Python Fundamentals

Introduction to Python

1) Introduction to Python and its Features (simple, high-level, interpreted language).

- Python is a simple, high-level, interpreted programming language.
- Developed by Guido van Rossum in 1991.
- Known for its easy syntax, making it beginner-friendly.
- Provides powerful libraries that make it suitable for professionals.

Key Features of Python:

- Easy to Learn – English-like syntax.
- High-Level – No need to manage hardware or memory.
- Interpreted – Executes code line by line.
- Cross-Platform – Works on Windows, Mac, Linux.
- Object-Oriented – Supports classes and objects.
- Rich Libraries – NumPy, Pandas, Django, TensorFlow.
- Open Source – Free to use and share.

2) History and evolution of Python.

History of Python

- Created by Guido van Rossum in the late 1980s at CWI, Netherlands.

- First released in 1991 as Python 0.9.0 with basic features.

- Named after the comedy show "Monty Python's Flying Circus", not the snake.

Evolution of Python

1. Python 1.0 (1991): Basic data types, functions, modules, exception handling.

2. Python 2.x (2000): Added list comprehensions, Unicode, garbage collection. Last version 2.7 ended in 2020.

3. Python 3.x (2008–present): Modern version with Unicode by default, print() function, better syntax. Currently the most used.

3) Advantages of using Python over other programming languages.

- Advantages of Using Python

1. Easy to Learn – Simple, English-like syntax.
2. Cross-Platform – Works on Windows, Mac, Linux.
3. Rich Libraries – NumPy, Pandas, Django, TensorFlow etc.
4. Fast Development – Fewer lines of code than Java/C++.
5. Interpreted – Runs line by line, easy debugging.
6. Versatile – Used in AI, Web, Data Science, Automation.
7. Open Source – Free to use and modify.
8. Large Community – Strong support and resources.

4) Installing Python and setting up the development environment (Anaconda, PyCharm, or VS Code).

1) Install Python (Core Software)

1. Go to the official website → https://www.python.org
2. Click on Downloads and select the latest version for your OS (Windows, macOS, Linux).
3. During installation, make sure to check "Add Python to PATH".
4. Verify installation using Command Prompt / Terminal:
5. python --version

2) Anaconda (Best for Data Science & ML)

Anaconda is a Python distribution that comes with most data science, AI, and ML libraries pre-installed (NumPy, Pandas, Matplotlib, Jupyter Notebook, etc.).

Steps:

1. Download from → https://www.anaconda.com
2. Install the Individual Edition (free).
3. Open Anaconda Navigator.
4. From there, launch Jupyter Notebook or Spyder IDE for coding.

3) PyCharm (Professional IDE for Python)

PyCharm is a powerful IDE developed by JetBrains, widely used in professional development.

Steps:

1. Download from → https://www.jetbrains.com/pycharm/
2. Choose Community Edition (free) or Professional Edition (paid).
3. Install and configure the Python interpreter (path of your installed Python).

4. Create a new project and start coding.

4) VS Code (Lightweight & Flexible Editor)

Visual Studio Code is a lightweight editor by Microsoft, great for beginners and professionals.

Steps:

1. Download from → https://code.visualstudio.com/
2. Install VS Code and open it.
3. Go to Extensions (left sidebar) → Search and install "Python" extension.
4. Select Python interpreter (Ctrl+Shift+P → Python: Select Interpreter).
5. Create a .py file and run your code.

5) Writing and executing your first Python program.

- Python run it in Terminal, VS Code, IDLE
- Run It in Different Environments

1.Terminal / Command Prompt

python hello.py

Output:

Hello, World!

2.Using VS Code

1. Open VS Code.
2. Create a new file → Save it as hello.py.
3. Write the code:
4. print("Hello, World!")
5. Click Run Python File
   Output:

   Hello, World!

