***Module 6 Python Fundamentals***

***Introduction to Python***

**Theory : 1 Introduction to Python**

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

Python is a **simple, high-level, interpreted programming language** that is widely used for software development, web development, data science, artificial intelligence, and automation. It was created by **Guido van Rossum** in **1991**. Python is popular because of its easy-to-read syntax, which is similar to the English language, making it beginner-friendly as well as powerful for professionals.

**Key Features of Python**

1. **Simple and Easy to Learn**
   * Python code is easy to write and understand.
   * Its syntax is close to English, reducing complexity for beginners.
2. **High-Level Language**
   * Developers do not need to manage memory or worry about low-level operations.
   * Python handles details like memory management automatically.
3. **Interpreted Language**
   * Python executes code line by line (not compiled before running).
   * This makes debugging and testing easier.
4. **Portable**
   * Python programs can run on different platforms (Windows, Mac, Linux) without changes.
5. **Extensive Libraries**
   * Python has a huge standard library and supports external libraries for AI, machine learning, web development, etc.
6. **Object-Oriented**
   * Supports classes, objects, inheritance, and encapsulation for structured programming.
7. **Dynamically Typed**
   * No need to declare variable types explicitly; Python detects them at runtime.
8. **Open-Source**
   * Python is free to use and has a large global community for support and development.

### *History and evolution of Python.*

**History and Evolution of Python**

1. **Origin (Late 1980s – Early 1990s)**
   * Python was created by **Guido van Rossum** at **Centrum Wiskunde & Informatica (CWI)** in the Netherlands.
   * The idea came in **1989**, during Christmas holidays, when Guido wanted a simple scripting language that was easy to learn and fun to use.
   * The language was inspired by **ABC (a teaching language)**, but with improvements such as exception handling and extensibility.
   * Name “Python” was inspired by the British comedy series **“Monty Python’s Flying Circus”**, not the snake.
2. **Python 1.0 (1991)**
   * Officially released in **1991**.
   * Features included: functions, exception handling, core data types (str, list, dict), and modules.
3. **Python 2.x Series (2000)**
   * Released in **2000** with many improvements.
   * Introduced:
     + List comprehensions
     + Garbage collection (using reference counting)
     + Unicode support
   * However, Python 2 had limitations and design issues, making migration to future versions difficult.
4. **Python 3.x Series (2008 – Present)**
   * Released in **2008** as a major upgrade, not fully backward-compatible with Python 2.
   * Key changes:
     + print became a function (print())
     + Better Unicode support (strings are Unicode by default)
     + Improved libraries and syntax consistency
   * Python 2 officially **retired in January 2020**.
5. **Modern Python (2020s – Now)**
   * Python continues to evolve with regular updates (latest stable versions are **Python 3.10, 3.11, 3.12, and 3.13**).
   * New features include:
     + Pattern matching (match statement in Python 3.10)
     + Performance improvements
     + Simplified syntax and developer-friendly updates
   * Python is now widely used in **AI, Machine Learning, Web Development, Data Science, Cybersecurity, and Automation**.

**Summary**

* **1989** → Idea of Python by Guido van Rossum
* **1991** → Python 1.0 released
* **2000** → Python 2.0 introduced
* **2008** → Python 3.0 released (major shift)
* **2020** → End of Python 2 support
* **Today** → Python 3.x is one of the most popular programming languages worldwide

### *Advantages of using Python over other programming languages.*

1. **Simple and Easy to Learn**
   * Python’s syntax is close to English, making it easier to read and write compared to C, C++, or Java.
   * Beginners can learn it faster with less effort.
2. **High Productivity**
   * Fewer lines of code are needed to perform tasks compared to Java or C++.
   * Developers can focus more on solving problems than dealing with complex syntax.
3. **Large Standard Library**
   * Comes with a huge collection of built-in modules (for file handling, networking, regular expressions, etc.).
   * Reduces the need to write code from scratch.
4. **Cross-Platform and Portable**
   * Python programs can run on Windows, macOS, and Linux without modification.
   * This gives it an edge over languages that require platform-specific adjustments.
5. **Strong Community Support**
   * Python has one of the largest developer communities.
   * Easy access to tutorials, documentation, and third-party libraries.
6. **Versatile Applications**
   * Widely used in **Data Science, Machine Learning, Artificial Intelligence, Web Development, Automation, Cybersecurity, Game Development**, and more.
   * Other languages are often specialized in fewer areas.
7. **Integration and Extensibility**
   * Python can easily integrate with languages like C, C++, Java, and .NET.
   * Can be embedded into other applications to provide scripting capabilities.
8. **Rapid Prototyping**
   * Because of its simplicity and libraries, Python allows developers to build prototypes and test ideas quickly.
   * Faster development cycle than languages like C++ or Java.
9. **Open Source and Free**
   * Python is free to download and use, unlike some proprietary programming languages.
10. **Dynamic Typing**

* No need to declare variable types explicitly (unlike C, C++, or Java).
* Python determines the type automatically at runtime, making coding faster and more flexible.

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

1. Go to the official website → [python.org](https://www.python.org/downloads/).
2. Download the latest stable version of Python (e.g., Python 3.x).
3. Run the installer:
   * ✅ Check the option **“Add Python to PATH”** before installing.
   * Click **Install Now** and complete the setup.
4. Verify installation:
   * Open Command Prompt (Windows) or Terminal (Mac/Linux).
   * Type:
   * python --version
   * You should see the installed Python version.

## **2. Installing Anaconda (For Data Science & AI Users)**

Anaconda is a Python distribution with built-in libraries for **Data Science, Machine Learning, and Scientific Computing**.

