Day 1: Python Basics & Setup

1. Introduction to Python

Python is a **high-level, interpreted, dynamically typed, and garbage-collected** programming language. It is widely used in:

- **Data Engineering & Data Science**
- **Web Development**
- **Automation & Scripting**
- **Artificial Intelligence & Machine Learning**
- **Software Development**

Why Use Python?

✔ **Simple & Readable** – Uses indentation instead of `{}` (like C, Java).
✔ **Cross-Platform** – Works on Windows, macOS, and Linux.
✔ **Dynamically Typed** – No need to declare variable types explicitly.
✔ **Huge Libraries** – Supports Pandas, NumPy, SQLAlchemy, TensorFlow, etc.
✔ **Strong Community Support** – Extensive documentation and resources available.

---

2. Installing Python

Option 1: Standard Python Installation

1. Download Python from python.org.
2. Install Python and ensure you check the option **"Add Python to PATH"**.

Open Command Prompt (Windows) or Terminal (Mac/Linux) and verify installation:
python --version

3.

Option 2: Using Anaconda (Recommended for Data Engineering)

1. Download and install **Anaconda** from anaconda.com.
2. Open **Anaconda Navigator** and launch **Jupyter Notebook**.

Verify Python installation in Jupyter Notebook by running:

```
 import sys
print(sys.version)
```

3.

---

3. Running Python Code

Python code can be executed in different ways:

- **Interactive Mode:** Open terminal and type `python`, then enter Python commands.

Script Mode: Save a file as `script.py` and run it using:

```
 python script.py
```

-
- **Jupyter Notebook:** Run code interactively using cells.
- **VS Code or PyCharm:** Use IDEs for better development.

---

4. Python Syntax & Structure

Python relies on **indentation** instead of curly brackets `{}`.

Basic Example:

```
# This is a comment
print("Hello, Python!")  # Output: Hello, Python!
```

Indentation Example:

```
if 10 > 5:
    print("10 is greater than 5")
```

Note: Incorrect indentation leads to an error.

---

5. Variables and Data Types

Declaring Variables

Python does not require explicit variable declarations:

```
x = 10       # Integer
pi = 3.14    # Float
name = "Alice"  # String
is_valid = True  # Boolean
```

Checking Data Types

```
print(type(x))    # Output: <class 'int'>
print(type(pi))   # Output: <class 'float'>
print(type(name)) # Output: <class 'str'>
print(type(is_valid)) # Output: <class 'bool'>
```

---

6. Python Operators

Arithmetic Operators

```
a = 10
b = 3
print(a + b)  # Addition -> 13
print(a - b)  # Subtraction -> 7
print(a * b)  # Multiplication -> 30
print(a / b)  # Division -> 3.33
print(a // b) # Floor Division -> 3
print(a % b)  # Modulus -> 1
print(a ** b) # Exponentiation -> 1000
```

Comparison Operators

```
x = 5
y = 10
print(x > y)   # False
print(x < y)   # True
print(x == y)  # False
print(x != y)  # True
```

Logical Operators

```
x = True
y = False
print(x and y)  # False
print(x or y)   # True
print(not x)    # False
```

---

7. Taking User Input

The `input()` function is used to take user input.

```
name = input("Enter your name: ")
print("Hello,", name)
```

Note: Input is always treated as a string unless converted explicitly.

```
age = int(input("Enter your age: "))
print("Your age in 5 years will be:", age + 5)
```

---

8. Printing Output in Python

The `print()` function is used to display output.

```
print("Hello, World!")
print("My name is", name)
```

Formatted Output (f-strings)

```
name = "Alice"
age = 25
print(f"My name is {name} and I am {age} years old.")
```

---

9. Type Conversion in Python

Python provides built-in functions to convert data types.

```
# Convert string to integer
```

```
num_str = "100"
num_int = int(num_str)
print(num_int, type(num_int))  # Output: 100 <class 'int'>
```

Common Type Conversion Functions:

Function	Purpose
`int(x)`	Converts `x` to an integer
`float(x)`	Converts `x` to a floating point number
`str(x)`	Converts `x` to a string
`bool(x)`	Converts `x` to a boolean

---

10. Comments in Python

Comments help in documenting code.

```
# This is a single-line comment

"""
This is a
multi-line comment
"""
```

Summary of Day 1

✅ Installed Python and set up environment
✅ Learned about Python syntax and indentation
✅ Understood variables and data types
✅ Explored operators in Python
✅ Used `input()` and `print()` functions
✅ Performed basic type conversions
✅ Learned about comments

📌 Day 2: Control Flow (Conditions & Loops) – Detailed Notes

🔹 Conditional Statements (if-elif-else)

Conditional statements allow the program to make decisions based on certain conditions.

- **if Statement**

The `if` statement executes a block of code only if the condition is True.

- **Syntax:**

```
if condition:
    # Code block executes if condition is True
```

- **Example:**

```
x = 10
if x > 5:
    print("x is greater than 5")  # Output: x is greater than 5
```

- **if-else Statement**

The `else` block runs if the `if` condition is False.

- **Syntax:**

if condition:

**   # Executes if True**

else:

**   # Executes if False**

🔹 **Example:**

num = 10

if num % 2 == 0:

**   print("Even number")**

else:

**   print("Odd number")**

✅ **Output: Even number**

🔹 **if-elif-else Statement**

The `elif` statement allows checking multiple conditions.

🔹 **Syntax:**

if condition1:

**   # Executes if condition1 is True**

elif condition2:

**   # Executes if condition2 is True**

```
else:
    # Executes if none of the conditions are True
```

🔹 **Example:**

```
marks = 85
if marks >= 90:
    print("Grade: A")
elif marks >= 75:
    print("Grade: B")
elif marks >= 50:
    print("Grade: C")
else:
    print("Grade: F")
```

✅ **Output: Grade: B**

🔹 **Nested if Statements**

You can have an `if` inside another `if` statement (nested condition).

🔹 **Example:**

```
x = 20
if x > 10:
```

```
    print("Above 10")
    if x > 15:
        print("Above 15")
```

✅ **Output:**

```
Above 10
Above 15
```

🔹 Loops in Python

Loops are used to repeat a block of code multiple times.

🔹 for Loop

Used to iterate over sequences (lists, tuples, strings, etc.).

🔹 Syntax:

```
for variable in sequence:
    # Loop body
```

🔹 Example:

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

✅ **Output:**

apple

banana

cherry

🔹 **Using range() in for Loops**

The `range()` function generates numbers.

🔹 **Example:**

```
for i in range(5):
    print(i)
```

✅ **Output:**

0

1

2

3

4

🔹 **range(start, stop, step)**

- `range(5)` → 0 to 4
- `range(1, 6)` → 1 to 5
- `range(0, 10, 2)` → 0, 2, 4, 6, 8

🔹 **while Loop**

Executes a block as long as the condition is True.

🔹 **Syntax:**

```
while condition:
    # Loop body
```

🔹 **Example:**

```
x = 5
while x > 0:
    print(x)
    x -= 1
```

✅ **Output:**

```
5
4
3
2
1
```

🔹 Loop Control Statements

🔹 break Statement

Stops the loop immediately.

🔹 Example:

```
for i in range(10):
    if i == 5:
        break
    print(i)
```

✅ **Output:**

```
0
1
2
3
4
```

🔹 continue Statement

Skips the current iteration and moves to the next.

🔹 **Example:**

```
for i in range(5):
    if i == 2:
        continue
    print(i)
```

✅ **Output:**

```
0
1
3
4
```

🔹 **pass Statement**

Used as a placeholder when a loop or function is required but no code is written yet.

🔹 **Example:**

```
for i in range(5):
    if i == 3:
        pass  # Placeholder
    print(i)
```

🔹 else with Loops

The `else` block runs only if the loop completes normally (without a `break`).

🔹 **Example:**

```
for i in range(5):
    print(i)
else:
    print("Loop completed")
```

✅ **Output:**

```
0
1
2
3
4
Loop completed
```

If a `break` is encountered, the `else` part does not execute.

🔹 Nested Loops

A loop inside another loop.

🔹 Example:

```
for i in range(3):
    for j in range(2):
        print(f"i={i}, j={j}")
```

✅ Output:

```
i=0, j=0
i=0, j=1
i=1, j=0
i=1, j=1
i=2, j=0
i=2, j=1
```

🔹 Summary Table

Concept	Description
`if` statement	Executes if condition is True

`if-else`	Runs either `if` or `else` block
`if-elif-else`	Checks multiple conditions
`for` loop	Iterates over sequences
`while` loop	Runs while the condition is True
`break`	Exits the loop
`continue`	Skips current iteration
`pass`	Placeholder with no effect
`else` in loop	Runs if loop completes without `break`
Nested loops	Loop inside another loop

✅ Key Takeaways

- `if-elif-else` is used for decision-making.
- `for` loops iterate over sequences.

- `while` loops run until the condition is False.
- `break` stops the loop, `continue` skips an iteration.
- `pass` is used as a placeholder.

📌 Day 3: Python Data Structures – Detailed Notes

Python provides built-in data structures that help store and manage data efficiently. The most commonly used are:

1. **Lists (Ordered, Mutable)**
2. **Tuples (Ordered, Immutable)**
3. **Sets (Unordered, Unique Values)**
4. **Dictionaries (Key-Value Pairs, Fast Lookup)**

🔹 1. Lists in Python

Lists are ordered, mutable (modifiable), and allow duplicate values.

🔹 Creating a List

```
fruits = ["apple", "banana", "cherry"]

numbers = [1, 2, 3, 4, 5]

mixed = [1, "hello", 3.14, True]


print(fruits)

print(numbers)

print(mixed)
```

✅ **Output:**

['apple', 'banana', 'cherry']

[1, 2, 3, 4, 5]

[1, 'hello', 3.14, True]

🔹 **Accessing Elements**

```
fruits = ["apple", "banana", "cherry"]
print(fruits[0])  # apple
print(fruits[-1]) # cherry (last element)
```

🔹 **Modifying Lists**

```
fruits[1] = "blueberry"
print(fruits)  # ['apple', 'blueberry', 'cherry']
```

🔹 **List Methods**

Method	Description

`append(x)`	Adds item `x` at the end
`insert(i, x)`	Inserts `x` at index `i`
`remove(x)`	Removes first occurrence of `x`
`pop(i)`	Removes item at index `i` (default last)
`sort()`	Sorts the list in ascending order
`reverse()`	Reverses the list
`count(x)`	Counts occurrences of `x`

```
fruits.append("mango")  # Add item at end
fruits.insert(1, "orange")  # Insert at index 1
fruits.remove("cherry")  # Remove item
fruits.sort()  # Sort alphabetically
print(fruits)
```

✅ **Output:**

['apple', 'blueberry', 'mango', 'orange']

🔹 2. Tuples in Python

Tuples are ordered, immutable (unchangeable), and allow duplicates.