3.IDLE (comes with Python)

1. Open IDLE (installed automatically with Python).
2. Go to File → New File.
3. Write the code: print("Hello, World!")
4. Save as hello.py.
5. Press F5 (Run).
   Output will appear in the Python Shell window:

   Hello, World!

2.Programming Style Theory

1) Understanding Python's PEP 8 guidelines.

What is PEP 8?

- PEP stands for *Python Enhancement Proposal*.
- PEP 8 is a style guide for writing clean, readable, and consistent Python code.
- It is not a rule enforced by Python, but following it makes your code easy to read, debug, and share with others.

Main Points of PEP 8:

1.Indentation

- Use 4 spaces per indentation level.

```
if True:
    print("Follow 4 spaces indentation")
```

2.Maximum Line Length

- Keep code lines up to 79 characters.
- For docstrings/comments, keep lines up to 72 characters.

3.Blank Lines

- Use blank lines to separate functions, classes, and code blocks for readability.

```
def function_one():
    pass
```

```
def function_two():
    pass
```

4.Imports

- Always import standard libraries first, then third-party libraries, and then local imports.
- Each import on a separate line:

```
import os
import sys
```

5.Naming Conventions

- Variables and functions: snake_case (lowercase with underscores).
- Classes: CamelCase.
- Constants: UPPER_CASE.

```
my_variable = 10
def calculate_sum():
    pass
class MyClass:
    pass
PI = 3.14
```

6.Whitespace

No extra spaces inside parentheses, brackets, or before commas.

```
# Correct
my_list = [1, 2, 3]
```

```
# Wrong
my_list = [ 1 , 2 , 3 ]
```

7.Comments

- Write comments to explain your code, not obvious things.
- Use # for single-line comments, """...""" for docstrings.

```
# This function calculates factorial
def factorial(n):
    """Return factorial of a number."""
    if n == 0:
        return 1
return n * factorial(n - 1)
```

2) Indentation, comments, and naming conventions in Python. Writing readable and maintainable code.

1) Indentation in Python

- Python does not use curly braces {} like C/Java.
- Instead, it uses indentation (spaces) to define code blocks.
- PEP 8 recommends 4 spaces per indentation level.

```
# Correct:
if True:
    print("This is indented with 4 spaces")
    for i in range(3):
        print(i)
# Wrong:
if True:
print("No indentation")   # ERROR
```

2) Comments in Python

Comments help explain the logic of your code so others (and future you) can understand it.

(1) Single-line comment

```
# This prints Hello World
print("Hello World")
```

(2)Multi-line comment

```
"""Return a greeting message with the given name."""
```

(3) Naming Conventions

PEP 8 suggests consistent naming so code is clear and predictable.

- Variables & functions → snake_case

```
user_name = "Dhiraj"
def calculate_sum(a, b):
    return a + b
```

- Classes → CamelCase

```
class StudentRecord:
    pass
```

- Constants → UPPER_CASE

```
PI = 3.14159
MAX_SPEED = 120
```

(4)Writing Readable & Maintainable Code

- Use meaningful names (not x, y if possible).

```
# Bad
def f(a, b):
    return a * b


# Good
def calculate_area(length, width):
    return length * width
```

- Keep functions small – each function should do one thing only.
- Use blank lines to separate logical sections.
- Follow consistency in style → makes it easier for teams to read.

3. Core Python Concepts

1) Understanding data types: integers, floats, strings, lists, tuples, dictionaries, sets.

