Q1. What is the purpose of the try statement?

Answer :

The purpose of the **try** statement in Python is to define a block of code in which exceptions might occur and to provide a mechanism for handling those exceptions if they are raised. It allows you to manage and respond to runtime errors in a controlled way without causing your entire program to crash.

Key Parts of the try Statement:

1. **try** Block:

- ❖ This block contains the code that might raise an exception.
- ❖ If an exception occurs in this block, the program immediately moves to the corresponding **except** block to handle it.

2. **except** Block:

- ❖ This block contains code that specifies how to handle specific exceptions.
- ❖ If an exception occurs in the try block, the code in the except block is executed.
- ❖ We can define multiple except blocks to handle different types of exceptions.

3. **else** Block *(optional)*:
    - ❖ If no exceptions are raised in the try block, the else block is executed.
    - ❖ It allows for executing code that should only run if no exceptions occurred.
4. **finally** Block *(optional)*:
    - ❖ The **finally** block is executed no matter what—whether an exception was raised or not.
    - ❖ It is typically used to release resources or perform cleanup tasks, like closing files or database connections.

Example:

Code Below:

```
try:
   num = int(input("Enter a number: "))
```

```
    result = 10 / num
except ValueError:
    print("That's not a valid number!")
except ZeroDivisionError:
    print("You can't divide by zero!")
else:
    print(f"Result is {result}")
finally:
    print("Execution finished.")
```

Output Scenarios:

1.User enters "5":

Code Below :

```
Result is 2.0
Execution finished.
```

2.User enters "abc" (raises ValueError):

Code Below :

```
That's not a valid number!
Execution finished.
```

3.User enters "0" (raises ZeroDivisionError):

Code Below :

```
You can't divide by zero!
Execution finished.
```

Purpose:

- **Error Handling**: The main purpose of the try statement is to handle potential errors (exceptions) that might occur during the execution of a block of code.
- **Program Stability**: It prevents the program from crashing due to unhandled exceptions.
- **Graceful Degradation**: Allows the program to degrade gracefully by catching and handling errors and continuing execution (or shutting down gracefully).

Q2. What are the two most popular try statement variations?

Answer :

The two most popular variations of the try statement in Python are:

1.try-except Block:

This is the most basic form of the try statement. It is used to handle exceptions that occur in the try block by catching them with an except block.

Syntax:

Code Below :

```
try:
   # Code that may raise an exception
except ExceptionType:
   # Code that runs if the specific exception occurs
```

Example:

Code Below :

```
try:
   result = 10 / 0
except ZeroDivisionError:
   print("You can't divide by zero!")
```

Explanation:

In this example, if a ZeroDivisionError occurs (dividing by zero), the except block will catch it and print the message. If no exception occurs, the except block is skipped.

2. try-except-finally Block:

This variation includes a finally block, which is executed regardless of whether an exception occurs or not. It is often used for cleanup actions like closing files or releasing resources.

Syntax:

Code Below :

```
try:
   # Code that may raise an exception
except ExceptionType:
   # Code that runs if the specific exception occurs
finally:
   # Code that runs no matter what (e.g., cleanup code)
```

Example:

```
try:
   file = open("data.txt", "r")
   content = file.read()
except FileNotFoundError:
   print("The file was not found.")
finally:
   print("Closing file.")
   file.close()
```

Explanation:

The finally block will always run, ensuring the file is closed regardless of whether an exception occurs.

Additional Variations:

- **try-except-else Block**: This is used when you want to execute a specific block of code only if no exception occurs.
- **try-except-else-finally Block**: Combines all parts, allowing for error handling (except), execution if no error occurs (else), and guaranteed execution (finally).

Example:

Code Below :

```
try:
   result = 10 / 2
except ZeroDivisionError:
   print("You can't divide by zero!")
else:
   print(f"Result is {result}")
finally:
   print("End of calculation.")
```

Explanation:

The else block runs if no exceptions are raised, and the finally block always runs.
These variations provide flexible ways to handle exceptions, ensuring both error catching and proper resource management.

Q3. What is the purpose of the raise statement?

Answer :

The raise statement in Python is used to explicitly trigger an exception in your code. It allows the programmer to forcefully raise a specific exception when certain conditions are met, even if no exception has naturally occurred.

Purpose of the raise Statement:

1. **Manually Trigger Exceptions**: We can use the raise statement to create an exception where one does not naturally occur, usually to signal that something went wrong in your program.

2. **Propagating Exceptions**: It can be used to re-raise an existing exception after catching it, allowing higher-level functions to handle the error.

3. **Custom Error Messages**: We can customize the error message when raising exceptions, providing more context or clarity to the user or developer.

Syntax:

Code Below :

```
raise ExceptionType("Custom error message")
```

Example 1: Manually Raising an Exception

Code Below :

```
def check_age(age):
    if age < 18:
        raise ValueError("Age must be 18 or older.")
    else:
        print("Access granted.")

check_age(16)
```

Output:

```
ValueError: Age must be 18 or older.
```

Explanation:

In this example, if the age is less than 18, the raise statement triggers a ValueError with a custom message.

Example 2: Re-raising an Exception

Code Below :

```
try:
   x = 1 / 0
except ZeroDivisionError:
   print("Caught ZeroDivisionError.")
   raise  # Re-raises the original exception
```

Output:

Code Below :

```
Caught ZeroDivisionError.
ZeroDivisionError: division by zero
```

Explanation:

In this example, the exception is first caught in the except block, but then it is re-raised using raise, allowing it to be propagated to a higher level for additional handling.

Custom Exceptions:

We can also define custom exceptions using classes and raise them in specific situations.

