Day-13 - Exception Handling in Production-Level Code

1. Introduction to Exception Handling

In software development, errors are inevitable. Exception handling is a structured way to manage errors that occur during the execution of a program. It ensures that the application does not crash unexpectedly and provides a controlled response to errors.

Why is Exception Handling Important in Production Code?

1. **Prevents Application Crashes** – A single unhandled exception can terminate an entire application. Proper exception handling keeps the program running.
2. **Improves User Experience** – Instead of showing cryptic error messages, users receive meaningful feedback.
3. **Simplifies Debugging** – Logging exceptions helps developers track issues efficiently.
4. **Maintains Data Integrity** – Prevents incomplete transactions or data corruption.
5. **Enhances Security** – Avoids exposing sensitive system details in error messages.

---

2. Exception Handling Concepts

2.1 What is an Exception?

An exception is an event that occurs during program execution and disrupts the normal flow. In Python, exceptions are objects that inherit from the base class Exception.

2.2 Types of Exceptions

1. **Built-in Exceptions** – Python has several predefined exceptions, such as:
    - ZeroDivisionError (e.g., 10 / 0)
    - ValueError (e.g., converting a non-numeric string to an integer)
    - IndexError (e.g., accessing an out-of-range list index)
    - KeyError (e.g., accessing a non-existent dictionary key)
    - FileNotFoundError (e.g., opening a missing file)

- TimeoutError (e.g., API or database request timeout)
- TypeError (e.g., adding an integer to a string)

2. **User-Defined Exceptions** – Developers can create custom exceptions to enforce business logic.

---

3. **Exception Handling Mechanism**
   Exception handling is a crucial concept in Python that allows a program to handle unexpected errors gracefully without crashing. It helps maintain the flow of execution by catching and handling runtime errors.

Python provides a **try-except-finally** mechanism to handle exceptions efficiently.

3.1 try-except Block

- The try block contains code that may raise an exception.
- The except block catches and handles the error.

Example 1: Handling Division by Zero

```
try:
    num1 = int(input("Enter a number: "))
    num2 = int(input("Enter another number: "))
    result = num1 / num2  # May cause ZeroDivisionError
    print("Result:", result)
except ZeroDivisionError:
    print("Error: Cannot divide by zero!")
finally:
    print("Execution completed.")
```

💡 **Key Takeaways:**

- The **try** block contains code that may raise an exception.
- The **except** block handles the exception if it occurs.
- The **finally** block executes whether an exception occurs or not.

2. Handling Multiple Exceptions

A program may raise different types of exceptions, and we can handle them using multiple except blocks.

Example 2: Handling Different Types of Exceptions

```
try:
        num1 = int(input("Enter a number: "))
        num2 = int(input("Enter another number: "))
        result = num1 / num2
        print("Result:", result)
        values = [10, 20, 30]
        print(values[5])  # IndexError

except ZeroDivisionError:
        print("Error: Cannot divide by zero!")
except ValueError:
        print("Error: Invalid input! Please enter numbers only.")
except IndexError:
        print("Error: List index is out of range.")
except Exception as e:
        print(f"An unexpected error occurred: {e}")
finally:
        print("Execution completed.")
```

Key Takeaways:

- You can have multiple except blocks to handle different exceptions.
- The generic except Exception as e: block catches any unexpected errors.
- The finally block runs regardless of whether an exception occurs or not.

3. Raising Exceptions

Sometimes, you may want to raise exceptions manually using the raise keyword.

Example 3: Raising a Custom Exception

```
def check_age(age):
  if age < 18:
    raise ValueError("Age must be 18 or above.")
  else:
    print("Access granted.")
try:
        user_age = int(input("Enter your age: "))
          check_age(user_age)
except ValueError as e:
        print(f"Error: {e}")
```

Key Takeaways:

- The raise statement is used to trigger an exception manually.
- It can be used for validation and enforcing constraints.

Conclusion

✅ **try-except-finally** allows us to handle exceptions effectively.
✅ **Handling multiple exceptions** ensures robust error handling.
✅ **Raising exceptions** helps enforce rules and constraints.

✅ **Use Specific Exceptions** – Avoid catching generic Exception, use KeyError, ValueError, etc.
✅ **Use Logging** – Print statements should be replaced with logs in production.
✅ **Use finally for Cleanup** – Close database connections, files, and API sessions.
✅ **Handle Expected Exceptions** – Check user input before processing.
✅ **Avoid Silent Failures** – Always log or display error messages.
✅ **Implement Retry Mechanisms** – Use try-except inside loops for transient errors.
✅ **Raise Meaningful Exceptions** – Provide clear messages for debugging and user understanding.