A) Integers (int)

- Whole numbers (positive, negative, or zero).
- No decimal point.

```
age = 20
temperature = -5
print(type(age))   # <class 'int'>
```

B) Floats (float)

- Numbers with **decimal points**.
- Can represent real numbers.

```
pi = 3.14159
height = 5.9
print(type(pi))   # <class 'float'>
```

C) Strings (str)

- Sequence of characters inside **quotes** (single ' ' or double " ").
- Used for text.

```
name = "Python"
greeting = 'Hello World'
print(type(name))   # <class 'str'>
```

D) Lists (list)

- Ordered collection of items.
- Items can be of different data types.
- Mutable → can be changed after creation.

```
fruits = ["apple", "banana", "cherry", 10, 5.5]
print(fruits[0])      # apple
fruits.append("mango")  # add new item
print(fruits)
```

E) Tuples (tuple)

- Ordered collection like a list, but **immutable** (cannot be changed).
- Useful for fixed data.

```
coordinates = (10, 20)
print(coordinates[1])   # 20
# coordinates[0] = 15   #  Error (cannot change tuple)
```

F) Dictionaries (dict)

- Store data in key–value pairs.
- Keys must be unique & immutable, values can be anything.

```
student = {
   "name": "Dhiraj",
   "age": 17,
   "marks": 92.5
}
print(student["name"])   # Dhiraj
student["age"] = 18      # update value
```

G) Sets (set)

- Unordered collection of unique items (no duplicates).
- Used for operations like union, intersection.

```
numbers = {1, 2, 3, 3, 4}
print(numbers)   # {1, 2, 3, 4} (duplicate removed)


a = {1, 2, 3}
b = {3, 4, 5}
print(a.union(b))        # {1, 2, 3, 4, 5}
print(a.intersection(b)) # {3}
```

Table

Data Type	Example	Mutable?
Integer (int)	10, -3, 0	No
Float (float)	3.14, -2.5, 0.0	No
String (str)	"Hello", 'Python'	No
List (list)	["a", 1, 2.5]	Yes
Tuple (tuple)	(1, 2, 3)	No
Dictionary (dict)	{"name": "Dhiraj", "age": 17}	Yes
Set (set)	{1, 2, 3}	Yes

2) Python variables and memory allocation.

What is a Variable?

- A **variable** is a name that stores a value in memory.
- Think of it as a **label** pointing to an object in memory.

x = 10

y = "Python"

Here:

- x points to an integer object 10 in memory.
- y points to a string object "Python".

How Python Handles Memory

- Python uses **dynamic typing** → you don't need to declare type, it's decided at runtime.
- Variables are just **references (pointers)** to objects in memory.

Example:

```
a = 100
b = a   # both point to same memory object 100
print(id(a), id(b))   # same memory id
```

If you change a, a new object may be created:

```
a = 200
print(id(a))   # new memory address
```

- ✓ This means Python **reuses objects** when possible (called *interning* for small integers and strings).

3) Python operators: arithmetic, comparison, logical, bitwise.

Python Operators

- An operator is a symbol that tells Python to perform a specific operation on one or more values (operands).

- Think of it like this:
  - **Operands** → The values (numbers, variables).
  - **Operator** → The tool/action.

    Example:

    x = 10

    y = 3

    z = x + y
  - x and y are operands (10 and 3).
  - + is the operator (addition).
  - z stores the result (13).

(1)Arithmetic Operators

- Work just like math.

Operator	Meaning	Example	Result
+	Addition	10 + 5	15
-	Subtraction	10 - 5	5
*	Multiplication	10 * 5	50
/	Division	10 / 3	3.33
//	Floor Division	10 // 3	3
%	Modulus (Remainder)	10 % 3	1

3) Comparison (Relational) Operators

- Used to compare values → result is always **True / False**.

Operator	Meaning	Example	Result
==	Equal	10 == 10	True
!=	Not equal	10 != 5	True
>	Greater than	10 > 5	True
<	Less than	10 < 5	False
>=	Greater or equal	10 >= 10	True
<=	Less or equal	5 <= 10	True

4) Logical Operators

- Used for combining conditions.

Operator	Meaning	Example	Result
and	Both must be True	(10 > 5) and (5 > 2)	True
or	Any one True	(10 > 5) or (5 < 2)	True
not	Reverses result	not (10 > 5)	False

5) Bitwise Operators

- Work at the **binary level** (0s and 1s).