🔹 Creating a Tuple

```
colors = ("red", "green", "blue")

numbers = (10, 20, 30, 40)

mixed = (1, "hello", 3.14, True)


print(colors[1])  # green
```

🔹 Tuple Methods

Method	Description
`count(x)`	Counts occurrences of `x`

`index(x)` Returns index of `x`

```
numbers = (1, 2, 3, 2, 4)
print(numbers.count(2))  # 2 occurrences
print(numbers.index(3))  # Index: 2
```

🔹 **Tuple Packing & Unpacking**

```
person = ("John", 25, "Engineer")
name, age, job = person  # Unpacking
print(name)  # John
print(age)   # 25
print(job)   # Engineer
```

🔹 3. Sets in Python

Sets are unordered, mutable, and do not allow duplicate values.

🔹 **Creating a Set**

```
numbers = {1, 2, 3, 4, 5}
fruits = {"apple", "banana", "cherry"}
```

🔹 **Set Methods**

Method	Description
`add(x)`	Adds element `x`
`remove(x)`	Removes `x` (error if not found)
`discard(x)`	Removes `x` (no error if missing)
`union(set2)`	Combines two sets
`intersection(set2)`	Finds common elements
`difference(set2)`	Finds elements in one set but not another

```
A = {1, 2, 3}
B = {3, 4, 5}

print(A.union(B))        # {1, 2, 3, 4, 5}
print(A.intersection(B))  # {3}
print(A.difference(B))    # {1, 2}
```

🔹 4. Dictionaries in Python

Dictionaries store key-value pairs and allow fast lookup.

🔹 Creating a Dictionary

```
person = {
    "name": "Alice",
    "age": 30,
    "city": "New York"
}
```

🔹 Accessing Values

```
print(person["name"])  # Alice
print(person.get("age"))  # 30
```

🔹 Adding & Updating Values

```
person["job"] = "Engineer"  # Adding
person["age"] = 31  # Updating
print(person)
```

- **Dictionary Methods**

Method	Description
`keys()`	Returns all keys
`values()`	Returns all values
`items()`	Returns key-value pairs
`update(dict2)`	Merges `dict2` into dictionary

```
print(person.keys())  # dict_keys(['name', 'age', 'city', 'job'])

print(person.values())  # dict_values(['Alice', 31, 'New York', 'Engineer'])

print(person.items())  # dict_items([('name', 'Alice'), ('age', 31), ('city', 'New York'), ('job', 'Engineer')])
```

- **List vs Tuple vs Set vs Dictionary**

Feature	List	Tuple	Set	Dictionary

Ordered	✅	✅	❌	✅
Mutable	✅	❌	✅	✅
Duplicate Values	✅	✅	❌	**Keys ❌, Values ✅**
Indexing	✅	✅	❌	**Keys instead of index**

🔹 Summary

- **Lists: Ordered, mutable, allows duplicates**
- **Tuples: Ordered, immutable, allows duplicates**
- **Sets: Unordered, mutable, unique values**
- **Dictionaries: Key-value pairs, fast lookup**

✅ Key Takeaways

- **Use lists when order and mutability are needed.**
- **Use tuples when data should be immutable.**
- **Use sets to store unique values efficiently.**
- **Use dictionaries when you need key-value lookups.**

📌 Day 4: Functions in Python – Detailed Notes

🔹 What is a Function?

A function is a block of reusable code that performs a specific task. Instead of writing the same code multiple times, we can define a function and call it whenever needed.

🔹 Defining a Function

A function is defined using the `def` keyword.

🔹 Syntax:

```
def function_name(parameters):
    """Docstring - Describes what the function does."""
    # Function body
    return value  # (Optional)
```

🔹 Example:

```
def greet():
    """This function prints a greeting message."""
    print("Hello! Welcome to Python.")

# Calling the function
greet()
```

✅ **Output:**

Hello! Welcome to Python.

🔹 Function Parameters & Arguments

Functions can accept parameters (input values).

🔹 Positional Arguments

```
def add(a, b):
    return a + b


result = add(5, 3)
print(result)  # 8
```

- **Order matters when passing values to positional arguments.**

🔹 Default Arguments

We can provide default values to parameters. If no value is given during the function call, the default is used.

```
def greet(name="User"):
    print(f"Hello, {name}!")
```

```
greet()  # Hello, User!
greet("Alice")  # Hello, Alice!
```

🔹 Keyword Arguments

Arguments can be passed by name, making them order-independent.

```
def person_info(name, age):
    print(f"Name: {name}, Age: {age}")

person_info(age=25, name="Bob")
```

✅ Output:

```
Name: Bob, Age: 25
```

🔹 Variable-Length Arguments (*args & **kwargs)

Used when we don't know how many arguments will be passed.

🔹 `*args` (Non-keyword arguments)

- Allows passing multiple positional arguments as a tuple.

```
def add_numbers(*args):
    return sum(args)
```

```
print(add_numbers(1, 2, 3, 4))  # 10
print(add_numbers(10, 20))  # 30
```

🔹 `**kwargs` (Keyword arguments)

- Allows passing multiple named arguments as a dictionary.

```
def display_info(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

display_info(name="Alice", age=30, job="Engineer")
```

✅ Output:

```
name: Alice
age: 30
job: Engineer
```

🔹 Return Statement

- A function returns a value using `return`.
- If no return statement is present, the function returns `None`.

```
def square(n):
```

```
    return n ** 2


print(square(5))  # 25
```

🔹 Scope of Variables (Local & Global)

- **Local Variable: Defined inside a function, accessible only within that function.**
- **Global Variable: Defined outside functions, accessible throughout the program.**

```
x = 10  # Global variable


def test():
    y = 5  # Local variable
    print(x, y)


test()
# print(y)  # This will cause an error because y is local
```

✅ **Output:**

```
10 5
```

🔹 Lambda (Anonymous) Functions

A lambda function is a small, one-line function without a name.

🔹 **Syntax:**

lambda arguments: expression

🔹 **Example:**

```
square = lambda x: x ** 2

print(square(5))  # 25
```

🔹 **Multiple Arguments in Lambda:**

```
multiply = lambda a, b: a * b

print(multiply(4, 5))  # 20
```

✅ **When to Use Lambda Functions?**
✔ **Useful for short, simple operations.**
✔ **Often used in sorting, filtering, and mapping data.**

🔹 Built-in Higher-Order Functions (`map()`, `filter()`, `reduce()`)

Python provides functional programming features like `map()`, `filter()`, and `reduce()`.

🔹 `map()`

Applies a function to all elements in an iterable.

```
numbers = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x**2, numbers))
print(squared)  # [1, 4, 9, 16, 25]
```

- `filter()`

Filters elements based on a condition.

```
numbers = [1, 2, 3, 4, 5, 6]
evens = list(filter(lambda x: x % 2 == 0, numbers))
print(evens)  # [2, 4, 6]
```

- `reduce()` **(from** `functools` **module)**

Performs a cumulative computation (e.g., sum, product).

```
from functools import reduce

numbers = [1, 2, 3, 4]
product = reduce(lambda x, y: x * y, numbers)
print(product)  # 24
```

🔹 Recursive Functions

A function that calls itself.

```
def factorial(n):
    if n == 1:
        return 1
    return n * factorial(n - 1)


print(factorial(5))  # 120
```

✅ **Recursion is useful for solving problems like:**

- **Factorial calculation**
- **Fibonacci sequence**
- **Tree traversal**

🔹 Summary Table

Concept	Description
Function	Block of reusable code
`def`	Defines a function

`return`	**Returns a value from the function**
Positional Arguments	**Passed in order**
Default Arguments	**Uses predefined values if no value is given**
Keyword Arguments	**Passed by name**
`*args`	**Accepts multiple positional arguments**
`**kwargs`	**Accepts multiple keyword arguments**
Lambda	**Anonymous, single-line function**
`map()`	**Applies function to each element**
`filter()`	**Filters elements based on a condition**

`reduce()`	Performs cumulative computation
Recursion	Function calling itself

✅ Key Takeaways

✔ Functions help reuse code and improve readability.
✔ `*args` allows multiple positional arguments.
✔ `**kwargs` allows multiple keyword arguments.
✔ `lambda` creates small, anonymous functions.
✔ Higher-order functions (`map()`, `filter()`, `reduce()`) process data efficiently.
✔ Recursion is useful for mathematical computations.

📌 Day 5: File Handling in Python – Detailed Notes

File handling is essential for reading, writing, and managing files (text, CSV, JSON, etc.) in Python. Python provides built-in functions for working with files using the `open()` function.

🔹 1. Opening a File in Python

Python uses the `open()` function to open files. The syntax is:

file = open("filename", "mode")

Mode	Description
`'r'`	Read mode (default)
`'w'`	Write mode (creates/truncates file)
`'a'`	Append mode (adds data to file)
`'x'`	Create mode (fails if file exists)
`'rb'`	Read binary mode
`'wb'`	Write binary mode

🔹 2. Reading from a File

🔹 Reading an Entire File

```
file = open("sample.txt", "r")  # Open file in read mode
```

```
content = file.read()  # Read entire file
print(content)
file.close()  # Always close the file
```

🔹 **Reading Line by Line**

```
file = open("sample.txt", "r")
for line in file:
    print(line.strip())  # strip() removes extra spaces/newline characters
file.close()
```

🔹 **Reading with `readline()` and `readlines()`**

- `readline()` → Reads one line at a time
- `readlines()` → Reads all lines as a list

```
file = open("sample.txt", "r")
print(file.readline())  # Read first line
print(file.readlines())  # Read all lines as a list
file.close()
```

🔹 3. Writing to a File

🔹 **Overwriting a File (`'w'` mode)**

```
file = open("output.txt", "w")

file.write("Hello, this is a new file!")

file.close()
```

🔹 **Important: If the file already exists, it will be erased and replaced with new content.**

🔹 **Appending to a File (`'a'` mode)**

```
file = open("output.txt", "a")

file.write("\nAdding new content without deleting old data.")

file.close()
```

✅ **This adds new text without overwriting the existing content.**

🔹 4. Using `with open()` (Best Practice)

Using `with open()` automatically closes the file after use.

```
with open("sample.txt", "r") as file:
    content = file.read()
    print(content)  # File automatically closes after this block
```

🔹 5. Working with Binary Files

Binary files (images, PDFs, etc.) should be opened in binary mode (`'rb'`, `'wb'`).

```
with open("image.jpg", "rb") as file:
    binary_data = file.read()
```

To write binary data:

```
with open("copy.jpg", "wb") as file:
    file.write(binary_data)
```

🔹 6. Checking if a File Exists (Using `os` Module)

Before reading a file, check if it exists to avoid errors.

```
import os

if os.path.exists("sample.txt"):
    with open("sample.txt", "r") as file:
        print(file.read())
else:
    print("File not found!")
```

🔹 7. Deleting a File (Using `os` Module)

```
import os

os.remove("output.txt")  # Deletes the file
```

⚠️ **Warning: This action cannot be undone.**

🔹 8. Working with CSV Files (`csv` Module)

Python provides a built-in `csv` module for handling CSV files.