1. Download Anaconda → anaconda.com.
2. Install it with default settings.
3. Open **Anaconda Navigator** or **Anaconda Prompt**.
4. Create a new environment (optional):
5. conda create -n myenv python=3.11
6. conda activate myenv
7. Launch **Jupyter Notebook**, **Spyder**, or **VS Code** from Anaconda Navigator.

## **3. Setting Up PyCharm (For Beginners & Professionals in Software Development)**

PyCharm is a powerful IDE (Integrated Development Environment) for Python.

1. Download PyCharm → [jetbrains.com/pycharm](https://www.jetbrains.com/pycharm/download/).
   * Choose **Community Edition** (free) or **Professional Edition** (paid).
2. Install PyCharm and open it.
3. Create a new project:
   * File → New Project → Select Python Interpreter.
   * Choose **Existing Python (from your system or Anaconda)** or create a **new virtual environment**.
4. Start writing Python code in the editor and run programs directly.

## **4. Setting Up Visual Studio Code (VS Code – Lightweight & Popular Editor)**

VS Code is a flexible editor widely used for Python, Web Development, and AI projects.

1. Download VS Code → code.visualstudio.com.
2. Install and open VS Code.
3. Install Python extension:
   * Go to **Extensions (Ctrl+Shift+X)** → Search for **Python** → Install.
4. Set Interpreter:
   * Press **Ctrl+Shift+P** → Type Python: Select Interpreter → Choose installed Python or Anaconda environment.
5. Create a .py file and start coding.
6. Run code:
   * Press **Ctrl+F5** (Run without debugging).

### *Writing and executing your first Python program.*

### ****Step 1: Open Python Environment****

You can use:

* **IDLE (comes with Python installation)**
* **VS Code / PyCharm / Jupyter Notebook / Anaconda Navigator**
* Or simply **Command Prompt / Terminal**

### ****Step 2: Write Your First Program****

Type the following code in a new file (hello.py) or directly in the Python shell:

# This is my first Python program

print("Hello, World!")

### ****Step 3: Save the Program****

* If using an editor (like VS Code or PyCharm), save the file as **hello.py**.
* .py is the extension for Python files.

### ****Step 4: Execute the Program****

#### **Option 1: Using Command Prompt/Terminal**

1. Open **Command Prompt** (Windows) or **Terminal** (Mac/Linux).
2. Navigate to the folder where the file is saved. Example:
3. cd Desktop
4. Run the program:
5. python hello.py
6. Output will be:
7. Hello, World!

#### **Option 2: Using IDLE**

1. Open **IDLE (Python’s default editor)**.
2. Write the program and press **F5** (Run).
3. You’ll see the output:
4. Hello, World!

#### **Option 3: Using VS Code or PyCharm**

* Click the **Run button** (▶️) in the editor.
* Output will appear in the terminal window inside the IDE.

### ✅ Output:

Hello, World!

⚡ **In short:**

1. Install Python.
2. Write code in a .py file.
3. Run it using python filename.py.
4. See your first output! 🎉

### Theory : 2 Programming Style

### *Understanding Python’s PEP 8 guidelines.*

### ****What is PEP 8?****

* **PEP** stands for **Python Enhancement Proposal**.
* **PEP 8** is the official **style guide for Python code**, written by **Guido van Rossum** and others.
* Its goal is to make Python code **readable, consistent, and maintainable** across different projects and developers.

## **Key PEP 8 Guidelines**

1. **Indentation**
   * Use **4 spaces per indentation level** (no tabs).
2. def greet():
3. print("Hello, Python!")
4. **Maximum Line Length**
   * Keep lines **≤ 79 characters** for code.
   * Docstrings and comments ≤ 72 characters.
5. **Blank Lines**
   * Use blank lines to separate functions, classes, and sections of code.
6. class Person:
7. pass
9. def my\_function():
10. pass
11. **Imports**
    * Each import should be on a separate line.
    * Standard library imports first, then third-party, then local.
12. import os
13. import sys
14. from math import sqrt
15. **Naming Conventions**
    * **Variables & functions:** lowercase\_with\_underscores
    * **Classes:** CapitalizedWords (PascalCase)
    * **Constants:** UPPERCASE\_WITH\_UNDERSCORES
16. my\_variable = 10
17. def calculate\_area():
18. pass
19. class Student:
20. pass
21. MAX\_SPEED = 120
22. **Whitespace Usage**
    * Put spaces **around operators**, **after commas**, but **not inside brackets**.
23. # Good
24. x = y + z
25. my\_list = [1, 2, 3]
26. # Bad
27. x=y+z
28. my\_list=[ 1 ,2 ,3 ]
29. **Comments & Docstrings**
    * Use comments to explain code.
    * Use **docstrings ("""...""")** for functions, classes, and modules.
30. def add(a, b):
31. """Return the sum of two numbers."""
32. return a + b
33. **Consistency**
    * Be consistent with style throughout the project.

## **Why PEP 8 is Important?**

* Makes code **clean, professional, and easy to read**.
* Helps multiple developers collaborate effectively.
* Reduces bugs and confusion caused by inconsistent formatting.

### *Indentation, comments, and naming conventions in Python.*

## **1. Indentation in Python**

* **Indentation** means spaces at the beginning of a line.
* In Python, indentation is **not just for readability** → it defines **code blocks**.
* Standard: **4 spaces per indentation level** (never mix tabs and spaces).

✅ Example (Correct):

def greet():

print("Hello")

print("Welcome to Python")

❌ Example (Incorrect – will cause IndentationError):

def greet():

print("Hello")

print("Welcome to Python")

## **2. Comments in Python**

Comments are used to explain code. Python ignores them at execution.