Let’s take:

a = 6 → (110 in binary)

b = 3 → (011 in binary)

Operator	Meaning	Example	Binary Result	Decimal
&	AND	6 & 3	010	2
`	`	OR	`6	3`
^	XOR	6 ^ 3	101	5
~	NOT	~6	flips bits	-7
<<	Left shift	6 << 1	1100	12
>>	Right shift	6 >> 1	11	3

4. Conditional Statements

1) Introduction to conditional statements: if, else, elif.

What are Conditional Statements?

- Conditional statements let a program make decisions.
- They check a condition (True/False) and run different code blocks depending on the result.

(1) if Statement

- Runs a block of code **only if the condition is True**.

Ex :-

```
age = 18
if age >= 18:
    print("You are an adult.")
```

(2) if–else Statement

- Adds an **alternative block** if condition is False.

Ex :-

```
age = 15
if age >= 18:
    print("You are an adult.")
else:
    print("You are a minor.")
```

(3) if–elif–else Statement

- Used when you need to check **multiple conditions**.
- Python checks each condition in order → runs the first True block.

```
marks = 75

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

2) Nested if-else conditions.

What are Nested if–else?

- Sometimes, one decision depends on another.
- In such cases, we put an if inside another if (or else) → this is called nested if–else.

Basic Syntax

```
if condition1:
    if condition2:
        # Code if both are True
    else:
        # Code if condition1 True but condition2 False
else:
    # Code if condition1 False
```

Example: Checking Positive/Negative and Even/Odd

```
num = 10

if num >= 0:
    if num % 2 == 0:
        print("Positive Even number")
    else:
        print("Positive Odd number")
else:
    print("Negative number")
```

Flow:

num = 10

First check: num >= 0 → True

Then check: num % 2 == 0 → True

Output → Positive Even number

5. Looping (For, While)

1) Introduction to for and while loops.

What is a Loop?

- A loop is used when you want to repeat a block of code multiple times.
- Instead of writing code again and again, you use a loop.

Types of Loops in Python

1) for loop

- Used when you know **how many times** you want to repeat.
- It **iterates over a sequence** (like list, tuple, string, range, etc.).

Syntax:

```
for variable in sequence:
    # code block
```

Example:

```
for i in range(5):  # runs 5 times (0 to 4)
    print("Hello", i)
```

Output:

```
Hello 0
Hello 1
Hello 2
Hello 3
Hello 4
```

2) while loop

- Used when you **don’t know the exact number of repetitions**, but you repeat until a condition becomes False.

Syntax:

```
while condition:
    # code block
```

Example:

```
x = 1
while x <= 5:
    print("Value:", x)
    x = x + 1
```

Output:

Value: 1

Value: 2

Value: 3

Value: 4

Value: 5

2) How loops work in Python.

1) How a for loop works

- A for loop iterates through a sequence (list, tuple, string, or range).

Example:

```
for i in range(3):
    print("i =", i)
```

Step by Step:

- First → i = 0 → print
- Next → i = 1 → print
- Next → i = 2 → print
- No more numbers → loop ends

Output:

```
i = 0
i = 1
i = 2
```

So the for loop automatically stops when the sequence is finished.

2) How a while loop works

- A while loop repeats until the condition becomes False.

Example:

```
x = 1
while x <= 3:
    print("x =", x)
    x = x + 1
```

Step by Step:

- Start: x = 1 → condition True → print → x becomes 2
- Next: x = 2 → condition True → print → x becomes 3
- Next: x = 3 → condition True → print → x becomes 4
- Now: x = 4 → condition False → loop stops

Output:

x = 1

x = 2

x = 3

- for loop → works with a collection/sequence → stops automatically.
- while loop → works with a condition → can become infinite if you forget to update the variable.

3) Using loops with collections (lists, tuples, etc.).

What is a Collection?

- A collection is a group of multiple items stored in one variable. Examples: list, tuple, string, set, dictionary.

1) Looping through a List

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

Output:

apple

banana

cherry

Meaning → The loop picks each item from the list one by one.

2) Looping through a Tuple

