Q1. Describe three applications for exception processing.

Answer :

Three Applications for Exception Processing in Python:

1.Handling Invalid User Input:

Exception processing is commonly used to handle incorrect or unexpected user inputs. For instance, if a program expects a number but receives a string, it can raise a ValueError. Instead of crashing, the program can catch this exception and prompt the user to enter a valid input.

Code Below :

```
try:
   user_input = int(input("Enter a number: "))
except ValueError:
   print("Invalid input! Please enter a valid number.")
```

2.Managing File Operations:

When working with file I/O (Input/Output), files may not exist, or the program may not have permission to read/write them. Exception handling ensures that the program deals with these scenarios gracefully, avoiding a crash.

Code Below :

```
try:
   with open("data.txt", "r") as file:
      data = file.read()
except FileNotFoundError:
   print("File not found! Please check the file path.")
except PermissionError:
   print("Permission denied! You don't have access to this file.")
```

3.Network and Database Operations:

Network or database operations can fail due to issues like connection timeouts, server unavailability, or incorrect credentials. Exception handling allows the program to retry the operation, notify the user, or take other actions in case of failure.

Code Below :

```
import requests

try:
   response = requests.get("https://example.com")
```

```
    response.raise_for_status()  # Check for HTTP errors
except requests.exceptions.HTTPError as http_err:
    print(f"HTTP error occurred: {http_err}")
except requests.exceptions.ConnectionError:
    print("Connection error! Unable to reach the server.")
except Exception as err:
    print(f"An unexpected error occurred: {err}")
```

Summary:

Exception processing ensures that programs handle errors robustly in various scenarios, such as invalid user input, file handling errors, and network or database failures. It allows developers to create user-friendly applications that avoid crashing and provide meaningful feedback when something goes wrong.

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

Answer :

If we don't handle or treat an exception in Python, the following happens:

1.Program Crashes:

The program will terminate abruptly at the point where the exception occurs. Any code after the exception will not be executed, leading to a crash.

2.Traceback is Displayed:

Python will display a **traceback** (a detailed error message) that shows the type of the exception, the line of code where the error occurred, and the call stack leading to the error. This traceback helps developers diagnose what went wrong, but it's not user-friendly.

Example of a traceback:

Code Below :

```
Traceback (most recent call last):
  File "example.py", line 1, in <module>
    x = 5 / 0
ZeroDivisionError: division by zero
```

3.Poor User Experience:

If exceptions are not handled properly, the user may see an unexpected error message or experience a frozen application, which can lead to frustration or confusion.

Example without exception handling:

Code Below :

```
x = 5 / 0  # This will cause a ZeroDivisionError and crash the program
print("This line will never be executed.")
```

Output:

```
ZeroDivisionError: division by zero
```

In Summary:

If an exception is not handled, the program crashes, displays an error traceback, and any subsequent code will not be executed. This can result in a poor user experience and an ungraceful exit from the application. Therefore, it's essential to treat exceptions to maintain the stability of the program.

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

Answer :

When an exception occurs in your script, we have several options for recovering from it and continuing the execution gracefully. Here are the most common strategies:

1.Use a try-except Block:

The basic method for handling exceptions is using a try-except block. You place the code that might raise an exception in the try block, and if an exception occurs, the code in the except block will execute instead of crashing the program.

Code Below :

```
try:
   x = 5 / 0  # This will raise a ZeroDivisionError
except ZeroDivisionError:
   print("Division by zero is not allowed, setting x to 0.")
   x = 0
```

Output:

```
Division by zero is not allowed, setting x to 0.
```

In this case, the program handles the exception, sets x to 0, and continues.

2. Use else for Code that Runs If No Exception Occurs:

We can add an else clause that runs only if no exception occurs. This allows you to separate the normal logic from the exception-handling logic.

Code Below :

```
try:
   x = 5 / 1
except ZeroDivisionError:
   print("Cannot divide by zero.")
else:
   print("Division succeeded, x =", x)
```

Output:

```
Division succeeded, x = 5.0
```

The code in the else block runs only if the try block does not raise an exception.

3. Use finally for Cleanup Actions:

A finally block is executed whether an exception occurs or not. This is often used for cleanup operations, such as closing files or releasing resources.

Code Below :

```
try:
   file = open("nonexistent.txt", "r")
except FileNotFoundError:
   print("File not found.")
finally:
   print("Cleaning up resources.")
```

Output:

```
File not found.
Cleaning up resources.
```

The finally block ensures that cleanup occurs even if an exception happens.

4. Raise an Exception Again (raise)

If we can't handle the exception in the current context, you can catch it and then re-raise it using raise. This allows the program to deal with it at a higher level.

Code Below :

```
try:
   x = int("abc")
except ValueError as e:
   print(f"Error occurred: {e}")
   raise  # Re-raises the exception after logging it
```

This strategy is useful when you need to log the exception or perform specific actions before re-raising it.

5. Use a Fallback or Default Action

When an exception occurs, you can provide a fallback or default behavior to recover and continue the script.

Code Below :

```
try:
   user_input = int(input("Enter a number: "))
except ValueError:
   print("Invalid input, using default value of 10.")
   user_input = 10
```

In this case, if the user enters something that can't be converted to an integer, the script will use a default value of 10 instead of crashing.