* **Single-line comments** → use #

# This function adds two numbers

def add(a, b):

return a + b

* **Multi-line comments / Docstrings** → use triple quotes """ ... """

def add(a, b):

"""

This function takes two numbers

and returns their sum.

"""

return a + b

## **3. Naming Conventions in Python (PEP 8 Style Guide)**

* **Variables & Functions** → lowercase\_with\_underscores
* student\_name = "Meet"
* def calculate\_area():
* pass
* **Classes** → PascalCase (CapWords style)
* class StudentRecord:
* pass
* **Constants** → UPPERCASE\_WITH\_UNDERSCORES
* MAX\_SPEED = 120
* PI = 3.14159
* **Private/Internal variables** → start with \_ (single underscore)
* \_hidden\_value = 42

### *Writing readable and maintainable code.*

## **Follow PEP 8 Guidelines**

* Use **consistent indentation (4 spaces)**.
* Keep line length ≤ 79 characters.
* Follow naming conventions for variables, functions, classes, and constants.

✅ Example:

class Student:

def \_\_init\_\_(self, name, roll\_no):

self.name = name

self.roll\_no = roll\_no

## **2. Use Meaningful Names**

* Variable, function, and class names should describe their purpose.

✅ Example:

# Good

total\_price = calculate\_total(items)

# Bad

x = func(a)

## **3. Write Comments and Docstrings**

* Use comments for complex logic.
* Use **docstrings** for functions, classes, and modules to explain their purpose.

✅ Example:

def calculate\_area(radius):

"""Return the area of a circle given its radius."""

return 3.14 \* radius \* radius

## **4. Break Code into Functions and Modules**

* Don’t write long blocks of code → split into reusable functions.
* Group related functions into modules.

✅ Example:

def get\_user\_input():

return input("Enter your name: ")

def greet\_user(name):

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

name = get\_user\_input()

greet\_user(name)

## **5. Use Consistent Formatting**

* Add spaces around operators and after commas.
* Use blank lines to separate functions/classes.

✅ Example:

# Good

x = y + z

numbers = [1, 2, 3]

# Bad

x=y+z

numbers=[1,2,3]

## **6. Avoid Repetition (DRY Principle – Don’t Repeat Yourself)**

* If you need the same logic multiple times, write a function instead of duplicating code.

✅ Example:

# Instead of repeating the same code:

print(len([1, 2, 3]))

print(len([4, 5, 6]))

# Use a function:

def list\_length(lst):

return len(lst)

print(list\_length([1, 2, 3]))

print(list\_length([4, 5, 6]))

## **7. Handle Errors Gracefully**

* Use exceptions (try-except) instead of letting the program crash.

✅ Example:

try:

number = int(input("Enter a number: "))

except ValueError:

print("Invalid input! Please enter a number.")

### Theory: 3 core python Concepts

### *Understanding data types: integers, floats, strings, lists, tuples, dictionaries,* *sets.*

## **1. Integers (**int**)**

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

✅ Example:

x = 10

y = -5

print(type(x)) # <class 'int'>

## **2. Floats (**float**)**

* Numbers with decimal points or in scientific notation.

✅ Example:

pi = 3.14

gravity = -9.81

exp\_num = 2.5e3 # 2500.0

print(type(pi)) # <class 'float'>

## **3. Strings (**str**)**

* Sequence of characters enclosed in single ('), double (") or triple quotes (""").

✅ Example:

name = "Python"

greeting = 'Hello'

message = """This is

a multi-line string"""

print(type(name)) # <class 'str'>

## **4. Lists (**list**)**

* Ordered, **mutable** (changeable) collection.
* Can contain mixed data types.

✅ Example:

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

fruits[1] = "mango" # list is mutable

print(fruits) # ['apple', 'mango', 'cherry']

## **5. Tuples (**tuple**)**

* Ordered, **immutable** collection.
* Useful when data should not be changed.

✅ Example:

coordinates = (10, 20, 30)

print(coordinates[0]) # 10

# coordinates[1] = 50 → ❌ Error (immutable)

## **6. Dictionaries (**dict**)**

* Key–value pairs.
* Keys are unique, values can be of any type.

✅ Example:

student = {"name": "Meet", "age": 20, "marks": 85}

print(student["name"]) # Meet

## **7. Sets (**set**)**

* Unordered, **unique elements only** (no duplicates).
* Useful for mathematical set operations (union, intersection).

✅ Example:

numbers = {1, 2, 3, 3, 4}

print(numbers) # {1, 2, 3, 4}

## ✅ **Summary Table**

| **Data Type** | **Ordered?** | **Mutable?** | **Example** |
| --- | --- | --- | --- |
| **int** | N/A | N/A | 10, -5, 0 |
| **float** | N/A | N/A | 3.14, -9.81, 2.5e3 |
| **str** | Yes | Immutable | "Hello", 'Python' |
| **list** | Yes | Mutable | ["apple", "banana"] |
| **tuple** | Yes | Immutable | (10, 20, 30) |
| **dict** | No | Mutable | {"name": "Meet", "age": 20} |
| **set** | No | Mutable | {1, 2, 3} |

### *Python variables and memory allocation.*

## **1. What is a Variable?**

* A **variable** is a name that refers to a value stored in memory.
* In Python, variables are created automatically when you assign a value.
* Unlike C/C++, you **don’t need to declare the type** — Python decides it at runtime.

✅ Example:

x = 10 # integer

name = "Meet" # string

pi = 3.14 # float

Here, x, name, and pi are variables referring to objects in memory.

## **2. Dynamic Typing**

* Python is **dynamically typed**, meaning the type of a variable is determined at runtime.
* The same variable can hold different types of data at different times.

