Exceptional Handling in python

Exception handling allows developers to manage errors and unexpected situations.

Exception handling is the process of managing errors and unexpected events that occur during the execution of a program.

When a Python program encounters an error, it raises an exception, which interrupts the normal flow of the program. Exception handling allows developers to handle these exceptions, ensuring that the program can recover from errors and continue executing with the normal flow.

Advantages of using Exceptional Handling

1. Robustness of code
2. Debugging
3. Clean code
4. Prevention of data loss

Disadvantages of using Exceptional handling

1. Increases the code complexity
2. False sense of security
3. Performance overhead

You can handle exceptions in Python using try, except, and finally keywords.

**try-except Block**: to catch the exception

* Try and except statements are used to catch and handle exceptions in Python.
* If an exception occurs within the try block, Python looks for an except block that can handle that specific type of exception.

Syntax :

try:

 Statements to be executed

except:

 Statements get executed if an exception occurs

Example :

=========================================

Input :

try:

 result = 18 / 0 # Division by zero that will cause exception

except ZeroDivisionError:

 print("Error: Division by zero!")

Output:

ERROR!

Error: Division by zero!

===============================================

**Handling Multiple Exceptions**

* Python supports handling multiple exceptions within a single try block by specifying multiple except blocks or using a tuple of exception types.
* A try statement can have more than one except clause, to specify handlers for different exceptions and at most one handler will be executed.

Syntax:

try :

 Statements that are to be executed

except exception\_1:

 Statements

# if exception\_1 occurs, this block will be executed

except exception\_2:

Statements

# if exception\_2 occurs, this block will be executed

Example:

=====================================================

Input :

my\_list = [76, 92, 53]

try:

 print(my\_list[5]) # Accessing an out-of-range index

 print(my\_list["value"]) # Accessing a key in a list

except IndexError:

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

except KeyError:

 print("Error: Key does not exist!")

Output :

ERROR!

Error: Index out of range!

================================================

**try with else clause**

* It provides a way to differentiate between the code that might raise an exception and the code that should execute only if no exceptions are raised in the code.
* The else clause is used to execute code only if no exceptions occur in the try block.

Syntax:

try:

 # Code that may raise an exception

except Exceptiontype1:

 # Handle Exceptiontype1

except Exceptiontype2:

 # Handle Exceptiontype2

# More except blocks if needed

else:

 # Code to execute if no exceptions occur in the try block

Example :

=============================================================

Input:

try:

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

 res = 10 / m

except ZeroDivisionError:

 print("Error: Division by zero!")

else:

 print("Division successful", res)

Output:

Enter a number: 50

Division successful 0.2

=======================================================

**Finally Keyword**

* The finally block is used to define cleanup actions that must be executed whether an exception occurs or not.
* It guarantees that specific code will be executed under all circumstances, irrespective of whether an exception is triggered and regardless of whether it's handled or not.
* The finally block is commonly used for releasing resources, closing files, or performing cleanup tasks that need to occur regardless of the program's state.
* If an exception is raised and not caught, the finally block still executes before the exception propagates further.

Syntax:

try:

 # Code that may raise an exception

except ExceptionType:

 # Handle the exception

finally:

 # Code that always executes

Example:

====================================================

Input:

try:

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

 result = 10 / number

 print("Result:", result)

except ZeroDivisionError:

 print("Error: Division by zero!") # if exception this will be printed

finally:

 print("Execution completed.") # This will be printed always

Output:

Enter a number: 99

Result: 0.10101010101010101

Execution completed.

================================================================

**raise - raising exception**

* The raise statement is used in error handling to intentionally raise exceptions during program execution.
* It serves several purposes, including signaling errors, enforcing preconditions, and controlling program flow.
* It allows developers to handle exceptional conditions or unexpected situations by signaling that an error has occurred in the program.

Syntax:

raise ExceptionClassName("Optional error message")

Example:

 Input

def calculate\_square\_root(number):

 if number < 0:

 raise ValueError("Cant calculate square root of a negative number")

 else:

 return number \*\* 0.5

try:

 result = calculate\_square\_root(-9) # This will raise a ValueError when given negative number

except ValueError as e:

 print("Error:", e)

Output:

ERROR!

Error: Cant calculate square root of a negative number