```
colors = ("red", "green", "blue")
for c in colors:
    print(c)
```

Output:

red

green

blue

Same as list, but tuple is immutable (can't be changed).

3) Looping through a String

```
for ch in "Python":
    print(ch)
```

Output:

P

y

t

h

o

n

Here, loop goes through each character.

4) Looping through a Set

```
nums = {10, 20, 30}
for n in nums:
    print(n)
```

Output (order may vary):

10

20

30

Sets don't maintain order → items may come in different order.

5) Looping through a Dictionary

A dictionary has key-value pairs.
You can loop through:

- keys
- values
- both (items)

```
student = {"name": "Alice", "age": 20, "grade": "A"}


# Loop through keys
for k in student:
    print(k)


# Loop through values
for v in student.values():
    print(v)


# Loop through both
for k, v in student.items():
    print(k, "=", v)
```

Output:

```
name
age
grade
Alice
20
A
```

name = Alice

age = 20

grade = A

6. Generators and Iterators

1) Understanding how generators work in Python.

What are Generators?

- Generators are special types of functions that allow you to generate a sequence of values one at a time instead of returning them all at once like normal functions.

- They are used to save memory and make your program more efficient, especially when dealing with large data.

How Generators Work

- A normal function uses return and ends after returning a value.
- A generator function uses yield instead of return.
- Each time the generator's __next__() (or next()) method is called, the function resumes from where it left off and continues until the next yield.

2) Difference between yield and return.

Feature	return	yield
Function Type	Used in normal functions	Used in generator functions
What It Does	Returns a single value and ends the function	Returns a generator object and pauses the function
Execution	Function terminates after return	Function saves its state and resumes from the last yield when called again
Usage	Used when we need only one result	Used when we need to produce a series of results one by one
Memory Usage	Stores all data in memory	More memory efficient (generates values one at a time)
Keyword Behavior	Ends function completely	Temporarily suspends function execution

3) Understanding iterators and creating custom iterators.

What is an Iterator?

- An iterator is an object that allows you to access elements of a sequence one at a time without needing to know how the data is stored internally.

- It follows the Iterator Protocol, which means:
    - The object must have a method called __iter__()
    - The object must have a method called __next__()

Together, these methods make an object *iterable* and *iterative*.

How Iterators Work

- When you pass an iterable (like a list, tuple, or string) to the iter() function, Python returns an iterator object.

- Each time you call next() on the iterator, it returns the next value.

- Once all items are exhausted, calling next() again raises a StopIteration exception — signaling the end of the sequence.

7. Functions and Methods

1) Defining and calling functions in Python.

What is a Function?

A function in Python is a block of code that performs a specific task. It helps make your program modular, reusable, and easier to read.

Defining a Function

We use the def keyword to define a function.

Syntax:

```
def function_name(parameters):
    # block of code
    return value   # (optional)
```

- def – used to define a function
- function_name – name you choose for the function
- parameters – data passed to the function (optional)
- return – sends a result back to the caller (optional)

Calling a Function

- After defining, you **call** (or use) the function by writing its name followed by parentheses:

```
function_name(arguments)
```

2) Function arguments (positional, keyword, default).

Function Arguments in Python

- When we define a function, we often pass **arguments (inputs)** into it so it can work with different data.

- Python supports several types of arguments. The three most common are:

1. Positional Arguments

- These are the most basic type of arguments.
- They are **matched by their position** (order) in the function call.
- The number and order of arguments in the call must match the definition.

Example:

```
def student_info(name, age):
    print("Name:", name)
    print("Age:", age)

# Calling with positional arguments
student_info("Aarav", 16)
```

Output:

Name: Aarav

Age: 16

-- Here:

- "Aarav" is passed to name (1st parameter)
- 16 is passed to age (2nd parameter)

-- If order is wrong:

student_info(16, "Aarav") # Wrong order!

Output:

Name: 16

Age: Aarav

So, **position matters** here.

2. Keyword Arguments

- In this type, we pass values by **explicitly naming the parameter**.
- Order **doesn't matter** when using keyword arguments.

Example:

