Training Report Day-16

24 June 2024

Introduction to Exception Handling

Exception handling in Python is a mechanism to respond to runtime errors, preventing the program from crashing and allowing the program to handle errors gracefully. It helps in debugging, maintaining clean code, and providing user-friendly error messages.

Key Concepts

1. Exception: An exception is an error that occurs during the execution of a program. When an exception is raised, the normal flow of the program is interrupted.

2. Try Block: The code that might raise an exception is placed inside a try block.

3. Except Block: The code that handles the exception is placed inside an except block.

4. Else Block: The code inside the else block is executed if no exceptions are raised.

5. Finally Block: The code inside the finally block is executed regardless of whether an exception is raised or not.

6. Raise: Used to raise an exception manually.

Example:-

```
# code that may raise an exception

# code that runs if the exception occurs

# code that runs if no exception occurs

# code that runs no matter what
```

Common Built-in Exceptions

1. Index Error
2. Key Error
3. Value Error
4. Type Error
5. ZeroDivisionError

6. FileNotFoundError
7. Error
8. Import Error
9. Attribute Error
10. Runtime Error

Example 1: Handling Division by Zero

```
def divide(a, b):
    try:
        result = a / b
    except ZeroDivisionError:
        return "Cannot divide by zero!"
    else:
        return result
    finally:
        print("Execution of divide function complete.")

print(divide(10, 2))  # Output: 5.0
print(divide(10, 0))  # Output: Cannot divide by zero!
```

Example 2: Handling File Operations

```
def read_file(file_path):
    try:
        with open(file_path, 'r') as file:
            data = file.read()
    except FileNotFoundError:
        return "File not found!"
    except IOError:
        return "Error reading file!"
    else:
        return data
    finally:
        print("Execution of read_file function complete.")

print(read_file("existing_file.txt"))  # Output: (contents of the file)
print(read_file("nonexistent_file.txt"))  # Output: File not found!
```

Best Practices for Exception Handling

Catch Specific Exceptions: Always catch specific exceptions instead of a generic Exception to handle errors more precisely.

Use Finally Block: Ensure that necessary cleanup (e.g., closing files or releasing resources) is performed by using the finally block.

Avoid Silent Failures: Do not use empty except blocks; always provide some logging or error message.

Log Exceptions: Use logging to record exceptions for future debugging and monitoring.

Use Custom Exceptions: Define custom exceptions for specific error conditions in your application to provide more meaningful error handling.

```
def get_list_element(lst, index):
    try:
        return lst[index]
    except IndexError as e:
        return f"IndexError: {e}"

my_list = [1, 2, 3]
print(get_list_element(my_list, 2))  # Output: 3
print(get_list_element(my_list, 5))  # Output: IndexError: list index out of range
```