✅ Example:

x = 10 # int

x = "Python" # str (type changed at runtime)

## **3. Memory Allocation in Python**

When you create a variable:

1. **An object is created in memory** to store the value.
2. **The variable name is a reference (pointer) to that object**.

✅ Example:

a = 100

b = 100

* Python optimizes memory by **reusing immutable objects** (like small integers, strings).
* Here, a and b may point to the **same memory location** for 100.

## **4. Object Identity**

* Every object in Python has a unique identity (memory address).
* You can check it using id().

✅ Example:

x = 50

print(id(x)) # prints memory address of object 50

## **5. Mutable vs Immutable Objects**

* **Immutable objects** → cannot be changed in place (e.g., int, float, str, tuple).
* **Mutable objects** → can be changed without creating a new object (e.g., list, dict, set).

✅ Example:

# Immutable

x = "Hello"

x = x + " World" # creates a NEW object

# Mutable

my\_list = [1, 2, 3]

my\_list.append(4) # modifies SAME object

## **6. Garbage Collection**

* Python uses **automatic garbage collection** (via reference counting).
* When no variable references an object, it is removed from memory automatically.

✅ Example:

x = [1, 2, 3]

y = x

del x

print(y) # [1, 2, 3] (object still exists because 'y' refers to it)

### *Python operators: arithmetic, comparison, logical, bitwise.*

## **1. Arithmetic Operators**

Used for basic mathematical operations.

| **Operator** | **Meaning** | **Example** | **Result** |
| --- | --- | --- | --- |
| + | Addition | 5 + 3 | 8 |
| - | Subtraction | 5 - 3 | 2 |
| \* | Multiplication | 5 \* 3 | 15 |
| / | Division (float) | 5 / 2 | 2.5 |
| // | Floor Division | 5 // 2 | 2 |
| % | Modulus (remainder) | 5 % 2 | 1 |
| \*\* | Exponentiation | 2 \*\* 3 | 8 |

✅ Example:

a, b = 10, 3

print(a + b) # 13

print(a / b) # 3.333...

print(a // b) # 3

## **2. Comparison (Relational) Operators**

Used to compare values, returns **True/False**.

| **Operator** | **Meaning** | **Example** | **Result** |
| --- | --- | --- | --- |
| == | Equal to | 5 == 5 | True |
| != | Not equal to | 5 != 3 | True |
| > | Greater than | 5 > 3 | True |
| < | Less than | 5 < 3 | False |
| >= | Greater or equal | 5 >= 5 | True |
| <= | Less or equal | 3 <= 5 | True |

✅ Example:

x, y = 7, 10

print(x > y) # False

print(x != y) # True

## **3. Logical Operators**

Used to combine conditional statements.

| **Operator** | **Meaning** | **Example** | **Result** |
| --- | --- | --- | --- |
| and | True if both True | (5 > 3) and (10 > 5) | True |
| or | True if any True | (5 > 3) or (10 < 5) | True |
| not | Reverses condition | not(5 > 3) | False |

✅ Example:

a, b = True, False

print(a and b) # False

print(a or b) # True

print(not a) # False

## **4. Bitwise Operators**

Used to perform operations on **binary (bit-level) data**.

| **Operator** | **Meaning** | **Example (a=6(110), b=3(011))** | **Result (decimal)** |
| --- | --- | --- | --- |
| & | AND (bits both 1) | a & b → 110 & 011 = 010 | 2 |
| ` | ` | OR (either bit 1) | `a |
| ^ | XOR (bits different) | a ^ b → 110 ^ 011 = 101 | 5 |
| ~ | NOT (invert bits) | ~a | -7 |
| << | Left shift | a << 1 → 1100 | 12 |
| >> | Right shift | a >> 1 → 011 | 3 |

✅ Example:

a, b = 6, 3

print(a & b) # 2

print(a | b) # 7

print(a ^ b) # 5

print(~a) # -7

print(a << 1) # 12

print(a >> 1) # 3

### Theory: 4 Conditional Statements

### *Introduction to conditional statements: if, else, elif.*

## **The** if **Statement**

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

✅ Example:

age = 18

if age >= 18:

print("You are eligible to vote.")

👉 Output:

You are eligible to vote.

## **2. The** if-else **Statement**

* Provides an **alternative block** if the condition is False.

✅ Example:

age = 16

if age >= 18:

print("You are eligible to vote.")

else:

print("You are NOT eligible to vote.")

👉 Output:

You are NOT eligible to vote.

## **3. The** if-elif-else **Ladder**

* Used when there are **multiple conditions**.
* elif means **else if**.
* Python checks conditions **in order**, and executes the first one that’s True.

✅ Example:

marks = 75

if marks >= 90:

print("Grade: A+")

elif marks >= 75:

print("Grade: A")

elif marks >= 60:

print("Grade: B")

else:

print("Grade: C")

👉 Output:

Grade: A

### *Nested if-else conditions.*

A **nested if-else** means placing one if (or else) statement **inside another**.  
It is used when you need to check **multiple levels of conditions**.

## **Basic Syntax**

if condition1:

if condition2:

# Code block if both condition1 and condition2 are True

else:

# Code block if condition1 is True but condition2 is False

else:

# Code block if condition1 is False

## **Example 1: Check if a number is positive, negative, or zero**

num = 5

if num >= 0:

if num == 0:

print("The number is Zero")

else:

print("The number is Positive")

else:

print("The number is Negative")

👉 Output:

The number is Positive

## **Example 2: Student grading system**

marks = 85

if marks >= 50:

if marks >= 90:

print("Grade: A+")

else:

print("Grade: A")

else:

print("Fail")

👉 Output:

Grade: A

## **Example 3: Voting eligibility**

age = 20

citizen = True

if age >= 18:

if citizen:

print("You are eligible to vote.")

else:

print("You must be a citizen to vote.")

else:

print("You are too young to vote.")

