1. What is the role of try and exception block?

Ans. The try and except block is a fundamental concept in error handling within programming languages. It allows developers to anticipate and handle potential errors or exceptions that may occur during the execution of their code. The try block encapsulates the code that could potentially generate an exception. If an exception occurs within the try block the flow of control immediately transfers to the except block. The except block contains the code to handle the exception providing a way for the program to gracefully recover from unexpected errors.

By using try and except blocks developers can ensure that their program continues to run smoothly even in the presence of errors. It allows for the handling of specific exceptions resulting in more granular control over error handling and providing the ability to handle different types of errors differently.

1. What is the syntax for a basic try-except block?

Ans.

try:

# Code block to be executed

# that may raise an exception

except ExceptionType:

# Code block to handle the exception

1. What happens if an exception occurs inside a try block and there is no matching except block?

Ans. If an exception occurs inside a try block and there is no matching except block to handle it the exception will propagate up the call stack until it is caught by an appropriate except block or until it reaches the top-level of the program. If the exception is not caught it will result in the program terminating and an error message will be displayed. The default behaviour of most programming languages is to print a stack trace which shows the sequence of function calls that led to the point where the exception occurred. In some cases, the program may be able to gracefully handle the exception by using a finally block if one is present. The code inside the finally block will always execute regardless of whether an exception occurred or not. This allows the program to perform any necessary cleanup or resource release before terminating. However, the finally block does not handle or catch the exception itself it is just executed before the program ends.

1. What is the difference between using a bare except block and specifying a specific exception type?

Ans. When writing code that includes error or exception handling there is a crucial distinction between using a bare `except` block and specifying a specific exception type.

1. Bare `except` block:

- A bare `except` block catches any type of exception that is raised within the `try` block.

- It doesn't differentiate between different exception types.

- This can be problematic because it makes it difficult to identify the specific cause of the error.

- It may result in unexpected or unintended error handling.

- It can make debugging and troubleshooting more challenging.

Example of a bare `except` block:

try:

# Some code that may raise exceptions

except:

# Exception handling code

2. Specifying a specific exception type:

- By specifying a specific exception type you can catch and handle only that particular type of exception.

- This allows for more accurate and targeted error handling.

- It provides clearer information about the cause of the error.

- It enables you to handle different exceptions in different ways tailoring the approach based on the type of error.

Example of specifying a specific exception type:

try:

# Some code that may raise exceptions

except ValueError:

# Handling code for ValueError

except FileNotFoundError:

# Handling code for FileNotFoundError

except:

# Catch-all for other exceptions

1. Can you have nested try-except blocks in Python? If yes, then give an example.

Ans. Yes, nested try-except blocks are allowed in Python. Nested try-except blocks can be used to handle different exceptions at different levels of code execution.

Example of nested try-except blocks in Python:

try:

# Outer try block

numerator = int(input("Enter the numerator: "))

denominator = int(input("Enter the denominator: "))

Input: -

try:

# Inner try block

result = numerator / denominator

print("Result” result)

except ZeroDivisionError:

print("Error: Division by zero is not allowed.")

except ValueError:

print("Error: Please enter valid integer values.")

1. Can we use multiple exception blocks, if yes then give an example.

Ans. Yes, it is possible to use multiple exception blocks in a try-except statement to handle different types of exceptions separately. Each except block can catch and handle a specific type of exception.

Input: -

try:

# Some code that could raise exceptions

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

result = 10 / num

print("Result result)

except ValueError:

# Handles exception raised when the input is not a valid integer

print("Invalid input. Please enter a valid integer.")

except ZeroDivisionError:

# Handles exception raised when a zero is entered as the denominator

print("Cannot divide by zero.")

except Exception as e:

# Handles any other unexpected exception

print("An error occurred str(e))

1. Write the reason due to which following errors are raised:

a. EOFError

b. FloatingPointError

c. IndexError

d. MemoryError

e. OverflowError

f. TabError

g. ValueError

Ans.

a. EOFError: This error is raised when the end of a file or input stream is reached unexpectedly while trying to read data. It occurs when an input function fails to receive the expected input before the end of the file.

b. Floating PointError: This error is raised when a floating-point operation fails to produce a valid result such as division by zero or arithmetic involving infinite or NaN (Not a Number) values.

c. IndexError: This error is raised when trying to access an index that is outside the range of valid indices for a sequence or list. It occurs when an index specified is negative or exceeds the length of the sequence.

d. MemoryError: This error is raised when an operation fails due to insufficient memory allocation. It occurs when the system runs out of memory while trying to allocate space for an object such as when allocating a large array or data structure.

e. OverflowError: This error is raised when the result of an arithmetic operation exceeds the maximum representable value for a numeric type. It occurs when performing calculations that result in a value exceeding the limits of the numeric type being used.

f. TabError: This error is raised when the indentation in Python code is not correctly aligned. It occurs when mixing tabs and spaces or using inconsistent indentation within a block of code.

g. ValueError: This error is raised when a function receives an argument of an unexpected type or value. It occurs when a built-in function or method is called with an argument that is of an inappropriate or unsupported type. It can also occur when trying to convert a value to a different type that is not possible or valid.

1. Write code for the following given scenario and add try-exception block to it.

a. Program to divide two numbers

b. Program to convert a string to an integer

c. Program to access an element in a list

d. Program to handle a specific exception

e. Program to handle any exception

Ans.

a. Program to divide two numbers:

Input: -

def divide\_numbers(num1 num2):

try:

result = num1 / num2

print("The result of division is result)

except ZeroDivisionError:

print("Error: Cannot divide by zero!")

divide\_numbers(10, 2) # Output: The result of division is: 5.0

divide\_numbers(5 ,0) # Output: Error: Cannot divide by zero!

b. Program to convert a string to an integer:

Input: -

def convert\_to\_integer(string):

try:

num = int(string)

print("The converted integer num)

except ValueError:

print("Error: Invalid input. Cannot convert to an integer!")

convert\_to\_integer("10") # Output: The converted integer: 10

convert\_to\_integer("abcd") # Output: Error: Invalid input. Cannot convert to an integer!

c. Program to access an element in a list:

Input: -

def access\_list\_element(lst index):

try:

element = lst[index]

print("The element at index index "is element)

except IndexError:

print("Error: Index out of range!")

my\_list = [1 2 3 4 5]

access\_list\_element(my\_list 2) # Output: The element at index 2 is: 3

access\_list\_element(my\_list 10) # Output: Error: Index out of range!

d. Program to handle a specific exception:

Input: -

def handle\_specific\_exception(num1 num2):

try:

result = num1 / num2

print("The result of division is result)

except ZeroDivisionError:

print("Error: Cannot divide by zero!")

handle\_specific\_exception(10 2) # Output: The result of division is: 5.0

handle\_specific\_exception(5 0) # Output: Error: Cannot divide by zero!

e. Program to handle any exception:

Input: -

def handle\_any\_exception(num1 num2):

try:

result = num1 / num2

print("The result of division is result)

except Exception as e:

print("An error occurred str(e))

handle\_any\_exception(10 2) # Output: The result of division is: 5.0

handle\_any\_exception(5 0) # Output: An error occurred: division by zero