🔹 Writing to a CSV File

```
import csv


data = [
    ["Name", "Age", "City"],
    ["Alice", 30, "New York"],
    ["Bob", 25, "London"]
]


with open("data.csv", "w", newline="") as file:
    writer = csv.writer(file)
    writer.writerows(data)
```

🔹 **Reading a CSV File**

```
with open("data.csv", "r") as file:
    reader = csv.reader(file)
    for row in reader:
        print(row)
```

✅ **Output:**

```
['Name', 'Age', 'City']
['Alice', '30', 'New York']
['Bob', '25', 'London']
```

🔹 9. Working with JSON Files (`json` Module)

JSON (JavaScript Object Notation) is a common format for storing structured data.

🔹 **Writing JSON Data**

```
import json

data = {
    "name": "Alice",
    "age": 30,
```

```
    "city": "New York"
}

with open("data.json", "w") as file:
    json.dump(data, file)  # Writes JSON to file
```

🔹 **Reading JSON Data**

```
with open("data.json", "r") as file:
    data = json.load(file)  # Reads JSON file
    print(data)
```

🔹 10. Summary Table

Operation	Method
Read entire file	`read()`
Read one line	`readline()`
Read all lines as list	`readlines()`

Write (overwrite)	`write()` **with mode** `'w'`
Append to file	`write()` **with mode** `'a'`
Best practice for opening files	`with open()`
Check if file exists	`os.path.exists("file.txt")`
Delete a file	`os.remove("file.txt")`
Read CSV file	`csv.reader(file)`
Write CSV file	`csv.writer(file)`
Read JSON file	`json.load(file)`
Write JSON file	`json.dump(data, file)`

✅ Key Takeaways

✔ Always use `with open()` to handle files safely.
✔ `'r'`, `'w'`, `'a'` are file access modes.
✔ Use `csv` and `json` modules for structured data files.
✔ `os.remove()` deletes files permanently.

📌 Day 6: Exception Handling in Python – Detailed Notes

🔹 What is Exception Handling?

- An exception is an error that occurs during program execution, causing the program to stop.
- Exception handling allows us to handle errors gracefully and prevent crashes.

🔹 Common Exceptions in Python

Exception	Description
`ZeroDivisionError`	Division by zero error
`TypeError`	Operation on incompatible types

`ValueError`	Invalid value given to a function
`IndexError`	List/tuple index out of range
`KeyError`	Accessing a non-existent dictionary key
`FileNotFoundError`	Trying to open a non-existent file
`ImportError`	Importing a missing module
`AttributeError`	Invalid attribute reference

🔹 Using `try-except` to Handle Errors

- `try`: The block where code execution is attempted.
- `except`: Handles specific exceptions and prevents program crashes.

🔹 **Basic Example**

```
try:
    x = 10 / 0  # This will cause ZeroDivisionError
except ZeroDivisionError:
```

```
    print("Cannot divide by zero!")
```

✅ **Output:**

```
Cannot divide by zero!
```

🔹 Handling Multiple Exceptions

We can catch different types of errors.

```
try:
    num = int("Python")  # Causes ValueError
except ValueError:
    print("Invalid number format!")
except ZeroDivisionError:
    print("Cannot divide by zero!")
```

✅ **Output:**

```
Invalid number format!
```

🔹 Handling Multiple Exceptions in One `except` Block

Instead of multiple `except` blocks, we can handle multiple exceptions together.

```
try:
    num = int("Python")
except (ValueError, TypeError):
    print("Error: Invalid input!")
```

✅ Output:

```
Error: Invalid input!
```

🔹 Using `else` with `try-except`

- The `else` block executes only if no exception occurs.

```
try:
    x = 10 / 2  # No error
except ZeroDivisionError:
    print("Cannot divide by zero!")
else:
    print("Division successful!")  # This runs
```

✅ Output:

```
Division successful!
```

🔹 Using `finally` Block

- The `finally` block always executes, whether an exception occurs or not.
- Used for clean-up operations like closing files or releasing resources.

```
try:
    file = open("sample.txt", "r")
    content = file.read()
except FileNotFoundError:
    print("File not found!")
finally:
    print("Closing file (if opened).")
```

✅ Output:

```
File not found!
Closing file (if opened).
```

🔹 Raising Exceptions (`raise` Keyword)

- We can manually trigger an exception using `raise`.

```
x = -5
if x < 0:
    raise ValueError("Negative numbers are not allowed!")
```

✅ **Output:**

```
Traceback (most recent call last):
  raise ValueError("Negative numbers are not allowed!")
ValueError: Negative numbers are not allowed!
```

🔹 Defining Custom Exceptions

We can define our own exceptions by creating a custom class.

```
class NegativeNumberError(Exception):
  """Custom exception for negative numbers."""
  pass

def check_number(num):
  if num < 0:
    raise NegativeNumberError("Negative number detected!")

try:
  check_number(-10)
except NegativeNumberError as e:
  print(f"Error: {e}")
```

✅ **Output:**

Error: Negative number detected!

🔹 Summary Table

Concept	Description
`try`	Code block where exceptions may occur
`except`	Handles specific exceptions
`else`	Runs if no exceptions occur
`finally`	Executes regardless of exceptions
`raise`	Manually triggers an exception
Custom Exception	Defines user-defined errors

✅ Key Takeaways

✔ Use `try-except` to handle errors and prevent program crashes.
✔ Multiple exceptions can be handled separately or in one `except` block.
✔ Use `else` when code must run only if no exception occurs.
✔ Use `finally` to execute cleanup operations (closing files, releasing resources).
✔ Custom exceptions make debugging easier in larger programs.

📌 Day 7: Object-Oriented Programming (OOP) in Python – Detailed Notes

Object-Oriented Programming (OOP) is a programming paradigm that organizes code into objects. It helps in structuring complex programs, code reusability, and maintainability.

🔹 1. Key OOP Concepts

Concept	Description
Class	Blueprint for creating objects
Object	Instance of a class
Attributes	Variables inside a class
Methods	Functions inside a class

Constructor	`__init__()` method, initializes objects
Encapsulation	Restricting access to data
Inheritance	Creating a new class from an existing class
Polymorphism	Methods with the same name but different implementations

---

🔹 2. Creating Classes & Objects

A class is a blueprint, and an object is an instance of a class.

🔹 Defining a Class

```
class Car:
    def __init__(self, brand, model, year):  # Constructor
        self.brand = brand
        self.model = model
        self.year = year

    def display_info(self):  # Method
        print(f"{self.year} {self.brand} {self.model}")
```

 # Creating Objects

car1 = Car("Toyota", "Corolla", 2022)

car2 = Car("Honda", "Civic", 2021)

car1.display_info() # 2022 Toyota Corolla

car2.display_info() # 2021 Honda Civic

✅ Key Points:
✔ `self` refers to the current instance of the class.
✔ `__init__()` is the constructor, automatically called when an object is created.
✔ `display_info()` is a method, which belongs to the class.

🔹 3. Instance & Class Variables

- **Instance Variables: Specific to an object.**
- **Class Variables: Shared across all objects.**

class Employee:

company = "Google" # Class variable (same for all employees)

def __init__(self, name, salary):

self.name = name # Instance variable

self.salary = salary # Instance variable

```
emp1 = Employee("Alice", 50000)
emp2 = Employee("Bob", 60000)

print(emp1.company)  # Google
print(emp2.company)  # Google

Employee.company = "Microsoft"  # Changing class variable
print(emp1.company)  # Microsoft
print(emp2.company)  # Microsoft
```

✅ **Key Points:**
✔ `company` is a class variable (shared).
✔ `name` and `salary` are instance variables (unique to each object).

🔹 4. Encapsulation (Data Hiding)

Encapsulation restricts direct access to data using private variables (`__variable`).

```
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance  # Private variable

    def deposit(self, amount):
        self.__balance += amount
```

```
    def get_balance(self):
        return self.__balance

account = BankAccount(1000)
account.deposit(500)
print(account.get_balance())  # 1500
```

✅ **Key Points:**
✔ `__balance` is private, cannot be accessed directly (`account.__balance` will cause an error).
✔ We access private data through getter methods (`get_balance()`).

🔹 5. Inheritance (Reusing Code)

Inheritance allows a child class to inherit properties from a parent class.

🔹 Single Inheritance

```
class Animal:
    def speak(self):
        print("Animal makes a sound")

class Dog(Animal):  # Dog inherits from Animal
    def speak(self):
        print("Dog barks")
```

```
dog = Dog()
dog.speak()  # Dog barks
```

✅ **Key Points:**
✔ `Dog` class inherits from `Animal`.
✔ The `speak()` method is overridden in the `Dog` class.

🔹 **Multiple Inheritance**

A class can inherit from multiple classes.

```
class A:
    def method_A(self):
        print("Method from class A")

class B:
    def method_B(self):
        print("Method from class B")

class C(A, B):  # Multiple inheritance
    pass

obj = C()
obj.method_A()  # Method from class A
```

```
obj.method_B()  # Method from class B
```

✅ **Key Points:**
✔ **C inherits methods from both A and B.**
✔ **Useful when a class needs features from multiple parent classes.**

🔹 6. Method Overriding

A child class redefines a method from the parent class.

```
class Parent:
    def show(self):
        print("Parent class method")

class Child(Parent):
    def show(self):  # Overriding method
        print("Child class method")

obj = Child()
obj.show()  # Child class method
```

✅ **Key Points:**
✔ **If a method exists in both parent and child, the child's method overrides the parent's method.**

🔹 7. Polymorphism (Same Method, Different Implementations)

Polymorphism allows the same method name to be used in different ways.

```
class Bird:
    def fly(self):
        print("Birds can fly")


class Penguin(Bird):
    def fly(self):
        print("Penguins cannot fly")


bird = Bird()
penguin = Penguin()


bird.fly()  # Birds can fly
penguin.fly()  # Penguins cannot fly
```

✅ **Key Points:**
✔ The `fly()` method behaves differently based on the object type.
✔ Polymorphism improves code flexibility.

🔹 8. The `super()` Method

- `super()` is used to call parent class methods.

```
class Parent:
    def __init__(self, name):
        self.name = name


class Child(Parent):
    def __init__(self, name, age):
        super().__init__(name)  # Calling Parent constructor
        self.age = age


obj = Child("Alice", 25)
print(obj.name)  # Alice
print(obj.age)   # 25
```

✅ Key Points:
✔ `super().__init__(name)` calls the parent class constructor.
✔ Used to avoid code duplication in child classes.

🔹 9. Abstract Classes (`abc` Module)

Abstract classes define methods that must be implemented in child classes.

```
from abc import ABC, abstractmethod
```

```
class Animal(ABC):
    @abstractmethod
    def sound(self):
        pass  # Must be implemented in child class

class Dog(Animal):
    def sound(self):
        print("Dog barks")

dog = Dog()
dog.sound()  # Dog barks
```

✅ Key Points:
✔ Abstract classes cannot be instantiated directly.
✔ All subclasses must implement the abstract method (`sound()`).