👉 Output:

You are eligible to vote.

### Theory : 5. Looping (For, While)

### *Introduction to for and while loops.*

Loops are used to **repeat a block of code multiple times** until a condition is met.  
Python mainly provides two types of loops:

1. **for loop** → used when we know **how many times** to run the loop.
2. **while loop** → used when we don’t know in advance, and want to repeat **until a condition is False**.

## **1. The** for **Loop**

* Iterates over a sequence (like list, string, range, tuple, etc.).
* Best when the number of iterations is **fixed or known**.

✅ Example 1: Using range()

for i in range(5): # runs from 0 to 4

print("Iteration:", i)

👉 Output:

Iteration: 0

Iteration: 1

Iteration: 2

Iteration: 3

Iteration: 4

✅ Example 2: Iterating over a list

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

for fruit in fruits:

print(fruit)

👉 Output:

apple

banana

cherry

## **2. The** while **Loop**

* Repeats a block of code **as long as the condition is True**.
* Useful when the number of iterations is **not fixed**.

✅ Example 1: Counting numbers

i = 1

while i <= 5:

print("Number:", i)

i += 1

👉 Output:

Number: 1

Number: 2

Number: 3

Number: 4

Number: 5

✅ Example 2: Infinite loop (⚠️ must have break to stop)

while True:

name = input("Enter your name (or 'exit' to stop): ")

if name == "exit":

break

print("Hello", name)

## **3. Key Differences**

| **Feature** | **for loop** | **while loop** |
| --- | --- | --- |
| **When to use** | Fixed/known number of iterations | Unknown/repeated until condition fails |
| **Syntax** | Iterates over sequence/range | Runs as long as condition is True |
| **Risk** | Rarely infinite | Can become infinite if condition wrong |

### *How loops work in Python.*

Loops allow a block of code to **execute repeatedly** based on a condition or sequence. Python supports two main types: **for loop** and **while loop**.

## **1. How a** for **Loop Works**

* A for loop **iterates over a sequence** (like a list, tuple, string, or range).
* On each iteration:
  1. Python picks the **next item** from the sequence.
  2. Executes the **code block** inside the loop.
  3. Moves to the **next item** until the sequence ends.

✅ Example:

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

for fruit in fruits:

print(fruit)

**Flow:**

Step 1: fruit = "apple" → print("apple")

Step 2: fruit = "banana" → print("banana")

Step 3: fruit = "cherry" → print("cherry")

Loop ends

## **2. How a** while **Loop Works**

* A while loop **repeats as long as a condition is True**.
* On each iteration:
  1. Python **checks the condition**.
  2. If True → executes the **code block**.
  3. Checks the condition again.
  4. If False → **loop ends**.

✅ Example:

i = 1

while i <= 3:

print(i)

i += 1

**Flow:**

i = 1 → True → print(1) → i = 2

i = 2 → True → print(2) → i = 3

i = 3 → True → print(3) → i = 4

i = 4 → False → Loop ends

## **3. Loop Control Flow**

* **Start** → check condition (while) or take first item (for)
* **Execute** → code inside the loop
* **Update** → move to next item or update loop variable
* **Repeat** → until condition is False (while) or sequence ends (for)
* Optional control statements:
  + break → exit loop immediately
  + continue → skip current iteration, move to next
  + pass → placeholder, does nothing

✅ Example with break and continue:

for i in range(1, 6):

if i == 3:

continue # skip 3

if i == 5:

break # stop loop

print(i)

**Output:**

1

2

4

### *Using loops with collections (lists, tuples, etc.).*

Python collections include **lists, tuples, sets, and dictionaries**. Loops are often used to **iterate through these collections**.

## **1. Using** for **Loop with Lists**

* Lists are **ordered and mutable**.
* for loops iterate through each element.

✅ Example:

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

for fruit in fruits:

print(fruit)

**Output:**

apple

banana

cherry

## **2. Using** for **Loop with Tuples**

* Tuples are **ordered but immutable**.
* Looping works the same way as lists.

✅ Example:

numbers = (1, 2, 3, 4)

for num in numbers:

print(num \* 2)

**Output:**

2

4

6

8

## **3. Using** for **Loop with Sets**

* Sets are **unordered collections of unique items**.
* The order of elements is **not guaranteed**.

✅ Example:

colors = {"red", "green", "blue"}

for color in colors:

print(color)

**Output (order may vary):**

green

blue

red

## **4. Using** for **Loop with Dictionaries**

* Dictionaries store **key-value pairs**.
* You can loop over **keys, values, or items**.

✅ Example 1: Loop over keys

student = {"name": "Meet", "age": 20, "marks": 85}

for key in student:

print(key)

**Output:**

name

age

marks

✅ Example 2: Loop over values

for value in student.values():

print(value)

**Output:**

Meet

20

85

✅ Example 3: Loop over key-value pairs

for key, value in student.items():

print(key, ":", value)

**Output:**

name : Meet

age : 20

marks : 85

## **5. Using** while **Loop with Collections**

* Less common than for, but useful when using an **index or condition**.

✅ Example:

numbers = [10, 20, 30, 40]

i = 0

while i < len(numbers):

print(numbers[i])

i += 1

**Output:**

10

20

30

40

### Theory: 6 Generators and Iterators

### *Understanding how generators work in Python*

## 1. What is a Generator?

A **generator** in Python is a special type of iterator that allows you to produce values one at a time using the yield keyword instead of returning everything at once.