```
def student_info(name, age):
    print("Name:", name)
    print("Age:", age)

# Calling with keyword arguments
student_info(age=16, name="Aarav")
```

Output:

Name: Aarav

Age: 16

-- Advantage: Your code becomes **more readable** and order-independent.

3. Default Arguments

- You can assign **default values** to parameters in the function definition.
- If no value is passed for that argument, the default value is used.

Example:

```
def greet(name, message="Good Morning"):
    print("Hello", name + "!", message)

# Calling function with both arguments
greet("Aarav", "How are you?")
# Calling function with only required argument
greet("Aarav")
```

Output:

```
Hello Aarav! How are you?
Hello Aarav! Good Morning
```

3) Scope of variables in Python.

In Python, variables mainly have **two types of scope**:

1. **Local Scope**
2. **Global Scope**

1. Local Scope

- A variable declared **inside a function** is called a **local variable**.
- It is accessible **only within that function** and is **created when the function starts** and **destroyed when it ends**.

Example:

```
def show():
    x = 10  # Local variable
    print("Inside function:", x)

show()
print(x)  # Error: x is not defined outside the function
```

--Here, `x` is local to the function `show()` and cannot be used outside.

2. Global Scope

- A variable declared **outside any function** is called a **global variable**.
- It can be **accessible from anywhere** in the program — inside or outside functions.

Example:

```
x = 100  # Global variable

def display():

    print("Inside function:", x)

display()

print("Outside function:", x)
```

➞ Here, x can be used both inside and outside the function.

4) Built-in methods for strings, lists, etc.

String Methods

upper(), lower(), title(), capitalize(), strip(), lstrip(), rstrip(), replace(), split(), join(), find(), count(), startswith(), endswith(), isdigit(), isalpha(), islower(), isupper(), swapcase(), center(), zfill()

List Methods

append(), insert(), extend(), remove(), pop(), clear(), sort(), reverse(), count(), index(), copy()

Tuple Methods

count(), index()

Dictionary Methods

keys(), values(), items(), get(), update(), pop(), popitem(), clear(), copy(), fromkeys(), setdefault()

Set Methods

add(), remove(), discard(), pop(), clear(), union(), intersection(), difference(), symmetric_difference(), update(), copy(), issubset(), issuperset(), isdisjoint()

8. Control Statements (Break, Continue, Pass)

1) Understanding the role of break, continue, and pass in Python loops.

1.break — Exit the loop completely

Purpose: Immediately stops the loop (for or while) and jumps out of it, even if the loop condition is still true.

Example:

```
for i in range(5):
    if i == 3:
        break
    print(i)
```

Output:

```
0
1
2
```

2.continue — Skip the current iteration

Purpose: Skips the rest of the code inside the loop for the current iteration and moves to the next iteration.

Example:

```
for i in range(5):
    if i == 3:
        continue
    print(i)
```

Output:

```
0
1
2
4
```

3.pass — Do nothing (placeholder)

Purpose: Does nothing at all. It is used as a placeholder when a statement is syntactically required but no action is needed yet.

Example:

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

Output:

