Python's exception handling system is both powerful and elegant, designed to help developers write robust, error-resilient code. Here's a **comprehensive breakdown** of all the key concepts you should know:

**🧠 Core Concepts of Exception Handling in Python**

**1. try Block**

* Contains code that might raise an exception.
* If an error occurs, control jumps to the corresponding except block.

try:

risky\_code()

**2. except Block**

* Catches and handles specific exceptions.
* You can have multiple except blocks for different error types.

except ValueError:

print("Caught a ValueError")

except ZeroDivisionError:

print("Can't divide by zero")

**3. Catching Multiple Exceptions**

* You can catch multiple exceptions in one block using a tuple.

except (ValueError, TypeError) as e:

print(f"Error occurred: {e}")

**4. Generic Exception Catching**

* Use except: to catch any exception (not recommended unless necessary).

except:

print("Something went wrong")

**5. else Block**

* Runs only if no exception was raised in the try block.

try:

print("No errors here")

except:

print("Error occurred")

else:

print("Success!")

**6. finally Block**

* Always executes, whether an exception occurred or not.
* Great for cleanup tasks like closing files or releasing resources.

finally:

print("This always runs")

**7. Raising Exceptions Manually**

* Use raise to trigger exceptions intentionally.

raise ValueError("Invalid input")

**8. Custom Exceptions**

* Define your own exception classes by inheriting from Exception.

class MyCustomError(Exception):

pass

raise MyCustomError("Something custom went wrong")

**9. Built-in Exceptions**

Python has many built-in exceptions like: | Exception Type | Description | |----------------------|--------------------------------------| | ValueError | Invalid value | | TypeError | Wrong data type | | IndexError | List index out of range | | KeyError | Missing dictionary key | | ZeroDivisionError | Division by zero | | FileNotFoundError | File not found | | ImportError | Module not found |

**🔒 Best Practices**

* Catch specific exceptions, not generic ones.
* Use finally for resource cleanup.
* Log errors instead of just printing them.
* Avoid suppressing exceptions unless absolutely necessary.

File handling in Python refers to the process of **creating, reading, writing, updating, and deleting files** using built-in functions. It allows your programs to interact with data stored on disk, making it essential for tasks that require **data persistence**—meaning the data remains available even after the program ends.

**📂 What Is File Handling?**

At its core, file handling involves:

* **Opening a file** using open()
* **Reading from a file** using methods like .read(), .readline(), or .readlines()
* **Writing to a file** using .write() or .writelines()
* **Closing a file** using .close() (or automatically with with statement)

Example:

with open("example.txt", "w") as file:

file.write("Hello, world!")

This creates a file called example.txt and writes a line of text into it.

**🛠️ Why Is File Handling Used?**

Here are the most common and practical uses:

**1. Data Storage**

* Save user data, logs, configurations, or results.
* Example: Saving scores in a game or user preferences in an app.

**2. Data Processing**

* Read large datasets (CSV, JSON, TXT) for analysis.
* Example: Processing sales data or sensor logs.

**3. Automation Scripts**

* Generate reports, logs, or backups automatically.
* Example: Writing daily summaries or exporting scraped data.

**4. Communication Between Programs**

* Share data between different parts of a system or across platforms.
* Example: One script writes a file, another reads it to continue processing.

**5. Web and Desktop Applications**

* Handle uploads, downloads, and user-generated content.
* Example: Saving uploaded profile pictures or documents.

List comprehension in Python is a **concise and elegant way to create lists** by applying an expression to each item in an iterable (like a list, string, or range), optionally filtering with a condition.

**🧪 Basic Syntax**

[expression for item in iterable if condition]

* expression: What to do with each item (e.g., transform it)
* item: The variable representing each element
* iterable: The source (e.g., list, range)
* condition: Optional filter to include only certain items

**✅ Examples**

**1. Simple Transformation**

squares = [x\*\*2 for x in range(5)]

# Output: [0, 1, 4, 9, 16]

**2. With Condition**

evens = [x for x in range(10) if x % 2 == 0]

# Output: [0, 2, 4, 6, 8]

**3. From a String**

vowels = [char for char in "hello world" if char in "aeiou"]

# Output: ['e', 'o', 'o']

**4. Nested Loops**

pairs = [(x, y) for x in range(3) for y in range(2)]

# Output: [(0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)]

**🧠 Why Use List Comprehension?**

* **Cleaner syntax** than traditional loops
* **Faster execution** in most cases
* **More readable** for simple transformations

def my\_decorator(func):

def wrapper():

print("Before the function runs")

func()

print("After the function runs")

return wrapper

@my\_decorator

def say\_hello():

print("Hello!")

say\_hello()

Decorators in Python are a powerful and elegant way to **modify or enhance the behavior of functions or methods**—without changing their actual code. Think of them as wrappers that add extra functionality before or after a function runs.

🧠 What Is a Decorator?

A **decorator** is a function that takes another function as input and returns a new function with added behavior.