exception is not handled or caught using exception handling mechanisms, it will propagate up the call stack until it reaches the top-level scope of your program. If the exception reaches the top-level scope without being caught, it will typically result in the termination of your program and an error message or traceback will be displayed.

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

When recovering from an exception in your script, you have several options depending on your specific requirements and the nature of the exception. Here are some common approaches:

1. Catch and Handle the Exception: You can use a try-except block to catch specific exceptions and handle them gracefully. By enclosing the code that may raise an exception within a try block and specifying the exception type(s) you want to catch in the except block, you can provide alternative code or error handling routines to recover from the exception.

```python

try:

# Code that may raise an exception

...

except ExceptionType:

# Code to handle the exception and recover

...

```

2. Retry the Operation: In some cases, you might want to retry the operation that raised the exception. You can achieve this by placing the code within a loop and catching the exception. By controlling the loop conditions, you can attempt the operation multiple times until it succeeds or until a maximum number of retries is reached.

```python

max\_retries = 3

retry\_count = 0

while retry\_count < max\_retries:

try:

# Code that may raise an exception

...

break # Operation succeeded, exit the loop

except ExceptionType:

# Code to handle the exception

retry\_count += 1

```

3. Provide Default Values or Alternatives: If an exception occurs, you can provide default values or alternative actions to ensure the program continues running without interruption. For example, you can assign default values to variables, use fallback options, or provide alternative computations to replace the failed operation.

```python

try:

# Code that may raise an exception

...

except ExceptionType:

# Code to handle the exception

# Provide default values or alternative actions

...

```

4. Log and Notify: You can log the exception details to a log file or send notifications to the appropriate parties. Logging exceptions helps in debugging and analyzing issues that occur during execution. Additionally, notifications can be sent to system administrators, developers, or users to inform them about the exceptional situation.

```python

import logging

try:

# Code that may raise an exception

...

except ExceptionType as e:

# Code to handle the exception

logging.exception("An exception occurred: %s", str(e))

# Send notifications or perform other actions

...

```

These are just a few options for recovering from exceptions in your script. The approach you choose will depend on the specific requirements of your application, the nature of the exception, and the desired behavior in exceptional situations.

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

In Python, there are several ways to trigger or raise exceptions in your script. Here are two common methods for triggering exceptions:

1. Using the `raise` statement: You can use the `raise` statement to explicitly raise an exception at any point in your code. This allows you to create and raise custom exceptions or raise built-in exceptions provided by Python. The `raise` statement is followed by the exception type or an instance of an exception class.

```python

# Raise a built-in exception

raise ValueError("Invalid value entered.")

# Raise a custom exception

class MyException(Exception):

pass

raise MyException("An error occurred.")

```

In the above examples, the `raise` statement is used to raise a `ValueError` exception with a custom error message and to raise a custom `MyException` exception.

2. Invoking a built-in function or method that raises exceptions: Many built-in functions and methods in Python can raise exceptions under certain conditions. By calling these functions or methods and providing input that violates their requirements, you can trigger exceptions. Some commonly used functions and methods that raise exceptions include `open()` for file I/O, `int()` for converting strings to integers, and `index()` for finding the index of an element in a list.

```python

# Triggering an exception with open() function

try:

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

except FileNotFoundError:

print("File not found!")

# Triggering an exception with int() function

try:

num = int("abc")

except ValueError:

print("Invalid integer format!")

# Triggering an exception with index() method

try:

my\_list = [1, 2, 3]

index = my\_list.index(4)

except ValueError:

print("Element not found in the list!")

```

In the above examples, the `open()` function raises a `FileNotFoundError` when attempting to open a file that does not exist, the `int()` function raises a `ValueError` when trying to convert a non-numeric string to an integer, and the `index()` method raises a `ValueError` when the element is not found in the list.

These are just two methods for triggering exceptions in your script. Depending on the situation, you can raise exceptions explicitly using the `raise` statement or encounter them when invoking functions or methods that have exception-raising behavior.

**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 specify actions to be executed at termination time, regardless of whether or not an exception exists, using two methods:

1. Using a `finally` block: The `finally` block is used in conjunction with a `try-except` block to define code that will always be executed, regardless of whether an exception is raised or not. The code within the `finally` block is executed after the `try` block completes, regardless of whether an exception occurred or was caught. This allows you to ensure that certain cleanup or finalization actions take place.

```python

try:

# Code that may raise an exception

...

except ExceptionType:

# Code to handle the exception

...

finally:

# Code to be executed at termination time

# regardless of exception existence

...

```

In the above example, the code within the `finally` block will always be executed, ensuring that the specified actions take place even if an exception occurs or is caught and handled.

2. Using the `atexit` module: The `atexit` module provides a way to register functions that will be called at the termination of the program, regardless of whether an exception occurred or not. These registered functions are executed in the reverse order of their registration. You can use the `atexit.register()` function to register a function to be executed at program termination.

```python

import atexit

def cleanup\_function():

# Code to be executed at termination time

# regardless of exception existence

...

# Register the cleanup function

atexit.register(cleanup\_function)

```

In this example, the `cleanup\_function()` will be automatically called when the program exits, regardless of whether or not an exception occurred. You can register multiple functions using multiple `atexit.register()` calls, and they will be executed in the reverse order of registration.

These methods provide a way to specify actions that should be executed at termination time, ensuring cleanup or finalization tasks are performed, regardless of whether an exception occurred or not. The `finally` block is useful for local exception handling scenarios, while the `atexit` module is suitable for global termination tasks.