```
0
1
2
3
4
```

9. String Manipulation

1) Understanding how to access and manipulate strings.

Access characters: s = "hello" → s[0] == 'h', s[-1] == 'o'

Slice (substrings): s[1:4] == 'ell', s[:3] == 'hel', s[2:] == 'llo'

Length: len(s) → 5

Concatenate / repeat: s + " world" → "hello world", s * 2 → "hellohello"

Iterate: for ch in s: print(ch)

Common methods:

- s.upper() → "HELLO"
- s.lower() → "hello"
- s.strip() removes outer spaces
- s.split() → list of words
- " ".join(list) → join words into string
- s.replace("l","L") → "heLLo"
- s.find("lo") → index of substring or -1 if not found

Formatting: f"Name: {name}, Age: {age}" for easy templates

Immutability: strings can't be changed in place — create a new string instead.

2) Basic operations: concatenation, repetition, string methods (upper(), lower(), etc.).

Concatenation → Joining two strings using +
Example: "Hello" + " World" → "Hello World"

Repetition → Repeating a string using *
Example: "Hi" * 3 → "HiHiHi"

String Methods → Common built-in functions:

- upper() → Converts to uppercase → "hello".upper() → "HELLO"
- lower() → Converts to lowercase → "HELLO".lower() → "hello"
- title() → Capitalizes first letter of each word
- strip() → Removes extra spaces
- replace("a","o") → Replaces characters
- split() → Splits string into list of words

3) String slicing.

What is String Slicing?

String slicing means **extracting a specific part** of a string using indexes. Strings in Python are **indexed** (each character has a position number).

Example:

```
s = "PYTHON"

# Index:  0 1 2 3 4 5
```

Syntax:

string[start : end : step]

- **start** → position to begin slicing (included)
- **end** → position to stop (excluded)
- **step** → how many characters to skip (default is 1)

Examples:

```
s = "PYTHON"

print(s[0:3])    # Output: 'PYT' (from index 0 to 2)

print(s[2:5])    # Output: 'THO'

print(s[:4])     # Output: 'PYTH' (start from 0)

print(s[3:])     # Output: 'HON'  (till end)

print(s[::2])    # Output: 'PTO'  (skip 1 character)

print(s[::-1])   # Output: 'NOHTYP' (reverse string)
```

10. Advanced Python (map(), reduce(), filter(), Closures and Decorators)

1) How functional programming works in Python.

- Functional programming is a **programming style** that focuses on using **functions** to perform tasks instead of writing step-by-step instructions.

- In this approach, the main idea is to use **pure functions**, which always give the same output for the same input and do not change data or variables outside the function.

- Python supports functional programming through **built-in functions** like map(), filter(), and reduce(), and also allows the use of **lambda (anonymous) functions**.

It helps to make programs:

- **Simple and clear**
- **Easy to test and debug**
- **Less error-prone**

Points:

- Functions are treated as values (can be passed or returned).
- Avoids using loops and mutable data.
- Focuses on *what to do* rather than *how to do it*.
- Encourages reusability and modular code.

2) Using map(), reduce(), and filter() functions for processing data.

1. map(function, sequence)

- Applies a function to each item in a sequence.
- Returns a new sequence (iterator).
  Example:
  map(str.upper, ["a", "b", "c"]) → ['A', 'B', 'C']

2. filter(function, sequence)

- Filters items based on a condition (function returns True/False).
  Example:
  filter(lambda x: x % 2 == 0, [1,2,3,4]) → [2,4]

3. reduce(function, sequence)

- Combines all elements into a single value.
- Comes from the functools module.
  Example:
  reduce(lambda x, y: x + y, [1,2,3,4]) → 10

In short:

- map() → Transform data
- filter() → Select data
- reduce() → Combine data

3) Introduction to closures and decorators.

Closures and Decorators in Python

Closures:

A **closure** is a function defined inside another function that remembers the **values of variables** from the outer function even after the outer function has finished executing.

It helps in **data hiding** and creating **function factories**.

Example (concept):
An inner function uses variables from its outer function — this forms a closure.

Key Points:

- Inner function is returned by outer function.
- Remembers outer variable values.
- Useful in callbacks and decorators.

Decorators:

A **decorator** is a special function that **modifies or adds new features** to another function **without changing its code**.
It is written using the **@decorator_name** syntax before a function definition.

Key Points:

- Reusable and clean way to add functionality.
- Commonly used for logging, authentication, or measuring execution time.
- Built-in decorators include @staticmethod, @classmethod, and @property.