🔹 Summary Table

Concept	Description
Class	Blueprint for creating objects
Object	Instance of a class

Constructor (`__init__()`)	**Initializes object attributes**
Encapsulation	**Restricts access using private variables (`__var`)**
Inheritance	**Child class inherits parent class properties**
Method Overriding	**Child class redefines parent method**
Polymorphism	**Same method name, different behavior**
`super()`	**Calls parent class methods**
Abstract Class	**A class with abstract methods that must be implemented**

✅ Key Takeaways

✔ OOP organizes code efficiently with classes & objects.
✔ Encapsulation restricts access to sensitive data.
✔ Inheritance promotes code reusability.
✔ Polymorphism allows flexible method implementation.
✔ Abstract classes enforce method implementation in child classes.

📌 Day 8: Python Modules & Packages – Detailed Notes

Python modules and packages help organize and reuse code effectively. Modules allow code reuse, and packages enable project structuring.

🔹 1. What is a Module?

- **A module is a Python file (`.py`) containing functions, variables, and classes.**
- **Modules help in code reusability by allowing us to import them into other programs.**

🔹 2. Creating & Importing Modules

🔹 Creating a Module (`mymodule.py`)

```
# mymodule.py (Custom module)

def greet(name):

    return f"Hello, {name}!"


PI = 3.14159
```

🔹 Importing a Module

- **Use `import module_name` to import a module.**

import mymodule

print(mymodule.greet("Alice")) # Hello, Alice!

print(mymodule.PI) # 3.14159

✅ Key Points:
✔ Use `import` to load modules.
✔ Access functions using `module_name.function()`.

🔹 Importing Specific Functions

Use `from module import function` to import only specific items.

from mymodule import greet

print(greet("Bob")) # Hello, Bob!

✅ We don't need `mymodule.greet()` since `greet()` is imported directly.

🔹 Importing with an Alias (`as`)

We can rename modules using `as`.

import mymodule as mm

print(mm.greet("Charlie")) # Hello, Charlie!

🔹 **Importing All Functions (*)**

```
from mymodule import *


print(greet("David"))  # Hello, David!
print(PI)  # 3.14159
```

⚠️ **Not recommended for large modules due to namespace conflicts.**

🔹 3. Built-in Python Modules

Python provides many built-in modules for common tasks.

🔹 **`math` Module (Mathematical Functions)**

```
import math


print(math.sqrt(16))  # 4.0
print(math.factorial(5))  # 120
print(math.pi)  # 3.141592653589793
```

🔹 **`random` Module (Generating Random Numbers)**

```
import random

print(random.randint(1, 10))  # Random number between 1 and 10
print(random.choice(["apple", "banana", "cherry"]))  # Random item
```

🔹 `datetime` Module (Working with Dates & Time)

```
from datetime import datetime

now = datetime.now()
print(now.strftime("%Y-%m-%d %H:%M:%S"))  # Current date & time
```

🔹 `os` Module (Interacting with Operating System)

```
import os

print(os.getcwd())  # Get current working directory
os.mkdir("new_folder")  # Create a new directory
os.remove("file.txt")  # Delete a file
```

🔹 `sys` Module (System-Specific Parameters & Functions)

```
import sys


print(sys.version)  # Python version

print(sys.path)  # List of directories searched for modules
```

🔹 4. What is a Package?

A package is a collection of Python modules inside a directory with a special `__init__.py` file.

🔹 Creating a Package

1. Create a folder (e.g., `mypackage`).
2. Inside `mypackage` folder, create modules (`module1.py`, `module2.py`).
3. Create an `__init__.py` file (can be empty).

```
📂 mypackage/
 ├── __init__.py
 ├── module1.py
 ├── module2.py
```

🔹 Importing from a Package

```
from mypackage import module1


print(module1.greet("Alice"))  # Assuming module1 has a greet function
```

✅ **Key Points:**
✔ **Packages organize large projects.**
✔ **`__init__.py` makes a directory a package.**

🔹 5. Installing External Packages (`pip` Tool)

- **Python uses `pip` (Python Package Installer) to install external packages.**
- **Packages are available on PyPI (Python Package Index).**

🔹 Installing a Package

pip install package_name

Example:

pip install requests

🔹 Listing Installed Packages

pip list

🔹 Uninstalling a Package

pip uninstall package_name

Example:

pip uninstall requests

🔹 **Using an Installed Package (`requests` Example)**

import requests

response = requests.get("https://api.github.com")

print(response.status_code) # 200 (Success)

🔹 6. Summary Table

Feature	Description
Module	**A `.py` file containing functions and classes**
Import Module	`import module_name`
Import Specific Function	`from module import function`

Alias Import	`import module as alias`
Package	A collection of modules in a folder
`pip install package`	Installs external packages
`pip list`	Lists installed packages

✅ Key Takeaways

✔ Modules help reuse and organize code.
✔ Packages help structure large projects.
✔ `pip` installs external libraries for additional functionality.
✔ Built-in modules like `math`, `random`, `os`, `sys`, and `datetime` are useful for various tasks.

📌 Day 9: Regular Expressions (RegEx) in Python – Detailed Notes

Regular Expressions (RegEx) are used for pattern matching in text, such as validating emails, phone numbers, extracting specific words, or searching for patterns in a string. Python provides the `re` module for working with regular expressions.

🔹 1. Importing the `re` Module

Before using regular expressions, import the `re` module.

```
import re
```

🔹 2. Basic RegEx Functions

Function	Description
`re.match(pattern, string)`	Checks if the pattern matches at the start of the string
`re.search(pattern, string)`	Searches the entire string for a match
`re.findall(pattern, string)`	Returns all occurrences of the pattern in a list
`re.finditer(pattern, string)`	Returns an iterator with match objects
`re.sub(pattern, replacement, string)`	Replaces pattern occurrences with a new string

`re.split(pattern, string)`	Splits string by the pattern

🔹 3. Using `re.match()`

- Checks if the pattern matches the beginning of the string.
- Returns a match object if successful, else `None`.

```
import re

text = "Hello World"
match = re.match(r"Hello", text)

if match:
    print("Match found:", match.group())  # Hello
else:
    print("No match")
```

✅ Key Point: `re.match()` only checks at the start of the string.

🔹 4. Using `re.search()`

- Searches anywhere in the string.

```
import re

text = "I love Python programming"
match = re.search(r"Python", text)

if match:
    print("Found:", match.group())  # Python
else:
    print("Not found")
```

🔹 5. Using `re.findall()`

- Returns all matches in a list.

```
import re

text = "apple, banana, apple, orange"
matches = re.findall(r"apple", text)

print(matches)  # ['apple', 'apple']
```

🔹 6. Using `re.finditer()`

- **Returns an iterator of match objects.**

```
import re

text = "I love Python and Python is fun"

matches = re.finditer(r"Python", text)

for match in matches:
    print("Found at:", match.start())  # Returns index positions
```

✅ **Output:**

```
Found at: 7
Found at: 18
```

🔹 7. Using `re.sub()` (Replace Text)

- **Replaces pattern occurrences in a string.**

```
import re

text = "I love Python"

new_text = re.sub(r"Python", "Java", text)
```

```
print(new_text)  # I love Java
```

🔹 8. Using `re.split()`

- Splits a string based on a pattern.

```
import re

text = "apple,banana;orange mango"
words = re.split(r"[ ,;]", text)

print(words)  # ['apple', 'banana', 'orange', 'mango']
```

✅ Explanation:
The pattern `[ ,;]` means split by comma, space, or semicolon.

🔹 9. Special Characters in RegEx

Symbol	Meaning	Example
`.`	Any character except newline	`a.b` matches `axb`, `a3b`, but not `acdb`

`^`	Start of string	`^Hello` matches `"Hello world"` but not `"Hi Hello"`
`$`	End of string	`world$` matches `"Hello world"`
`\d`	Any digit (0-9)	`\d+` matches `"123"` in `"Age: 123"`
`\D`	Any non-digit character	`\D+` matches `"Age: "` in `"Age: 123"`
`\w`	Any alphanumeric (A-Z, a-z, 0-9, _)	`\w+` matches `"hello"` in `"hello123"`
`\W`	Any non-word character	`\W+` matches `"!@#"` in `"hello!@#"`
`\s`	Any whitespace (space, tab, newline)	`\s+` matches spaces
`\S`	Any non-whitespace character	`\S+` matches `"Hello123"` in `"Hello123 "`
`\b`	Word boundary	`\bHello\b` matches `"Hello"` but not `"HelloWorld"`

🔹 10. Character Sets and Groups

🔹 Using Square Brackets `[ ]` (Character Sets)

Matches one of the specified characters.

```
import re

text = "Cat, Bat, Hat, Rat"
match = re.findall(r"[CBH]at", text)

print(match)  # ['Cat', 'Bat', 'Hat']
```

✅ **Explanation:**

- `[CBH]at` matches "Cat", "Bat", and "Hat".

🔹 Using `|` (OR Operator)

Matches either pattern.

```
import re

text = "I have an apple or banana"
match = re.findall(r"apple|banana", text)

print(match)  # ['apple', 'banana']
```

✅ **Explanation:**

- `apple|banana` matches either "apple" or "banana".

🔹 **Using Parentheses `()` (Grouping)**

Groups parts of a pattern.

```
import re


text = "John: 25, Alice: 30"

match = re.findall(r"(\w+): (\d+)", text)


print(match)  # [('John', '25'), ('Alice', '30')]
```

✅ **Explanation:**

- `(\w+)` captures the name.
- `(\d+)` captures the age.

🔹 11. Quantifiers in RegEx

Quantifiers specify how many times a character should appear.

Symbol	Meaning	Example

`*`	0 or more occurrences	`ab*c` matches `"ac"`, `"abc"`, `"abbc"`
`+`	1 or more occurrences	`ab+c` matches `"abc"`, `"abbc"`, but not `"ac"`
`?`	0 or 1 occurrence (optional)	`ab?c` matches `"ac"` or `"abc"`
`{n}`	Exactly `n` times	`a{3}` matches `"aaa"`
`{n,}`	At least `n` times	`a{2,}` matches `"aa"`, `"aaa"`, `"aaaa"`
`{n,m}`	Between `n` and `m` times	`a{2,4}` matches `"aa"`, `"aaa"`, `"aaaa"`

- **Example: Using Quantifiers**

```
import re


text = "aaaa abc aabbc"

match = re.findall(r"a{2,3}b", text)
```

```
print(match)  # ['aab', 'aab']
```

✅ **Explanation:**

- `a{2,3}b` matches `"aab"` in `"aabbc"`.

