Exception & Exception Handling in Python

What is an Exception in Python?

An **exception** is an **error** that occurs during the **execution** of a program.
When Python encounters an error (like dividing by zero, accessing an undefined variable, etc.), it **raises an exception**.
If the exception is **not handled**, the program will **stop** (terminate) and display an **error message** (called a **traceback**).

Why do we use Exception?

- Prevent program crashes — handle errors instead of stopping the program.
- Maintain normal program flow — allows the program to continue running after an error.
- Show user-friendly error messages — instead of confusing technical errors.
- Easier debugging and maintenance — helps identify where and why errors occur.
- Handle unexpected situations — like missing files, invalid input, or network errors.
- Improve reliability and robustness — makes programs safer and more professional.

Common Exceptions in Python

Exception Type	Description
ZeroDivisionError	Division by zero
ValueError	Invalid value for a function
TypeError	Wrong data type used
IndexError	List index out of range
KeyError	Dictionary key not found
FileNotFoundError	File doesn't exist

1. ZeroDivisionError
Raised when dividing by zero.
print(10 / 0)

Output: ZeroDivisionError: division by zero

Exception & Exception Handling in Python

2. OverflowError
Occurs when a numerical result is too large.

```
import math
print(math.exp(1000))
```

Output: OverflowError: math range error

3. FloatingPointError
Raised when a floating-point operation fails (rarely used).

4. AssertionError
Raised when an assert statement fails.

```
x = 5
assert x > 10, "x must be greater than 10"
```

Output: AssertionError: x must be greater than 10

5. IndexError
Raised when accessing a list index that doesn't exist.

```
lst = [1, 2, 3]
print(lst[5])
```

Output: IndexError: list index out of range

6. KeyError
Raised when a dictionary key doesn't exist.

```
d = {"name": "Alice"}
print(d["age"])
```

Output: KeyError: 'age'

7. TypeError
Raised when operation is applied to wrong type.

```
print("2" + 3)
```

Output: TypeError: can only concatenate str (not "int") to str

8. MemoryError
Raised when an operation runs out of memory.

```
# Creating a huge list (be careful running this)
# x = [1] * (10**10)
```

9. Value Error:

Exception & Exception Handling in Python

A ValueError occurs when a function receives the correct data type, but the value is inappropriate or invalid for that operation.

```
num = int("12.5")
```

Output:
ValueError: invalid literal for int() with base 10: '12.

Exception Handling

Exception handling in Python is a mechanism used to detect, manage, and respond to runtime errors (called exceptions) so that the program doesn't crash unexpectedly.
It allows you to control what happens when an error occurs, ensuring the program continues to run smoothly or fails gracefully.

Python provides a structured mechanism for this using:

Keyword	Purpose
try	Block of code that may raise an exception
except	Block that handles the exception
else	Runs only if no exception occurs
finally	Always executes (for cleanup or closing resources)
raise	Used to raise an exception manually
assert	Used to check conditions (for debugging)

1. try-except
Purpose: Used to **catch and handle** exceptions gracefully — so your program doesn't crash.

Example:

```
try:
    x = int(input("Enter a number: "))
    y = 10 / x
    print("Result:", y)
except ZeroDivisionError:
    print("Error: Cannot divide by zero!")
except ValueError:
    print("Error: Please enter a valid integer.")
```

Exception & Exception Handling in Python

Explanation:
- The code inside the try block may raise an exception.
- If you enter 0 → ZeroDivisionError will be caught.
- If you enter "abc" → ValueError will be caught.
- If no exception occurs → result is printed normally.

2. **else**

Purpose: The else block runs **only if no exception occurs** in the try block.

Example:

```
try:
    num = int(input("Enter a number: "))
except ValueError:
    print("Error: Invalid input!")
else:
    print("Success! You entered:", num)
```

🧠 **Explanation:**
- If you enter "abc" → ValueError → runs the except block.
- If you enter 5 → no error → runs the else block.

3. **finally**

Purpose: The finally block **always executes**, whether an exception occurs or not.
Commonly used for **cleanup** (closing files, releasing resources, etc.)

Example:

```
try:
    f = open("myfile.txt", "r")
    data = f.read()
except FileNotFoundError:
    print("File not found!")
else:
    print("File contents:", data)
finally:
    print("Closing file (if open).")
```

Explanation:
- If file doesn't exist → FileNotFoundError caught.
- If file exists → content is read.
- Either way, finally runs → ensures the file is closed or cleanup happens.

4. **raise**

Purpose: Used to **manually raise an exception** in your code when something invalid happens (like invalid input or failed validation).

Exception & Exception Handling in Python

Example:

```
age = int(input("Enter your age: "))
if age < 18:
    raise ValueError("You must be at least 18 years old.")
else:
    print("Access granted!")
```

Explanation:

- raise generates a ValueError intentionally if the condition fails.
- Program stops and displays the message:

ValueError: You must be at least 18 years old.

5. assert

Purpose: Used for **debugging** — it checks a condition and raises an AssertionError if the condition is false.
It's like saying "this **must** be true."

Example:

```
x = int(input("Enter a number greater than 0: "))
assert x > 0, "Number must be positive!"
print("Valid number:", x)
```

Explanation:

- If you enter 5 → assertion passes → prints "Valid number: 5"
- If you enter -3 → fails the assertion → raises

AssertionError: Number must be positive!

Combined Example: try-except-else-finally-raise-assert

Let's combine everything into one meaningful program:

```
def divide_numbers()
    try:
        a = int(input("Enter numerator: "))
        b = int(input("Enter denominator: "))

        # Assertion — ensure denominator is not negative
        assert b >= 0, "Denominator must be non-negative!"

        if b == 0:
            raise ZeroDivisionError("Denominator cannot be zero!")  # manual raise

        result = a / b
```

Exception & Exception Handling in Python

```
except ValueError:
    print("Error: Please enter valid integers.")
except AssertionError as ae:
    print("Assertion Error:", ae)
except ZeroDivisionError as ze:
    print("Division Error:", ze)
else:
    print("Division successful! Result =", result)
finally:
    print("Operation completed. (finally block executed)")
```

Example Runs:

Case 1: Valid input
Enter numerator: 10
Enter denominator: 2
Division successful! Result = 5.0
Operation completed. (finally block executed)

Case 2: Non-integer input
Enter numerator: ten
Error: Please enter valid integers.
Operation completed. (finally block executed)

Case 3: Denominator is zero
Enter numerator: 10
Enter denominator: 0
Division Error: Denominator cannot be zero!
Operation completed. (finally block executed)

Case 4: Assertion failure
Enter numerator: 10
Enter denominator: -5
Assertion Error: Denominator must be non-negative!
Operation completed. (finally block executed)

Exception & Exception Handling in Python

Summary Table:

Keyword	Purpose	Runs When	Example
try	Code that may cause an error	Always	try: x=1/0
except	Handles specific or general errors	Only if an exception occurs	except ZeroDivisionError:
else	Runs when no exception occurs	Only if try is successful	else: print("Success")
finally	Cleanup actions	Always (even if error)	finally: file.close()
raise	Manually trigger an exception	On programmer request	raise ValueError("error")
assert	Debug condition must be true	If false, raises AssertionError	assert x>0, "must be positive"