👉 They are memory-efficient because they don’t store the entire sequence in memory—values are generated **on demand**.

## 2. Normal Function vs Generator

### Normal Function

def get\_numbers():

return [1, 2, 3]

* Returns a list.
* Entire list stored in memory at once.

### Generator Function

def get\_numbers():

yield 1

yield 2

yield 3

* Uses yield.
* Generates values one by one.
* Does **not** store the entire sequence in memory.

## 3. How Generators Work

When you call a generator function:

gen = get\_numbers()

* gen is a **generator object** (an iterator).
* You can iterate over it using next() or a for loop.

print(next(gen)) # 1

print(next(gen)) # 2

print(next(gen)) # 3

# If called again -> StopIteration

Or:

for num in get\_numbers():

print(num)

## 4. Why Use Generators?

* **Efficiency:** Handles large datasets without loading everything into memory.
* **Lazy evaluation:** Values are computed only when needed.
* **Readable code:** Easy to write complex iterators with yield.

Example:

def countdown(n):

while n > 0:

yield n

n -= 1

for i in countdown(5):

print(i)

Output:

5

4

3

2

1

## 5. Generator Expressions

Just like list comprehensions but with parentheses:

squares = (x\*x for x in range(5))

print(next(squares)) # 0

print(next(squares)) # 1

Much more memory efficient than [x\*x for x in range(1000000)].

## 6. Under the Hood

* A generator **remembers its state** between yield calls.
* When you call next(), execution resumes right where it left off.

### *Difference between yield and return.*

## 1. Basic Idea

* **return** → Ends the function immediately and sends back **a single value**.
* **yield** → Pauses the function, sends back a **value**, but keeps the function’s state so execution can continue from where it left off.

## 2. Example with return

def normal\_func():

return 1

return 2 # never executed

return 3 # never executed

print(normal\_func())

👉 Output:

1

* Once return is hit, function ends permanently.

## 3. Example with yield

def generator\_func():

yield 1

yield 2

yield 3

gen = generator\_func()

print(next(gen)) # 1

print(next(gen)) # 2

print(next(gen)) # 3

👉 Output:

1

2

3

* yield pauses execution after each value.
* Next call resumes from where it left off.

## 4. Key Differences

| **Feature** | **return** | **yield** |
| --- | --- | --- |
| Behavior | Ends the function permanently | Pauses function, can resume later |
| Values | Returns a **single value** | Produces a **sequence of values** (one at a time) |
| Function type | Normal function | Generator function |
| Memory usage | Can return large structures (uses more memory) | Generates values lazily (saves memory) |
| State preservation | Loses all state after returning | Remembers local variables and execution state |
| Usage | For final single result | For iteration/streaming multiple results |

## 5. Visual Analogy

* **return** → Like **finishing a task and leaving the room**.
* **yield** → Like **pausing, giving you something, then continuing when you ask again**.

### *Understanding iterators and creating custom iterators.*

# 1. What is an Iterator?

An **iterator** is an object in Python that:

1. Implements the **\_\_iter\_\_()** method → returns the iterator object itself.
2. Implements the **\_\_next\_\_()** method → returns the next item, and raises StopIteration when there are no items left.

👉 Lists, tuples, sets, dictionaries, strings, etc. are **iterables**, and they can be turned into iterators using iter().

# 2. Example: Built-in Iterator

numbers = [10, 20, 30]

it = iter(numbers) # create an iterator

print(next(it)) # 10

print(next(it)) # 20

print(next(it)) # 30

print(next(it)) # StopIteration error

✅ Each call to next() gives the next element.

# 3. Creating a Custom Iterator

To create your own iterator, you need a class with \_\_iter\_\_() and \_\_next\_\_().

### Example: Countdown Iterator

class Countdown:

def \_\_init\_\_(self, start):

self.current = start

def \_\_iter\_\_(self):

return self # an iterator must return itself

def \_\_next\_\_(self):

if self.current <= 0:

raise StopIteration

num = self.current

self.current -= 1

return num

# Using our custom iterator

for num in Countdown(5):

print(num)

👉 Output:

5

4

3

2

1

# 4. Difference Between Iterator and Iterable

* **Iterable**: Any object that can return an iterator (e.g., list, tuple, dict, string).
* **Iterator**: The actual object that produces values one at a time (via \_\_next\_\_()).

Check:

from collections.abc import Iterable, Iterator

nums = [1, 2, 3]

print(isinstance(nums, Iterable)) # True

print(isinstance(nums, Iterator)) # False

print(isinstance(iter(nums), Iterator)) # True

# 5. Why Use Custom Iterators?

* To model **infinite sequences** (like natural numbers).
* To generate data dynamically.
* To control iteration behavior beyond built-in data structures.

Example: Infinite iterator for even numbers

class EvenNumbers:

def \_\_init\_\_(self):

self.n = 0

def \_\_iter\_\_(self):

return self

def \_\_next\_\_(self):

self.n += 2

return self.n

evens = EvenNumbers()

for i in range(5):

print(next(evens))

Output:

2

4

6

8

10

### Theory: 7 Functions and Methods.

### *Defining and calling functions in Python*

# 1. What is a Function?

A **function** is a block of reusable code that performs a specific task.  
It helps make programs **modular, reusable, and organized**.

# 2. Defining a Function

In Python, you define a function with the def keyword:

def greet():

print("Hello, welcome to Python!")

Here:

* def → keyword to define a function.
* greet → function name.
* () → parentheses (can hold parameters).
* : → start of function block.
* Body → indented block of code.

