EXCEPTION HANDLING

Exception handling in Python is a simple and safe way to deal with errors that happen while a program is running. Instead of letting the entire program stop when something goes wrong—like wrong input, missing files, or invalid calculations—it allows the program to switch to another path and handle the problem smoothly. With the help of blocks like try, except, else, and finally, the program can show meaningful error messages, guide the user correctly, fix or ignore mistakes, clean up resources, and continue running without crashing. This makes programs more reliable, user-friendly, and able to handle unexpected situations without breaking.

ADVANTAGES OF EXCEPTION HANDLING

1. Prevents Program Crashes- Instead of stopping the program when an error occurs, exception handling catches the error and allows the program to continue executing safely.

2. Improves Program Reliability - Programs become more stable and robust because unexpected errors are managed gracefully.

3. Simplifies Error Detection and Debugging- Exception messages help identify what went wrong and where, making debugging easier.

4. Separates Error-Handling Code from Regular Code- It keeps normal logic clean and focused while placing error-handling logic separately in except blocks.

5. Ensures Resource Cleanup (finally block) - Files, database connections, network links, etc. can be closed properly, even if an error occurs.

6. Allows Handling of Multiple Types of Errors - Different exceptions can be handled independently (ValueError, ZeroDivisionError, FileNotFoundError, etc.)

7. Supports Custom Error Messages- Meaningful, user-friendly messages can be displayed instead of confusing Python errors.

8. Encourages Professional-Quality Code- Exception handling is a standard coding practice in real-world development, used in APIs, databases, web apps, and automation.

Uses: Input validation, file handling, network/database operations, arithmetic error handling, web/API requests, custom business rules, cleanup operations, and developing user-friendly applications.

A compile-time error is an error that occurs before the program begins execution and is caused by incorrect syntax, improper indentation, missing symbols, or invalid structure in the code. These errors are detected by the Python interpreter during the initial code-checking phase, and the program cannot run until they are corrected.

On the other hand, **a runtime error,** also called an exception, occurs while the program is executing. The code is syntactically correct, but an invalid operation—such as dividing by zero, accessing an invalid index, using incompatible data types, or opening a non-existent file—causes the program to stop abruptly.

Unlike compile-time errors, runtime errors can be gracefully handled using exception handling mechanisms (try–except–finally), allowing the program to continue running or fail safely. In simple terms, compile-time errors stop the program from starting, whereas runtime errors occur after the program has started and can be managed during execution**.**

TYPES OF EXCEPTIONS IN PYTHON

Python exceptions are mainly divided into two categories:

1. Built-in Exceptions - These are already provided by Python.
Examples: ValueError, TypeError, ZeroDivisionError, IndexError, etc.

2. User-Defined (Custom) Exceptions - Created by the programmer using classes.
Used when built-in exceptions are not enough.

Exception	***Meaning (Easy Explanation)***
ValueError	Value is wrong
TypeError	Types don’t match
ZeroDivisionError	Dividing by zero
IndexError	Invalid index
KeyError	Key not present
FileNotFoundError	File missing
NameError	Variable not defined
AttributeError	Attribute/function not available
ImportError	Cannot import module

1. try Block - Put the risky code here…Python first runs the try block.

Example:

```
try:
   x = 10 / 0
```

This will throw an error. Now we need except!

2. except Block - What to do **if an error happens**.

Syntax:

```
except:
   print("Error happened")
```

Example:

```
try:
   x = 10 / 0
except:
   print("Cannot divide by zero!")
```

Good for beginners BUT Not recommended always (too broad)

a) except with Specific Exception:- You can catch *only one type* of error.

```
except ExceptionType:
   # handle it
```

Example:

```
try:
   num = int("hello")
except ValueError:
   print("Invalid number!")
```

Why use it?

- More accurate
- Avoid catching unwanted errors
- Good practice in real coding

b) Multiple except blocks (for different errors):- You can handle **different errors separately**.

```
try:
   x = int("abc") / 0
except ValueError:
   print("Conversion error")
except ZeroDivisionError:
   print("Cannot divide by zero")
```

Python will match the FIRST correct exception type.

c) Multiple Exceptions in One except (Tuple)- If you want the same message for multiple errors:

```
try:
   x = int("abc") / 0
except (ValueError, ZeroDivisionError):
   print("Error occurred in calculation")
```

Good for **grouping** similar exceptions.

d)Using Exception Class(except Exception as e:) - This catches **almost all errors**, except very low-level ones. Useful when you want to log or debug.