6. Log the Exception for Later Analysis

Sometimes you want to log the exception for later debugging without affecting the user experience. This can be done by using the logging module.

Code Below :

```
import logging
logging.basicConfig(filename='error.log', level=logging.ERROR)

try:
   result = 5 / 0
except ZeroDivisionError as e:
   logging.error(f"Error occurred: {e}")
   print("An error occurred, but it's logged for analysis.")
```

Output :

```
An error occurred, but it's logged for analysis.
```

This doesn't fix the issue, but it allows developers to track errors.

Summary:

To recover from an exception in your script, you can:

1. Use a try-except block to handle the error and continue.
2. Use an else clause to execute code when no exception occurs.
3. Use a finally block to ensure certain code runs regardless of exceptions.
4. Re-raise the exception with raise if it needs to be handled elsewhere.
5. Provide fallback actions, such as default values or alternative logic.
6. Log the exception for future debugging.

These techniques help maintain the stability and robustness of your program.

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

Answer :

There are two primary methods for triggering exceptions in Python:

1.Using the raise Statement

- The raise statement allows you to trigger an exception manually in your script. You can either raise a built-in exception or define your custom exceptions.
- The basic syntax is:

Code Below :

raise <ExceptionType>(<OptionalErrorMessage>)

This is typically used to enforce certain conditions or to indicate that something has gone wrong in the program logic.

Example 1: Raising a built-in exception:

Code Below:

```
def divide(a, b):
    if b == 0:
        raise ZeroDivisionError("Cannot divide by zero!")
    return a / b

try:
    divide(10, 0)
except ZeroDivisionError as e:
    print(e)
```

Output:

Cannot divide by zero!

Example 2: Raising a custom exception

Code Below:

```
class CustomError(Exception):
   pass

def check_value(x):
   if x < 0:
      raise CustomError("Negative value is not allowed!")

try:
   check_value(-1)
except CustomError as e:
   print(e)
```

Output:

Code Below :

```
Negative value is not allowed!
```

The raise statement gives you control over when and why exceptions are triggered, allowing you to enforce rules in your program.

2. Using Assertions (assert statement)

- The assert statement is another way to trigger exceptions based on conditions. It is used to test whether a condition in your program is True. If the condition evaluates to False, it raises an AssertionError.
- Assertions are useful for catching bugs in code during development by ensuring that certain conditions hold true.

Syntax:

Code Below :

```
assert <condition>, <OptionalErrorMessage>
```

Code Below :

```
def process_number(x):
   assert x > 0, "Input must be a positive number!"
   return x * 2

try:
   process_number(-5)
except AssertionError as e:
   print(e)
```

Output:

Code Below :

Input must be a positive number!

Assertions are often used to validate data or check assumptions made by the programmer. However, they are generally not used for handling runtime errors in production code, as they can be disabled with the -O (optimize) flag when running Python.

Summary:

1. **raise statement**: Explicitly triggers exceptions to indicate errors or invalid conditions.
2. **assert statement**: Checks a condition, raising an AssertionError if the condition is False. It is mainly used for debugging purposes during development.

Both methods give you control over how and when exceptions should occur in your script.

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

Answer :

Two primary methods in Python can be used to specify actions to be executed at termination time, regardless of whether an exception occurs or not:

1.finally Block:

- The finally block is part of the try-except-finally structure and ensures that the code inside it is always executed, regardless of whether an exception was raised or not.
- The finally block is typically used to release resources or perform cleanup tasks, such as closing files or network connections.

Syntax:

Code Below :

```
try:
   # Code that might raise an exception
except <ExceptionType>:
   # Handle the exception
finally:
   # Code to execute regardless of exception
```

Code Below :

```
try:
   file = open('example.txt', 'r')
   # Perform file operations
   content = file.read()
   print(content)
except FileNotFoundError:
   print("File not found!")
finally:
   file.close()  # This will always run
   print("File closed.")
```

Output (when the file exists):

Code Below :

```
<contents of the file>
File closed.
```

Output (when the file doesn't exist):

Code Below :

File not found!
File closed.

Explanation: Even if an exception is raised in the try block (like FileNotFoundError), the finally block will still execute, ensuring the file is closed or any necessary cleanup is performed.

2.with Statement (Context Managers):

- The with statement is used to ensure that resources are properly managed, even if exceptions occur. The with statement is often used with objects that support the context management protocol (i.e., objects that define __enter__ and __exit__ methods).
- It simplifies the process of cleanup by automatically handling the setup and teardown of resources, such as file handling, network connections, etc

Syntax:

Code Below :

```
with <context_manager> as <variable>:
   # Code to execute with the resource
```

Example:

Code Below :

```
try:
   with open('example.txt', 'r') as file:
      content = file.read()
      print(content)
except FileNotFoundError:
   print("File not found!")
```

Explanation: The with statement automatically takes care of closing the file, even if an exception occurs during the file reading. This eliminates the need to manually close the file, as would be necessary with the finally block.

Key Points:

1. **finally Block**: Ensures that the specified actions are performed, such as cleanup, regardless of whether an exception occurs or not.

2. **with Statement (Context Manager)**: Ensures the proper handling of resources by automatically performing setup and teardown actions, such as opening and closing files or connections.

Both methods are commonly used for resource management and cleanup, ensuring smooth termination of code execution.