# 3. Calling a Function

You call (execute) the function by writing its name followed by ():

greet()

👉 Output:

Hello, welcome to Python!

# 4. Functions with Parameters

Functions can accept **inputs** (parameters/arguments):

def greet\_user(name):

print("Hello,", name)

greet\_user("Meet") # passing argument

👉 Output:

Hello, Meet

# 5. Functions with Return Values

Functions can return a value using return:

def add(a, b):

return a + b

result = add(5, 3)

print("Sum:", result)

👉 Output:

Sum: 8

# 6. Default Parameters

You can set default values for parameters:

def greet(name="Guest"):

print("Hello,", name)

greet() # Hello, Guest

greet("Kapadiya") # Hello, Kapadiya

# 7. Multiple Return Values

Functions can return **multiple values** as a tuple:

def calculate(a, b):

return a+b, a-b, a\*b

s, d, p = calculate(10, 5)

print("Sum:", s, "Difference:", d, "Product:", p)

# 8. Types of Functions

* **Built-in functions** → e.g., len(), print(), type().
* **User-defined functions** → functions you create with def.
* **Lambda functions** → small anonymous functions:
* square = lambda x: x \* x

print(square(4)) # 16

### *Function arguments (positional, keyword, default).*

# 1. Positional Arguments

* The most common type.
* Values are assigned to parameters **in the order they are passed**.

def student\_info(name, age):

print("Name:", name)

print("Age:", age)

student\_info("Meet", 20) # order matters

👉 Output:

Name: Meet

Age: 20

⚠️ If you change the order:

student\_info(20, "Meet")

👉 Output:

Name: 20

Age: Meet

# 2. Keyword Arguments

* You specify parameter names explicitly while calling.
* Order doesn’t matter.

def student\_info(name, age):

print("Name:", name)

print("Age:", age)

student\_info(age=20, name="Meet")

👉 Output:

Name: Meet

Age: 20

# 3. Default Arguments

* You can assign **default values** to parameters in the function definition.
* If the argument is not passed, the default value is used.

def greet(name="Guest"):

print("Hello,", name)

greet() # uses default → "Guest"

greet("Kapadiya") # overrides default

👉 Output:

Hello, Guest

Hello, Kapadiya

# 4. Mixing Arguments

You can combine **positional, keyword, and default arguments**, but order matters:

1. Positional arguments first
2. Then default arguments
3. Then keyword arguments

def profile(name, age=18, city="Rajkot"):

print(f"Name: {name}, Age: {age}, City: {city}")

# Positional + Default

profile("Meet")

# Name: Meet, Age: 18, City: Rajkot

# Positional + Keyword

profile("Kapadiya", city="Ahmedabad")

# Name: Kapadiya, Age: 18, City: Ahmedabad

⚠️ Wrong order causes errors:

profile(age=21, "Meet") # ❌ SyntaxError

# 5. Summary Table

| **Argument Type** | **Definition** | **Example Call** |
| --- | --- | --- |
| **Positional** | Assigned based on order | func(10, 20) |
| **Keyword** | Assigned using parameter name | func(b=20, a=10) |
| **Default** | Uses a default value if not provided | func(10) (if b=20 is default) |

**In short:**

* Use **positional** when order is clear.
* Use **keyword** when order is not important.
* Use **default** to provide fallback values.

### *Scope of variables in Python.*

# 1. What is Scope?

**Scope** defines the part of a program where a variable is accessible.  
In Python, we use the **LEGB Rule** to determine variable scope.

# 2. The LEGB Rule

Python looks for variables in this order:

1. **L → Local** → Inside the current function.
2. **E → Enclosing** → Variables in outer (nested) functions.
3. **G → Global** → Defined at the top level of a module.
4. **B → Built-in** → Predefined names in Python (len, print, etc.).

# 3. Local Scope

Variables defined **inside a function** are local and accessible only inside that function.

def my\_func():

x = 10 # local variable

print("Inside function:", x)

my\_func()

print("Outside function:", x) # ❌ Error: x is not defined

# 4. Global Scope

Variables defined **outside any function** are global and can be accessed anywhere in the file.

x = 50 # global variable

def my\_func():

print("Inside function:", x)

my\_func()

print("Outside function:", x)

👉 Output:

Inside function: 50

Outside function: 50

# 5. Enclosing (Nonlocal) Scope

For **nested functions**, inner functions can access variables from their enclosing function.

def outer():

x = "outer variable"

def inner():

print("Inner sees:", x) # enclosing scope

inner()

outer()

👉 Output:

Inner sees: outer variable

You can use **nonlocal** to modify enclosing variables:

def outer():

x = "outer"

def inner():

nonlocal x

x = "changed"

inner()

print("After inner:", x)

outer()

👉 Output:

After inner: changed

# 6. Global Keyword

Use **global** to modify a global variable inside a function.

x = 10

def change():

global x

x = 20

change()

print(x) # 20

Without global, Python would treat x as a local variable inside the function.