🔹 12. Validating an Email Address

```
import re

email = "user@example.com"
pattern = r"^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$"

if re.match(pattern, email):
    print("Valid email")
else:
    print("Invalid email")
```

🔹 Summary Table

Function	Description

`re.match()`	Matches pattern at the start
`re.search()`	Finds first occurrence in the string
`re.findall()`	Finds all matches
`re.sub()`	Replaces pattern with new text
`re.split()`	Splits string by pattern

✅ Key Takeaways

✔ RegEx is useful for pattern matching, validation, and text processing.
✔ Use `re.match()` for matching at the start, `re.search()` for anywhere, and `re.findall()` for all matches.
✔ Quantifiers (`*`, `+`, `{}`) control how many times a pattern appears.
✔ Use character sets (`[abc]`), groups (`()`) and alternatives (`|`) for flexible pattern matching.

📌 Day 10: Working with Databases in Python – Detailed Notes

Python provides built-in support for **SQL databases (SQLite, MySQL, PostgreSQL, MSSQL, etc.)** using various libraries like `sqlite3`, `pyodbc`, and `sqlalchemy`.

🔹 1. Database Basics

A **database** is a structured collection of data that allows for **storing, retrieving, and managing information efficiently**.

✅ **Common Databases Used in Python:**

- **SQLite** → Lightweight, file-based database (**built-in with Python**)
- **MySQL** → Popular open-source relational database
- **PostgreSQL** → Advanced open-source database
- **Microsoft SQL Server (MSSQL)** → Enterprise-grade relational database
- **MongoDB** → NoSQL document-based database

🔹 2. Connecting to SQLite (`sqlite3` – Built-in Python Library)

SQLite is a **lightweight, file-based** database that comes **pre-installed** with Python.

🔹 Connecting to SQLite Database

```
import sqlite3

# Connect to database (or create if not exists)
conn = sqlite3.connect("my_database.db")

# Create a cursor object to execute SQL commands
cursor = conn.cursor()

print("Connected to SQLite successfully!")
```

✅ If `"my_database.db"` does not exist, it will be **created automatically**.

🔹 3. Creating a Table

```
cursor.execute('''CREATE TABLE IF NOT EXISTS employees (
                    id INTEGER PRIMARY KEY,
                    name TEXT NOT NULL,
                    age INTEGER,
                    department TEXT
                )''')

conn.commit()  # Save changes
print("Table created successfully!")
```

✅ `PRIMARY KEY` ensures **each row has a unique identifier**.

🔹 4. Inserting Data

```
cursor.execute("INSERT INTO employees (name, age, department) VALUES (?, ?, ?)",
               ("Alice", 30, "IT"))

conn.commit()  # Save changes
print("Data inserted successfully!")
```

✅ **Use `?` for parameterized queries** (prevents SQL injection).

🔹 5. Fetching Data (SELECT)

```
cursor.execute("SELECT * FROM employees")
rows = cursor.fetchall()  # Fetch all rows

for row in rows:
    print(row)
```

✅ **Output Example:**

(1, 'Alice', 30, 'IT')

🔹 6. Updating Data (`UPDATE`)

```
cursor.execute("UPDATE employees SET age = ? WHERE name = ?", (31, "Alice"))
conn.commit()
print("Data updated successfully!")
```

🔹 7. Deleting Data (`DELETE`)

```
cursor.execute("DELETE FROM employees WHERE name = ?", ("Alice",))
conn.commit()
print("Data deleted successfully!")
```

🔹 8. Using `fetchone()` & `fetchmany()`

- `fetchone()` → Fetches **one row**
- `fetchmany(n)` → Fetches **n rows**

```
cursor.execute("SELECT * FROM employees")
row = cursor.fetchone()  # Fetches first row
print(row)

rows = cursor.fetchmany(2)  # Fetches first 2 rows
print(rows)
```

🔹 9. Closing the Database Connection

Always **close** the connection when done.

```
conn.close()
print("Database connection closed.")
```

🔹 10. Connecting to MySQL (`mysql-connector-python`)

To connect Python to **MySQL**, install the MySQL connector:

```
pip install mysql-connector-python
```

🔹 Connecting to MySQL

```
import mysql.connector

conn = mysql.connector.connect(
    host="localhost",
    user="root",
    password="password",
    database="test_db"
)

cursor = conn.cursor()
print("Connected to MySQL successfully!")
```

🔹 11. Connecting to MSSQL Server (`pyodbc`)

For **Microsoft SQL Server (MSSQL)**, install `pyodbc`:

```
pip install pyodbc
```

🔹 Connecting to MSSQL

```
import pyodbc

conn = pyodbc.connect(
    "DRIVER={ODBC Driver 17 for SQL Server};"
    "SERVER=your_server_name;"
    "DATABASE=your_database_name;"
    "UID=your_username;"
    "PWD=your_password"
)

cursor = conn.cursor()
```

```
print("Connected to MSSQL successfully!")
```

🔹 12. Using SQLAlchemy for Database Operations

SQLAlchemy is an **Object-Relational Mapper (ORM)** that provides a higher-level way to interact with databases.

🔹 Installing SQLAlchemy

```
pip install sqlalchemy
```

🔹 Connecting to SQLite with SQLAlchemy

```
from sqlalchemy import create_engine

engine = create_engine("sqlite:///my_database.db")
conn = engine.connect()
print("Connected using SQLAlchemy!")
```

🔹 13. Summary Table

Operation	SQLite Example
Connect to DB	`sqlite3.connect("my_database.db")`
Create Table	`CREATE TABLE employees (...)`
Insert Data	`INSERT INTO employees VALUES (...)`
Select Data	`SELECT * FROM employees`
Update Data	`UPDATE employees SET ...`
Delete Data	`DELETE FROM employees WHERE ...`

Close Connection	`conn.close()`

✅ Key Takeaways

✔ **`sqlite3` is built into Python** and great for small projects.
✔ Use **parameterized queries (`?`)** to prevent SQL injection.
✔ **MySQL & MSSQL** require **external connectors** (`mysql-connector-python`, `pyodbc`).
✔ **SQLAlchemy** provides an **ORM approach** for database management.

📌 Day 11: Multithreading & Multiprocessing in Python – Detailed Notes

Python supports **concurrent execution** using **multithreading** and **multiprocessing**. These techniques improve **performance** and **efficiency**, especially for CPU-bound and I/O-bound tasks.

🔹 1. What is Multithreading?

Multithreading allows **multiple threads** to run **within the same process**, sharing the same memory space.

✅ **Use Case:** I/O-bound tasks (file operations, network requests, database queries).
✅ **Library Used:** `threading`

🔹 2. Creating Threads in Python (`threading` Module)

We use the `threading` module to run **multiple threads** concurrently.

import threading

```
def print_numbers():
    for i in range(5):
        print(f"Number: {i}")

# Creating a thread
thread = threading.Thread(target=print_numbers)

# Starting the thread
thread.start()

# Waiting for thread to complete
thread.join()

print("Main program finished")
```

✅ **Key Points:**
✔ `Thread(target=function)` creates a new thread.
✔ `.start()` begins execution.
✔ `.join()` ensures the thread **completes before moving forward**.

🔹 3. Running Multiple Threads

```
import threading

def task(name):
    print(f"Task {name} is running")
```

```
threads = []
for i in range(3):
    t = threading.Thread(target=task, args=(i,))
    threads.append(t)
    t.start()

for t in threads:
    t.join()
```

✅ **Output Example:**

```
Task 0 is running
Task 1 is running
Task 2 is running
```

✅ **Each task runs in a separate thread.**

🔹 4. Thread Synchronization (Avoiding Race Conditions)

Threads **share memory**, which can lead to **race conditions**. We use **Locks (`threading.Lock()`)** to synchronize access to shared resources.

```
import threading

lock = threading.Lock()
counter = 0
```

```
def increment():
    global counter
    with lock:  # Lock acquired
        for _ in range(100000):
            counter += 1

t1 = threading.Thread(target=increment)
t2 = threading.Thread(target=increment)

t1.start()
t2.start()
t1.join()
t2.join()

print("Final Counter:", counter)
```

✅ **Why Use Locks?**

- Without locks, **race conditions** can cause unpredictable results.
- Using `lock.acquire()` and `lock.release()` ensures **thread safety**.

🔹 5. What is Multiprocessing?

Multiprocessing allows multiple **processes** to run in **parallel**, utilizing multiple CPU cores.

✅ **Use Case:** CPU-bound tasks (data processing, mathematical computations).
✅ **Library Used:** `multiprocessing`

🔹 6. Creating Processes in Python (`multiprocessing` Module)

Each process runs **in a separate memory space**.

```
import multiprocessing


def task(name):
    print(f"Process {name} is running")


if __name__ == "__main__":
    process = multiprocessing.Process(target=task, args=("A",))
    process.start()
    process.join()
```

✅ **Key Points:**
✔ Each process **runs independently** in memory.
✔ `.start()` begins execution.
✔ `.join()` ensures process completion before moving forward.

🔹 7. Running Multiple Processes

```
import multiprocessing


def square(num):
    print(f"Square of {num}: {num ** 2}")
```

```
if __name__ == "__main__":
    numbers = [1, 2, 3, 4, 5]
    processes = []

    for num in numbers:
        p = multiprocessing.Process(target=square, args=(num,))
        processes.append(p)
        p.start()

    for p in processes:
        p.join()
```

✅ **Each process computes independently using separate memory**.

🔹 8. Using Process Pool (`multiprocessing.Pool`)

A **process pool** allows us to execute functions **in parallel**.

```
import multiprocessing

def square(num):
    return num ** 2

if __name__ == "__main__":
    numbers = [1, 2, 3, 4, 5]
    pool = multiprocessing.Pool(processes=3)  # 3 parallel processes
```

```
results = pool.map(square, numbers)


pool.close()

pool.join()


print(results)  # [1, 4, 9, 16, 25]
```

✅ **Key Points:**
✔ `Pool(processes=n)` creates a **pool of workers**.
✔ `.map(function, iterable)` applies **function to each element in parallel**.
✔ `.close()` and `.join()` clean up resources.

🔹 9. Differences: Multithreading vs Multiprocessing

Feature	Multithreading (`threading`)	Multiprocessing (`multiprocessing`)
Execution	Concurrent (shared memory)	Parallel (separate memory)
Memory Usage	Shared	Separate for each process
Best for	I/O-bound tasks (file I/O, network requests)	CPU-bound tasks (heavy computations)
Python GIL Affected?	✅ Yes	❌ No

Example	Downloading files, web scraping	Matrix multiplication, data processing

✅ **Python's Global Interpreter Lock (GIL)** prevents **true parallelism** in **multithreading**, but **multiprocessing bypasses this issue**.

