# Exception Handling

---

## 🔹 Introduction to Exceptions

### ✅ What is an Exception?

An **exception** is an error that occurs **during program execution**, interrupting the normal flow of instructions.

When Python runs your code and encounters an operation it **can't perform** (like dividing by zero, or trying to open a non-existent file), it **raises an exception**.

If the exception is **not handled**, the program **stops abruptly** and shows an error message (called a *traceback*).

---

### 🔸 Example (Without Handling)

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

b = int(input("Enter another number: "))

print(a / b)
```

If you enter b = 0, the output is:

```
ZeroDivisionError: division by zero
```

Here, Python stops execution due to an unhandled exception.

---

## 🔹 How to Handle Exceptions

Python provides special **keywords** to manage errors gracefully:

- try
- except
- else
- finally

---

### ✅ 1. try Block

The code that **might raise an error** is placed inside a try block.

```
try:
    risky_code_here
```

---

### ✅ 2. except Block

The code that **handles the error** goes inside an except block.

```
try:
    print(10 / 0)
except ZeroDivisionError:
    print("You cannot divide by zero!")
```

🧩 **Output:**

```
You cannot divide by zero!
```

---

### ✅ 3. else Block

If **no exception** occurs, code under else will execute.

```
try:
    num = int(input("Enter number: "))
    print(10 / num)
except ZeroDivisionError:
    print("Cannot divide by zero.")
else:
    print("Division successful!")
```

🧩 **If user enters 5:**

```
2.0
Division successful!
```

---

### ✅ 4. finally Block

The finally block **always executes** — whether or not an error occurs.

It is mainly used for **cleanup operations** like closing files, releasing resources, disconnecting from databases, etc.

```
try:
    f = open("data.txt", "r")
    content = f.read()
except FileNotFoundError:
    print("File not found.")
finally:
    print("This runs no matter what!")
```

🧩 **Output:**

File not found.

This runs no matter what!

---

🧱 **Understanding Multiple Exceptions**

In real programs, multiple types of errors may occur.
We can handle each **separately** or **together**.

---

✅ **1. Handling Different Exceptions Separately**

```
try:
    a = int(input("Enter number: "))
    b = int(input("Enter another: "))
    print(a / b)
except ZeroDivisionError:
    print("Division by zero not allowed.")
except ValueError:
    print("Invalid input! Please enter numbers only.")
```

---

✅ **2. Handling Multiple Exceptions Together**

If multiple exception types need the same handling, use a **tuple**.

```
try:
    x = int(input("Enter number: "))
    y = int(input("Enter another: "))
    print(x / y)
except (ZeroDivisionError, ValueError) as e:
    print("An error occurred:", e)
```

---

### ✅ 3. Generic Exception Handling

If you are unsure which error might occur, use the general Exception class.

```
try:
    print(10 / 0)
except Exception as e:
    print("Error:", e)
```

This catches **all exception types**, but should be used carefully — it can hide real bugs.

---

### ⚙️ Custom Exceptions

You can create your **own exceptions** to handle specific situations in your program.
They make your code more meaningful and professional.

---

### ✅ How to Create a Custom Exception

You can define one by creating a new class that inherits from Python's built-in Exception class.

```
class NegativeNumberError(Exception):
    """Raised when a negative number is entered."""
    pass
```

---

### ✅ Using Custom Exceptions

```
try:
    num = int(input("Enter a positive number: "))
    if num < 0:
        raise NegativeNumberError("Negative numbers are not allowed!")
except NegativeNumberError as e:
    print("Custom Exception:", e)
else:
    print("You entered:", num)
finally:
    print("Program finished.")
```

🧩 **If user enters -5:**

```
Custom Exception: Negative numbers are not allowed!
Program finished.
```

---

🧠 **Why Use Custom Exceptions?**

- To make error messages **more descriptive**
- To handle **domain-specific errors** (e.g., "InsufficientBalanceError" in a bank app)
- To make debugging and maintenance **easier**

---

🔍 **Example: Handling Multiple Custom Exceptions**

```
class AgeTooSmallError(Exception):
    pass

class AgeTooLargeError(Exception):
    pass

try:
    age = int(input("Enter your age: "))
```

```
    if age < 18:
        raise AgeTooSmallError("Age too small! You must be 18 or older.")
    elif age > 100:
        raise AgeTooLargeError("Age too large! Please enter a valid age.")
    else:
        print("Age accepted:", age)
except (AgeTooSmallError, AgeTooLargeError) as e:
    print("Custom Exception:", e)
finally:
    print("End of program.")
```

---

### 🧩 In Short

| **Block** | **Purpose** |
|---|---|
| try | Contains risky code that may raise an error |
| except | Handles specific or general exceptions |
| else | Executes if no exception occurs |
| finally | Executes always (used for cleanup) |
| raise | Manually trigger an exception |
| custom exception | User-defined error class for meaningful handling |