Code Below :

```
class CustomError(Exception):
   pass

raise CustomError("This is a custom error.")
```

Key Points:

- The raise statement provides control over exception handling by allowing you to raise or propagate errors based on your program's needs.
- It's useful for enforcing conditions, handling unexpected situations, or passing errors up the call chain for more general handling.

Q4. What does the assert statement do, and what other statement is it like?

Answer :

The assert statement in Python is used for debugging purposes. It tests a condition (expression) and triggers an exception if the condition is not met (i.e., if the condition evaluates to False). When an assert statement fails, Python raises an AssertionError with an optional custom message. It is similar to an **if statement combined with a raise statement**, as it checks a condition and raises an error if the condition is false.

Purpose of the assert Statement:

1. **Debugging**: It helps identify bugs during development by checking conditions that should always be true in a program.
2. **Sanity Check**: It acts as a safety mechanism to ensure that a particular assumption holds before the code proceeds.

Syntax:

Code Below :

```
assert condition, optional_message
```

- **condition:** An expression that is expected to evaluate to True.
- **optional_message:** (Optional) A message that gets displayed if the assertion fails.

Example 1: Basic Use of assert

Code Below :

```
x = 5
assert x > 0, "x must be greater than 0"
```

Output:

Code Below :

```
AssertionError: x must be greater than 0
```

Explanation:

In this example, since x > 0 is False, the assert statement raises an AssertionError, displaying the message "x must be greater than 0".

How assert is Similar to Other Statements:

Similar to an if statement with a raise: The assert statement checks a condition like an if statement does, and if the condition is false, it raises an AssertionError, which is similar to how you would manually raise an exception.

- For example, the following code:

Code Below :

```
if not condition:
   raise AssertionError(optional_message)
```

behaves similarly to:

Code Below :

```
assert condition, optional_message
```

When to Use assert:

Assertions are typically used during development and testing to catch logic errors. They are not meant to handle regular errors that might occur in production.

Important Note:

Assertions can be disabled: When running Python in optimized mode (using the -O flag), all assertions are skipped and not executed. For example:

Code Below :

```
python -O script.py
```

This makes assertions suitable only for testing and debugging, not for handling production errors.

Summary:

- The assert statement is used for debugging by testing assumptions in the code.
- It's similar to an if statement combined with a raise statement, as it checks conditions and raises an AssertionError when the condition is false.
- Assertions are useful during development but should not be relied upon for production error handling.

Q5. What is the purpose of the with/as argument, and what other statement is it like?

Answer :

The with statement in Python is used to simplify resource management, such as file handling, where resources need to be properly opened and closed. The with statement ensures that resources are properly cleaned up after use, even if errors occur during execution. It is often used with the as keyword to assign the resource to a variable.

Purpose of with/as:

1. **Simplifies Resource Management**: It manages the setup and cleanup of resources (e.g., files, network connections, or locks) without having to explicitly close or release them.

2. **Handles Exceptions Gracefully**: If an exception occurs during the execution of the block, the resource is still properly cleaned up, avoiding potential resource leaks.

3. **Avoids Repetitive Code**: We don't need to explicitly write the resource cleanup code (like close()), as it's automatically handled.

Syntax:

Code Below:

```
with expression [as variable]:
   # Code block
```

expression: An object (often one that supports the context management protocol, having __enter__() and __exit__() methods).

variable: (Optional) A variable to which the result of the expression is assigned, typically used within the block.

Example 1: Using with to Open a File

```
with open('example.txt', 'r') as file:
   data = file.read()
   print(data)
```

Explanation: Here, the with statement ensures that the file example.txt is opened, read, and then automatically closed once the block of code is done, even if an error occurs while reading the file.

Equivalent Without with:

Without the with statement, you'd need to explicitly open and close the file like this:

Code Below:

```
file = open('example.txt', 'r')
try:
   data = file.read()
   print(data)
finally:
   file.close()
```

Explanation: The finally block ensures that the file is closed even if an error occurs. The with statement simplifies this code.

Example 2: Using with/as with Locks:

Code Below :

```
from threading import Lock

lock = Lock()
with lock:
   # Critical section
   print("Lock acquired, performing operations")
```

Explanation: The with statement acquires the lock at the beginning of the block and releases it at the end, even if an exception occurs during the execution of the critical section.

Similarity to Other Statements:

Similar to try/finally: The with statement is functionally similar to using try/finally for resource management. The try block would contain the resource usage, and the finally block would handle cleanup. In the with statement, the resource cleanup is managed automatically, making it shorter and less error-prone.

How the with Statement Works Internally:

The object passed to the with statement must implement two special methods:

- __enter__(): This is called when the execution enters the with block. It typically returns the resource (e.g., a file object).
- __exit__(): This is called when the execution exits the with block. It handles any necessary cleanup (e.g., closing the file), even if an exception is raised.

Example 3: Custom Context Manager:

We can define our own context manager by implementing the __enter__() and __exit__() methods:

Code Below :

```
class MyContext:
   def __enter__(self):
      print("Entering the context")
      return self

   def __exit__(self, exc_type, exc_value, traceback):
      print("Exiting the context")

with MyContext() as context:
   print("Inside the with block")
```

Output:

```
Entering the context
Inside the with block
Exiting the context
```

Summary

- The with statement is used for resource management and ensures that resources are properly handled.
- It automatically handles the setup and cleanup of resources, reducing the need for explicit try/finally blocks.
- The as keyword allows the resource to be assigned to a variable for use within the block.
- This is most commonly used for file operations, thread locks, and network connections, but can also be applied to custom context managers.