🔹 10. Summary Table

Concept	Description
`threading.Thread()`	Creates a new thread
`thread.start()`	Starts a thread
`thread.join()`	Waits for thread to finish
`threading.Lock()`	Prevents race conditions
`multiprocessing.Process()`	Creates a new process
`multiprocessing.Pool()`	Manages a pool of worker processes
`GIL (Global Interpreter Lock)`	Restricts true parallel threading in Python

✅ Key Takeaways

✔ **Multithreading** is best for **I/O-bound tasks** (network requests, file reading).
✔ **Multiprocessing** is best for **CPU-bound tasks** (data processing, computations).
✔ Use **Locks (`threading.Lock()`)** to prevent race conditions in multithreading.
✔ Use `multiprocessing.Pool()` to execute multiple processes in parallel.
✔ **Multiprocessing bypasses Python's GIL**, allowing real parallelism.

📌 Day 12: Python Decorators & Generators – Detailed Notes

🔹 1. What are Decorators in Python?

A **decorator** is a function that **modifies the behavior of another function** without changing its code.

✅ **Use Cases:**

- Logging function calls
- Measuring execution time
- Authentication checks
- Modifying return values

🔹 2. Creating a Simple Decorator

```
def decorator_function(original_function):

    def wrapper():

        print("Wrapper executed before", original_function.__name__)

        return original_function()

    return wrapper
```

```
@decorator_function  # Applying decorator
def say_hello():
    print("Hello, World!")

say_hello()
```

✅ **Output:**

```
Wrapper executed before say_hello
Hello, World!
```

✅ **Explanation:**

- `@decorator_function` **wraps** `say_hello()` inside `wrapper()`.
- The wrapper **executes extra code before calling the original function**.

🔹 3. Using `*args` and `**kwargs` in Decorators

To decorate **functions with arguments**, use `*args` and `**kwargs`.

```
def decorator_function(original_function):
    def wrapper(*args, **kwargs):
        print(f"Executing {original_function.__name__} with arguments {args}")
        return original_function(*args, **kwargs)
    return wrapper

@decorator_function
def add(a, b):
```

```
    return a + b


print(add(5, 3))
```

✅ **Output:**

```
Executing add with arguments (5, 3)
8
```

✅ **Key Points:**
✔ `*args` allows any number of positional arguments.
✔ `**kwargs` allows keyword arguments.

🔹 4. Decorating Multiple Functions

```
def logger(func):
    def wrapper(*args, **kwargs):
        print(f"Calling {func.__name__} with arguments {args}, {kwargs}")
        return func(*args, **kwargs)
    return wrapper


@logger
def multiply(a, b):
    return a * b


@logger
def greet(name):
```

```
    print(f"Hello, {name}!")


print(multiply(4, 5))

greet("Alice")
```

✅ **Output:**

```
Calling multiply with arguments (4, 5), {}

20

Calling greet with arguments ('Alice',), {}

Hello, Alice!
```

🔹 5. Measuring Execution Time with `time` Module

We can use decorators to measure **how long a function takes to execute**.

```
import time


def timer(func):
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        print(f"{func.__name__} executed in {end_time - start_time:.4f} seconds")
        return result
    return wrapper
```

```
@timer
def slow_function():
    time.sleep(2)
    print("Function completed")

slow_function()
```

✅ **Output:**

```
Function completed
slow_function executed in 2.0001 seconds
```

🔹 6. What are Generators in Python?

A **generator** is a special function that **produces values lazily (one at a time) instead of returning them all at once**.

✅ **Use Cases:**

- Efficient memory usage for large datasets
- Infinite sequences (e.g., Fibonacci series)
- Reading large files line by line

🔹 7. Creating a Simple Generator (`yield` Keyword)

Instead of `return`, we use `yield` in a function to create a generator.

```
def simple_generator():
    yield 1
```

```
    yield 2
    yield 3

gen = simple_generator()
print(next(gen))  # 1
print(next(gen))  # 2
print(next(gen))  # 3
```

✅ **Key Points:**
✔ `yield` suspends function execution and **remembers its state**.
✔ Calling `next(generator)` **resumes execution from the last yield**.

🔹 8. Using Generators in a Loop

```
def count_up_to(n):
    count = 1
    while count <= n:
        yield count
        count += 1

for num in count_up_to(5):
    print(num)
```

✅ **Output:**

```
1
2
```

3

4

5

✅ **Why Use Generators?**

- Saves **memory** (does not store all values at once).
- More **efficient** for large datasets.

🔹 9. Generator Expression (Like List Comprehension)

A **generator expression** is similar to a list comprehension but uses **()`instead of[]`**.

```
gen_exp = (x**2 for x in range(5))


print(next(gen_exp))  # 0

print(next(gen_exp))  # 1

print(list(gen_exp))  # Remaining values: [4, 9, 16]
```

✅ **Why Use Generator Expressions?**

- More **memory-efficient** than list comprehensions.
- Works well for **large datasets**.

🔹 10. Infinite Sequence Generator

```
def infinite_numbers():

    num = 1

    while True:
```

```
        yield num
        num += 1

gen = infinite_numbers()
print(next(gen))  # 1
print(next(gen))  # 2
print(next(gen))  # 3
```

✅ **Useful for streaming real-time data processing**.

🔹 11. Combining Decorators and Generators

We can **decorate a generator function**.

```
def log_generator(func):
    def wrapper(*args, **kwargs):
        print(f"Generator {func.__name__} started")
        yield from func(*args, **kwargs)  # Pass values from generator
        print(f"Generator {func.__name__} ended")
    return wrapper

@log_generator
def number_sequence(n):
    for i in range(1, n+1):
        yield i
```

```
for num in number_sequence(3):
    print(num)
```

✅ **Output:**

```
Generator number_sequence started
1
2
3
Generator number_sequence ended
```

✅ **Explanation:**

- `yield from` passes values **from one generator to another**.
- Logs when the generator **starts and ends**.

🔹 12. Summary Table

Concept	Description
Decorators	Modify a function's behavior without changing its code
`@decorator_function`	Used to apply a decorator to a function
`*args, **kwargs` in decorators	Allows decorating functions with different arguments

Generators	Functions that use `yield` to produce values lazily
`yield`	Returns a value and **pauses execution**
`next(generator)`	Retrieves the next value from a generator
`yield from`	Passes values from another generator
Generator Expression	`(x**2 for x in range(5))`, more memory-efficient than list comprehension

---

✅ Key Takeaways

✔ **Decorators** modify functions **without altering their code**.
✔ Use decorators for **logging, performance measurement, authentication, etc.**
✔ **Generators** efficiently produce values **one at a time**, saving memory.
✔ `yield from` allows **delegating** a generator's output to another generator.
✔ **Generator expressions** (`(x for x in range(5))`) are **memory-efficient** alternatives to list comprehensions.

📌 Day 13: Working with APIs & Web Scraping in Python – Detailed Notes

APIs (**Application Programming Interfaces**) allow communication between different software systems, and **Web Scraping** helps extract data from websites.

🔹 1. Working with APIs in Python (`requests` Module)

Python provides the `requests` module to interact with **REST APIs**.

✅ **Common Use Cases:**

- Fetching data from web services
- Sending data to a server
- Working with authentication (API keys, OAuth)

🔹 Install `requests` Module

pip install requests

🔹 2. Making a GET Request (Fetching Data)

The **GET** method is used to **retrieve data** from an API.

```
import requests


response = requests.get("https://jsonplaceholder.typicode.com/posts/1")


if response.status_code == 200:
    data = response.json()  # Convert response to JSON
    print(data)
else:
    print("Failed to fetch data:", response.status_code)
```

✅ **Output (Example API Response):**

```
{
    "userId": 1,
    "id": 1,
    "title": "Sample Post",
    "body": "This is the content of the post."
}
```

🔹 3. Making a POST Request (Sending Data)

The **POST** method is used to **send data** to an API.

```
import requests

url = "https://jsonplaceholder.typicode.com/posts"
data = {
    "title": "New Post",
    "body": "This is a test post.",
    "userId": 1
}

response = requests.post(url, json=data)

if response.status_code == 201:
    print("Data sent successfully:", response.json())
else:
```

```
    print("Error:", response.status_code)
```

✅ **Response Example:**

```
{
    "id": 101,
    "title": "New Post",
    "body": "This is a test post.",
    "userId": 1
}
```

🔹 4. Handling API Headers

Some APIs require **headers** (e.g., authentication, JSON format).

```
headers = {"Authorization": "Bearer YOUR_API_KEY", "Content-Type": "application/json"}


response = requests.get("https://api.example.com/data", headers=headers)
print(response.json())
```

🔹 5. Handling Query Parameters

Query parameters help **filter** API responses.

```
params = {"userId": 1}
response = requests.get("https://jsonplaceholder.typicode.com/posts", params=params)
```

```
print(response.json())  # Returns posts from userId = 1
```

🔹 6. Introduction to Web Scraping (`BeautifulSoup`)

Web Scraping extracts **data from websites** using **HTML parsing**.

✅ **Use Cases:**

- Extracting data from web pages
- Automating web-based tasks
- Analyzing website content

🔹 7. Installing `BeautifulSoup` and `requests`

```
pip install beautifulsoup4 requests
```

🔹 8. Fetching and Parsing Web Content

```
import requests

from bs4 import BeautifulSoup


url = "https://example.com"

response = requests.get(url)


if response.status_code == 200:
```

```
    soup = BeautifulSoup(response.text, "html.parser")
    print(soup.prettify())  # Prints formatted HTML
else:
    print("Failed to retrieve page")
```

✅ **Key Points:**
✔ `BeautifulSoup(response.text, "html.parser")` parses HTML content.
✔ `soup.prettify()` formats the HTML.

🔹 9. Extracting Specific Elements

```
title = soup.find("title").text
print("Page Title:", title)

headings = soup.find_all("h2")  # Extract all `<h2>` elements
for h in headings:
    print(h.text)
```

✅ **Explanation:**
✔ `soup.find("title")` → Extracts the `<title>` tag.
✔ `soup.find_all("h2")` → Finds **all** `<h2>` elements.

🔹 10. Extracting Links from a Web Page

```
links = soup.find_all("a")  # Finds all anchor `<a>` tags

for link in links:
```

```
    print(link.get("href"))  # Extracts href attribute (URL)
```

🔹 11. Extracting Table Data

```
table = soup.find("table")

rows = table.find_all("tr")


for row in rows:

    columns = row.find_all("td")

    data = [col.text.strip() for col in columns]

    print(data)
```

✅ **Explanation:**
✔ `find("table")` selects the **first** `<table>`.
✔ `find_all("tr")` gets **all rows**.
✔ `find_all("td")` extracts **column data**.

🔹 12. Web Scraping with Pagination

Many websites have **multiple pages** of data. We can scrape multiple pages using a loop.

```
for page in range(1, 4):  # Scrape first 3 pages

    url = f"https://example.com/page/{page}"

    response = requests.get(url)

    soup = BeautifulSoup(response.text, "html.parser")
```

```
    titles = soup.find_all("h2")
    for title in titles:
        print(title.text)
```

✅ **Explanation:**
✔ `f"https://example.com/page/{page}"` dynamically changes URLs.
✔ Extracts **headings** from **multiple pages**.