Example:

```
try:
    x = int("abc")
except Exception as e:
    print("Error:", e)
```

Here `e` shows the actual error message.

Explaination - **except Exception as e** is used to catch almost all types of errors in Python because most built-in exceptions (like ValueError, TypeError, ZeroDivisionError, FileNotFoundError, etc.) are subclasses of the main Exception class. While we can catch specific errors individually, using except Exception as e is helpful when we don't know exactly which error may occur or when we want one common block to handle many possible errors. The as e part stores the actual error message in a variable so we can print it, debug it, or log it. This approach is safer than using a plain except: because it avoids catching system-level interruptions, and it gives more information than writing separate except blocks for everything. In short, we use it because it provides a clean, general, and informative way to handle multiple exceptions without crashing the program.

4. *Else Block* - Runs **only when NO exception occurs** in the try block.

```
try:
   # risky code
except:
   # error handling
else:
   # runs if everything is fine
```

Example:

```
try:
   x = 5 + 5
except:
   print("Error")
else:
   print("All good! Result =", x)
```

5. finally Block - Runs **whether there is an exception or not**.
Perfect for cleanup operations.

Syntax:

```
finally:
   # always runs
```

Example:

```
try:
   file = open("data.txt")
except:
   print("File not found")
finally:
   print("Execution Ended")
```

6. raise Statement

Used to **manually throw** an exception. This forces Python to create an error and jump into the except block.

In simple words

1. raise = “Create your own error on purpose.”

2. It stops the normal flow of the program.
3. It sends the error to the nearest matching except block
4. It allows you to show meaningful messages (like real apps).

Example

```
age = 15
if age < 18:
   raise ValueError("You must be 18+ to continue")
```

Python didn't know the rule.
YOU told it:
"This situation is wrong — throw an error."

One-Line Exam Definition Summary

Concept	**Short, Exam-Friendly Definition**
try	Block that contains risky code.
except	Handles the error if it occurs.
except ExceptionType	Catches only a specific type of error.
else	Runs only if no exception occurs.
finally	Always runs, used for cleanup.
raise	Manually trigger an exception.

REAL-LIFE EXAMPLE — ATM CASH WITHDRAWAL

```
def withdraw_money(balance):
   print("Welcome to the ATM!")


   try:
      amount = int(input("Enter amount to withdraw: "))  # risky: wrong input type

      result = balance - amount                    # risky: negative balance

      if result < 0:
         # This error will come naturally
         x = 1 / 0     # simulating an error like insufficient balance
```

```
    except ValueError:
        print("Invalid input! Please enter numbers only.")    # specific exception

    except ZeroDivisionError:
        print("Insufficient balance!")                    # specific exception

    except Exception as e:
        print("Unexpected error occurred:", e)             # general exception

    else:
        print("Withdrawal successful! Remaining balance =", result)  # runs if no error

    finally:
        print("Thank you for using our ATM.")              # always runs

# calling the function
withdraw_money(500)
```

WHY THIS IS A REAL-LIFE EXAMPLE

- **User enters wrong amount (string)** → ValueError
- **Withdrawal exceeds balance** → we trigger a natural error (division) → ZeroDivisionError
- **Everything correct** → Money withdrawn smoothly
- **finally block** → ATM closes session properly every time

```
def withdraw_money(balance):
    print("Welcome to the ATM!")

    try:
        amount = int(input("Enter amount to withdraw: "))

        if amount <= 0:
            raise ValueError("Amount must be greater than zero.")
```

```
        if amount > balance:
            raise Exception("Insufficient balance.")

        result = balance - amount

    except ValueError as e:
        print("Input Error:", e)

    except Exception as e:
        print("Transaction Error:", e)

    else:
        print("Withdrawal successful! Remaining balance =", result)

    finally:
        print("Thank you for using our ATM.")

# calling the function
withdraw_money(500)
```

Using raise allows us to manually create errors with meaningful messages. Instead of using a fake error like division by zero, we use raise to tell Python exactly why something went wrong. This makes the program cleaner, realistic, and easier to understand.

If multiple except blocks are written but only one exception occurs, Python executes only the matching except block and skips all others. Order matters → Python checks them top to bottom. If no except matches → program crashes with the original error