🔹 13. Handling JavaScript-Rendered Content (`Selenium`)

Some websites **load data dynamically** using JavaScript. In such cases, `requests` and `BeautifulSoup` **won't work**. Instead, we use `selenium`.

🔹 Install Selenium

```
pip install selenium
```

🔹 Using Selenium to Scrape JavaScript Content

```
from selenium import webdriver

driver = webdriver.Chrome()  # Open Chrome browser
driver.get("https://example.com")  # Load page

print(driver.page_source)  # Get rendered HTML content
driver.quit()  # Close browser
```

✅ **Key Points:**
✔ **Selenium controls a browser** and retrieves dynamically loaded content.
✔ Use it when `requests` and `BeautifulSoup` **fail to extract content**.

🔹 14. Summary Table

Concept	Description
`requests.get(url)`	Fetches data from an API
`response.json()`	Converts API response to JSON
`requests.post(url, json=data)`	Sends data using POST request
`BeautifulSoup(html, "html.parser")`	Parses HTML content
`soup.find("tag")`	Extracts **first** occurrence of a tag
`soup.find_all("tag")`	Extracts **all** occurrences of a tag
`soup.find("a")["href"]`	Extracts **URL from a link**
`selenium.webdriver.Chrome()`	Opens a Chrome browser

✅ Key Takeaways

✔ **APIs** allow communication between applications; `requests` is used to **fetch and send data**.
✔ **Web Scraping** extracts information from **web pages**.
✔ `BeautifulSoup` parses **HTML**, making data extraction **easy**.
✔ `selenium` is used to scrape **JavaScript-rendered websites**.

📌 Day 14: Unit Testing in Python – Detailed Notes

Unit testing ensures that **individual components** of a program **work as expected**. Python provides the built-in `unittest` module for testing.

✅ **Why Unit Testing?**
✔ **Catches bugs early** before deployment
✔ **Ensures reliability** of functions and modules
✔ **Automates testing** for continuous integration (CI/CD)
✔ **Reduces debugging time**

🔹 1. Introduction to `unittest` Module

- The `unittest` module provides a framework for writing test cases.
- Tests are written in **separate test files** and executed using `python -m unittest`.

🔹 2. Writing a Basic Test Case

1️⃣ **Create a Python file (`calculator.py`)**

```
def add(a, b):
    return a + b
```

```
def subtract(a, b):
    return a - b
```

2️⃣ **Create a Test File (`test_calculator.py`)**

```
import unittest
import calculator  # Import the module to test

class TestCalculator(unittest.TestCase):

    def test_add(self):
        self.assertEqual(calculator.add(2, 3), 5)  # Test case 1
        self.assertEqual(calculator.add(-1, 1), 0)  # Test case 2

    def test_subtract(self):
        self.assertEqual(calculator.subtract(5, 3), 2)
        self.assertEqual(calculator.subtract(10, 5), 5)

if __name__ == "__main__":
    unittest.main()
```

✅ **Run the test in the terminal:**

```
python -m unittest test_calculator.py
```

✅ **Output (if tests pass)**

..

Ran 2 tests in 0.001s

OK

✅ **Key Points:**
✔ `unittest.TestCase` is the base class for tests.
✔ `assertEqual(x, y)` checks if `x == y`.
✔ Running `unittest.main()` executes all test cases.

🔹 3. Common Assertions in `unittest`

Assertion Method	Description
`assertEqual(a, b)`	Check if `a == b`
`assertNotEqual(a, b)`	Check if `a != b`
`assertTrue(x)`	Check if `x` is `True`
`assertFalse(x)`	Check if `x` is `False`
`assertIn(x, y)`	Check if `x` is in `y`

`assertIsNone(x)`	Check if `x` is `None`

🔹 4. Using `setUp()` and `tearDown()` for Setup & Cleanup

- `setUp()` runs **before** each test case.
- `tearDown()` runs **after** each test case (useful for closing database connections).

```
import unittest


class TestExample(unittest.TestCase):

    def setUp(self):

        print("Setting up...")


    def test_case_1(self):

        print("Running test case 1")

        self.assertEqual(2 + 2, 4)


    def tearDown(self):

        print("Tearing down...")


if __name__ == "__main__":

    unittest.main()
```

✅ **Output:**

Setting up...

Running test case 1

Tearing down...

.

Ran 1 test in 0.000s

OK

✅ **Key Points:**
✔ `setUp()` prepares test cases (e.g., open DB connection).
✔ `tearDown()` cleans up after tests (e.g., close DB connection).

🔹 5. Mocking in Unit Tests (`unittest.mock`)

✅ **Why Mocking?**
✔ **Simulates external dependencies** (e.g., API calls, databases)
✔ **Speeds up tests** by avoiding real network requests
✔ **Prevents altering production data**

🔹 Example: Mocking an API Call

```
import unittest

from unittest.mock import patch

import requests


def get_weather(city):

    url = f"https://api.weather.com/{city}"

    response = requests.get(url)

    return response.json()
```

```
class TestWeatherAPI(unittest.TestCase):
    @patch("requests.get")  # Mock API call
    def test_get_weather(self, mock_get):
        mock_get.return_value.json.return_value = {"temperature": 25}

        result = get_weather("New York")
        self.assertEqual(result["temperature"], 25)

if __name__ == "__main__":
    unittest.main()
```

✅ **Key Points:**
✔ `@patch("requests.get")` replaces `requests.get()` with a **mocked response**.
✔ `mock_get.return_value.json.return_value` **sets the expected response**.

🔹 6. Running All Tests in a Directory

To run **all test files** in a folder:

```
python -m unittest discover
```

✅ This finds **all test files (`test_*.py`)** in the directory and runs them.

🔹 7. Testing Exceptions (`assertRaises`)

Use `assertRaises()` to **check if an exception is raised**.

```
def divide(a, b):
    if b == 0:
        raise ValueError("Cannot divide by zero!")
    return a / b

class TestDivide(unittest.TestCase):
    def test_divide_by_zero(self):
        with self.assertRaises(ValueError):
            divide(10, 0)

if __name__ == "__main__":
    unittest.main()
```

✅ **Key Points:**
✔ `assertRaises(ValueError)` expects `divide(10, 0)` to raise a `ValueError`.

🔹 8. Automating Tests with `pytest` (Alternative to `unittest`)

- `pytest` is a popular testing framework with **less boilerplate code**.

✅ **Install `pytest`:**

```
pip install pytest
```

🔹 Example Test in `pytest`

```
# test_example.py
```

```
import pytest

import calculator


def test_add():

    assert calculator.add(2, 3) == 5

    assert calculator.add(-1, 1) == 0


def test_subtract():

    assert calculator.subtract(5, 3) == 2
```

✅ **Run tests with `pytest`:**

```
pytest
```

✅ **Key Points:**
✔ `pytest` requires **no class structure** (`unittest.TestCase` not needed).
✔ `assert` is used instead of `self.assertEqual()`.

🔹 9. Test Coverage Report (`coverage.py`)

`coverage.py` measures **how much of your code is tested**.

✅ **Install `coverage`:**

```
pip install coverage
```

✅ **Run tests and check coverage:**

```
coverage run -m unittest discover
```

coverage report

✅ **Key Points:**
✔ Shows which **lines of code are untested**.
✔ Helps **improve test coverage**.

🔹 10. Summary Table

Concept	Description
`unittest`	Built-in Python module for unit testing
`assertEqual(a, b)`	Checks if `a == b`
`assertRaises(Exception)`	Ensures exception is raised
`setUp()` / `tearDown()`	Runs before / after each test case
`@patch("requests.get")`	Mocks an API request
`pytest`	Alternative to `unittest`, uses `assert`
`coverage.py`	Measures test coverage

✅ Key Takeaways

✔ **Unit testing ensures reliability** and catches bugs early.
✔ Use `unittest` for structured testing, or `pytest` for simplicity.
✔ Mock **external dependencies** to isolate tests.
✔ `coverage.py` helps identify **untested code**.
✔ Running `unittest discover` executes **all test files automatically**.

📌 Day 15: Python Best Practices & Code Optimization – Detailed Notes

On the final day, we will focus on **writing clean, efficient, and optimized Python code** by following **best practices**.

🔹 1. Writing Clean & Readable Code (PEP 8 Guidelines)

PEP 8 is the **official style guide** for writing Python code.

✅ **Key PEP 8 Rules:**
✔ **Use 4 spaces per indentation level (not tabs).**
✔ **Limit lines to 79 characters** (for readability).
✔ **Use meaningful variable names.**
✔ **Use blank lines to separate code blocks.**
✔ **Follow consistent naming conventions.**

✅ **Example: Good vs Bad Code**

🚫 **Bad Code:**

```
def calc(a,b):return a+b  # Single-line, unclear function
```

✅ **Good Code:**

```
def calculate_sum(a, b):
    """Returns the sum of two numbers."""
    return a + b
```

🔹 **Why?**
✔ Uses **descriptive function name** (`calculate_sum` instead of `calc`).
✔ Uses **docstring** to describe function.
✔ Proper **indentation & spacing**.

🔹 2. Using List Comprehensions (Faster & Cleaner)

List comprehensions make **loops shorter and more efficient**.

🚫 **Using a loop:**

```
numbers = [1, 2, 3, 4, 5]
squared = []
for num in numbers:
    squared.append(num ** 2)
```

✅ **Using list comprehension:**

```
squared = [num ** 2 for num in numbers]
```

🔹 3. Using Generators Instead of Lists (Memory Optimization)

Generators **save memory** by yielding values **one at a time**.

🚫 **Using a list (Consumes more memory):**

```
numbers = [x ** 2 for x in range(1000000)]  # Stores all values in memory
```

✅ **Using a generator (Memory-efficient):**

```
numbers = (x ** 2 for x in range(1000000))  # Uses lazy evaluation
```

🔹 4. Using `enumerate()` Instead of `range(len())`

🚫 **Using `range(len())`:**

```
fruits = ["apple", "banana", "cherry"]
for i in range(len(fruits)):
    print(i, fruits[i])
```

✅ **Using `enumerate()`:**

```
for index, fruit in enumerate(fruits):
    print(index, fruit)
```

🔹 5. Using `zip()` to Iterate Multiple Lists

```
names = ["Alice", "Bob", "Charlie"]
ages = [25, 30, 35]

for name, age in zip(names, ages):
```

```
    print(f"{name} is {age} years old.")
```

✅ **Output:**

Alice is 25 years old.

Bob is 30 years old.

Charlie is 35 years old.

🔹 6. Using `set()` to Remove Duplicates

```
numbers = [1, 2, 2, 3, 4, 4, 5]

unique_numbers = list(set(numbers))

print(unique_numbers)  # [1, 2, 3, 4, 5]
```

🔹 7. Using `defaultdict` Instead of Checking Keys

The `defaultdict` in `collections` automatically assigns a default value.

🚫 **Without `defaultdict` (Checking if key exists):**

```
word_count = {}

words = ["apple", "banana", "apple"]


for word in words:
    if word in word_count:
        word_count[word] += 1
```

```
    else:
        word_count[word] = 1
```

✅ **Using `defaultdict`:**

```
from collections import defaultdict

word_count = defaultdict(int)
words = ["apple", "banana", "apple"]

for word in words:
    word_count[word] += 1
```

🔹 8. Using `Counter` for Counting Elements

The `Counter` module **counts occurrences** in a list.

```
from collections import Counter

words = ["apple", "banana", "apple", "orange", "banana"]
word_count = Counter(words)

print(word_count)  # Counter({'apple': 2, 'banana': 2, 'orange': 1})
```

🔹 9. Using `try-except-else-finally` Properly

🚫 **Bad Error Handling:**

```
try:
    result = 10 / 0
except:
    print("Something went wrong")
```

❌ **Catches all exceptions (not specific).**

✅ **Good Error Handling:**

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Cannot divide by zero!")
else:
    print("No errors occurred.")
finally:
    print("Execution completed.")
```

✅ **Output:**

```
Cannot divide by zero!
Execution completed.
```

🔹 10. Avoiding Global Variables

🚫 **Bad Practice (Modifying a global variable inside a function):**

```
x = 10

def update():
    global x
    x += 5  # Modifying global variable
```

✅ **Better Approach (Pass variable as parameter):**

```
def update(x):
    return x + 5

x = update(10)
```

🔹 11. Using `timeit` for Performance Testing

The `timeit` module **measures execution time**.

```
import timeit

execution_time = timeit.timeit("sum(range(1000))", number=1000)
print(f"Execution Time: {execution_time:.5f} seconds")
```

🔹 12. Using `functools.lru_cache` for Caching

The `lru_cache` decorator **stores function results**, speeding up repeated calls.

```
from functools import lru_cache


@lru_cache(maxsize=100)
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)


print(fibonacci(10))  # Faster execution due to caching
```

🔹 13. Using `with open()` for File Handling

🚫 **Without `with open()` (Forgetting to close file):**

```
file = open("test.txt", "r")
content = file.read()
file.close()
```

✅ **Using `with open()` (Better approach):**

```
with open("test.txt", "r") as file:
    content = file.read()
```

✔ **Automatically closes file** after execution.

🔹 14. Avoiding Unnecessary List Copies (`copy()` vs `=`)

🚫 **Bad Practice (Creates a reference, not a copy):**

```
a = [1, 2, 3]

b = a  # Modifying 'b' also affects 'a'
```

✅ **Better (Use `.copy()` to create a separate copy):**

```
b = a.copy()
```

🔹 15. Summary Table

Concept	Best Practice
PEP 8	Follow style guide for readability
List Comprehensions	Use instead of `for` loops for efficiency
Generators	Use `yield` to optimize memory
enumerate()	Use instead of `range(len())`
zip()	Iterate multiple lists together
set()	Remove duplicates efficiently

defaultdict	Handle missing dictionary keys gracefully
Counter	Count occurrences in a list efficiently
try-except	Use specific exceptions for better debugging
timeit	Measure function execution time
lru_cache	Cache function results for faster execution
with open()	Use for better file handling

✅ Key Takeaways

✔ Follow **PEP 8** for clean, readable code.
✔ Use **list comprehensions** for cleaner loops.
✔ **Generators** are better than storing large lists in memory.
✔ Use `enumerate()`, `zip()`, and `defaultdict` for better efficiency.
✔ Apply **caching (`lru_cache`)** for faster repeated function calls.
✔ Use `timeit` to measure performance.

🎉 Congratulations! You've Completed the 15-Day Python Course! 🎉

Let me know if you need **additional topics, projects, or advanced exercises**! 🚀

📌 Essential Python Functions for Data Engineering Projects

In **Data Engineering**, Python is used for **data ingestion, transformation, cleaning, validation, storage, and processing**. Below is a categorized list of **key Python functions** along with their **uses in Data Engineering**.

🔹 1. File Handling Functions

Used for **reading, writing, and managing files** (CSV, JSON, Parquet, etc.).

Function	Usage
`open(filename, mode)`	Opens a file in **read/write** mode
`read()`	Reads **entire file** as a string
`readline()`	Reads **one line** at a time
`readlines()`	Reads **all lines** as a list
`write(data)`	Writes data to a file
`writelines(lines)`	Writes a list of lines to a file

`with open()`	Automatically closes file after operation

✅ **Example:** Read a file and process data

```
with open("data.txt", "r") as file:
    data = file.readlines()
```

🔹 2. Data Processing & Transformation Functions

Used for **handling and transforming data**.

Function	Usage
`map(function, iterable)`	Applies a function to all elements
`filter(function, iterable)`	Filters elements based on a condition
`reduce(function, iterable)`	Performs cumulative computation
`zip(iterables)`	Merges multiple iterables
`sorted(iterable, key, reverse)`	Sorts data

`enumerate(iterable)`	Iterates with an index

✅ **Example:** Using `map()` and `filter()`

```
from functools import reduce

numbers = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x ** 2, numbers))  # Apply function
evens = list(filter(lambda x: x % 2 == 0, numbers))  # Filter even numbers
sum_all = reduce(lambda x, y: x + y, numbers)  # Sum all numbers
```

🔹 3. Pandas Functions (Data Manipulation)

Used for **handling structured data (CSV, JSON, Databases, etc.).**

Function	Usage
`pd.read_csv("file.csv")`	Load CSV file
`pd.read_json("file.json")`	Load JSON file
`df.to_csv("file.csv", index=False)`	Save DataFrame as CSV

`df.to_json("file.json")`	Save DataFrame as JSON
`df.head(n)`	View first `n` rows
`df.info()`	Summary of dataset
`df.describe()`	Summary statistics
`df.fillna(value)`	Fill missing values
`df.dropna()`	Remove missing values
`df.groupby(column).agg()`	Aggregate data
`df.merge(df2, on="column")`	Merge two DataFrames

✅ **Example:**

```
import pandas as pd


df = pd.read_csv("data.csv")

df["new_column"] = df["existing_column"] * 2

df.dropna(inplace=True)  # Remove missing values

df.to_csv("cleaned_data.csv", index=False)
```

🔹 4. NumPy Functions (Efficient Data Processing)

Used for **fast numerical computations and matrix operations**.

Function	Usage
`np.array([1, 2, 3])`	Create NumPy array
`np.zeros((3,3))`	Create zero matrix
`np.ones((3,3))`	Create ones matrix
`np.mean(arr)`	Compute mean
`np.median(arr)`	Compute median
`np.std(arr)`	Compute standard deviation
`np.dot(A, B)`	Matrix multiplication
`np.concatenate([A, B])`	Merge arrays

✅ **Example:**

```
import numpy as np

data = np.array([1, 2, 3, 4, 5])
mean_value = np.mean(data)
```

🔹 5. Database & SQL Functions

Used for **storing and retrieving data from databases**.

Function	Usage
`sqlite3.connect("db.sqlite")`	Connect to SQLite
`cursor.execute("SQL QUERY")`	Execute SQL query
`cursor.fetchall()`	Fetch all rows
`cursor.fetchone()`	Fetch one row
`conn.commit()`	Save changes
`conn.close()`	Close connection

✅ **Example: Fetch data from SQL database**

```
import sqlite3


conn = sqlite3.connect("data.db")
cursor = conn.cursor()
cursor.execute("SELECT * FROM employees")
rows = cursor.fetchall()
conn.close()
```

🔹 6. API & Web Scraping Functions

Used for **fetching data from web APIs and websites**.

Function	Usage
`requests.get(url)`	Fetch data from API
`requests.post(url, json=data)`	Send data to API
`soup.find("tag")`	Extract HTML tag
`soup.find_all("tag")`	Extract multiple elements

✅ **Example: Fetch data from API**

```
import requests
```

response = requests.get("https://jsonplaceholder.typicode.com/posts/1")

data = response.json()

🔹 7. Cloud Storage (Azure, AWS, GCP)

Used for **storing and retrieving data from cloud storage**.

Function	Usage
`blob_client.upload_blob(data)`	Upload file to Azure Blob Storage
`s3_client.upload_file(filename, bucket, key)`	Upload file to AWS S3
`gcs_client.upload_blob(bucket, filename)`	Upload file to Google Cloud

✅ **Example: Uploading to Azure Blob Storage**

from azure.storage.blob import BlobServiceClient

blob_service = BlobServiceClient.from_connection_string("CONNECTION_STRING")

blob_client = blob_service.get_blob_client(container="mycontainer", blob="data.csv")

with open("data.csv", "rb") as data:

    blob_client.upload_blob(data)

🔹 8. Logging & Error Handling

Used for **debugging and tracking errors**.

Function	Usage
`logging.debug(msg)`	Log debug messages
`logging.info(msg)`	Log info messages
`logging.warning(msg)`	Log warnings
`logging.error(msg)`	Log errors
`logging.exception(msg)`	Log exceptions

✅ **Example: Logging Errors**

import logging

logging.basicConfig(filename="app.log", level=logging.INFO)

```
try:
    x = 10 / 0
except ZeroDivisionError:
    logging.exception("Attempted division by zero")
```

🔹 9. Job Scheduling & Workflow Automation

Used for **scheduling and automating data pipelines**.

Function	Usage
`schedule.every(n).minutes.do(func)`	Schedule function execution
`dag = DAG()`	Define an Airflow DAG
`PythonOperator(task_id, python_callable=func)`	Run Python in Airflow
`spark.read.csv("file.csv")`	Load data in Spark

✅ **Example: Scheduling a function every 5 minutes**

```
import schedule
import time

def job():
```

```
    print("Running scheduled job...")

schedule.every(5).minutes.do(job)

while True:
    schedule.run_pending()
    time.sleep(1)
```

✅ Summary Table

Category	Key Functions
File Handling	`open()`, `read()`, `write()`, `with open()`
Data Processing	`map()`, `filter()`, `reduce()`, `zip()`
Pandas	`read_csv()`, `fillna()`, `groupby()`, `merge()`
NumPy	`array()`, `mean()`, `dot()`, `concatenate()`

Databases	`connect()`, `execute()`, `fetchall()`, `commit()`
APIs & Web Scraping	`requests.get()`, `find()`, `find_all()`
Cloud Storage	`upload_blob()`, `upload_file()`
Logging	`logging.info()`, `logging.error()`
Job Scheduling	`schedule.every()`, `DAG()`, `PythonOperator()`

✅ Key Takeaways

✔ **Python provides powerful functions** for Data Engineering tasks.
✔ Use **Pandas & NumPy** for data processing.
✔ Use **APIs & Web Scraping** for external data sources.
✔ Use **Logging & Error Handling** to debug pipelines.
✔ Automate workflows with **Airflow & Scheduling**.

Let me know if you need **more details or examples** on any topic! 🚀