# 7. Built-in Scope

Python has built-in functions and keywords available everywhere.  
Example: print(), len(), min(), etc.

print(len([1, 2, 3])) # uses built-in len()

# 8. Summary

* **Local** → inside function
* **Enclosing** → outer (nested) function
* **Global** → top-level in script
* **Built-in** → provided by Python

### *Built-in methods for strings, lists, etc.*

# 1. String Methods

Strings are immutable, so methods return a **new string**.

text = " Hello World "

print(text.lower()) # " hello world "

print(text.upper()) # " HELLO WORLD "

print(text.strip()) # "Hello World" (removes spaces)

print(text.replace("World", "Python")) # " Hello Python "

print(text.split()) # ['Hello', 'World']

print("-".join(["a", "b", "c"])) # "a-b-c"

print(text.find("World")) # 8

print(text.startswith(" He")) # True

print(text.endswith("ld ")) # True

# 2. List Methods

Lists are mutable.

nums = [3, 1, 4, 1, 5]

nums.append(9) # [3, 1, 4, 1, 5, 9]

nums.insert(2, 7) # [3, 1, 7, 4, 1, 5, 9]

nums.remove(1) # removes first 1 → [3, 7, 4, 1, 5, 9]

nums.pop() # removes last element → [3, 7, 4, 1, 5]

nums.sort() # [1, 3, 4, 5, 7]

nums.reverse() # [7, 5, 4, 3, 1]

print(nums.count(1)) # 1 (count occurrences)

print(nums.index(5)) # 1st index of 5

# 3. Tuple Methods

Tuples are immutable, so very few methods exist:

t = (1, 2, 3, 2, 4)

print(t.count(2)) # 2

print(t.index(3)) # 2

# 4. Set Methods

Sets are unordered and don’t allow duplicates.

s1 = {1, 2, 3}

s2 = {3, 4, 5}

s1.add(6) # {1, 2, 3, 6}

s1.remove(2) # {1, 3, 6}

s1.discard(10) # no error if element not found

print(s1.union(s2)) # {1, 3, 4, 5, 6}

print(s1.intersection(s2)) # {3}

print(s1.difference(s2)) # {1, 6}

print(s1.issubset(s2)) # False

print(s1.issuperset(s2)) # False

# 5. Dictionary Methods

Dictionaries store key-value pairs.

student = {"name": "Meet", "age": 20, "city": "Rajkot"}

print(student.keys()) # dict\_keys(['name', 'age', 'city'])

print(student.values()) # dict\_values(['Meet', 20, 'Rajkot'])

print(student.items()) # dict\_items([('name', 'Meet'), ('age', 20), ('city', 'Rajkot')])

student.update({"age": 21})

print(student) # {'name': 'Meet', 'age': 21, 'city': 'Rajkot'}

print(student.get("city")) # Rajkot

print(student.pop("age")) # removes age → 21

student.clear() # {}

# 6. Common Built-in Functions (work on many types)

* len() → number of elements
* max(), min() → largest/smallest element
* sum() → sum of numbers
* sorted() → returns a new sorted list
* type() → data type

### Theory: 8 Control Statements (Break, Continue, Pass)

### *Understanding the role of break, continue, and pass in Python loops.*

# 1. break

* Used to **exit the loop immediately**, even if the condition is still true.
* Control jumps **out of the loop**.

for i in range(1, 6):

if i == 3:

break

print(i)

print("Loop ended")

👉 Output:

1

2

Loop ended

# 2. continue

* Skips the **current iteration** and moves to the **next one**.
* Loop does not exit; it just ignores the rest of that cycle.

for i in range(1, 6):

if i == 3:

continue

print(i)

print("Loop finished")

👉 Output:

1

2

4

5

Loop finished

(3 was skipped)

# 3. pass

* Does **nothing** → a placeholder statement.
* Useful when you need a syntactically valid block but don’t want any action.

for i in range(1, 6):

if i == 3:

pass # do nothing for now

print(i)

👉 Output:

1

2

3

4

5

# 4. Comparison Table

| **Keyword** | **Action** |
| --- | --- |
| **break** | Exits the loop completely |
| **continue** | Skips to the next iteration |
| **pass** | Does nothing (placeholder) |

# 5. Real-Life Analogy

* **break** → “Stop the game entirely.”
* **continue** → “Skip this turn, move to next turn.”
* **pass** → “Do nothing, but keep the place reserved.”

### Theory: 9 String Manipulation

### *Understanding how to access and manipulate strings*

# 1. Strings in Python

* A **string** is a sequence of characters enclosed in quotes (' ' or " ").
* Strings are **immutable** → you can’t change characters directly, but you can create new strings.

text = "Python Programming"

# 2. Accessing Strings

### (a) Indexing

* Strings are indexed → first character is 0.

print(text[0]) # 'P'

print(text[7]) # 'P'

print(text[-1]) # 'g' (last character, using negative index)

### (b) Slicing

* Extract parts of a string:

print(text[0:6]) # 'Python' (from index 0 to 5)

print(text[7:]) # 'Programming' (from index 7 to end)

print(text[:6]) # 'Python' (from start to index 5)

print(text[::2]) # 'Pto rgamn' (every 2nd char)

print(text[::-1]) # 'gnimmargorP nohtyP' (reverse string)

# 3. String Concatenation & Repetition

a = "Hello"

b = "World"

print(a + " " + b) # "Hello World"

print(a \* 3) # "HelloHelloHello"

# 4. String Methods (Manipulation)

s = " Python is Fun "

print(s.lower()) # " python is fun "

print(s.upper()) # " PYTHON IS FUN "

print(s.strip()) # "Python is Fun" (removes spaces)

print(s.replace("Fun", "Powerful")) # " Python is Powerful "

print(s.split()) # ['Python', 'is', 'Fun'] (splits by space)

print(".".join(["A", "B", "C"])) # "A.B.C"

# 5. Searching in Strings

s = "Python Programming"

print(s.find("Pro")) # 7 (index where substring starts)

print(s.count("m")) # 2 (count occurrences)

print(s.startswith("Py")) # True

print(s.endswith("ing")) # True

# 6. Checking String Properties

word = "Python123"

print(word.isalpha()) # False (has digits too)

print("Python".isalpha()) # True

print("123".isdigit()) # True

print("hello".islower()) # True

print("HELLO".isupper()) # True

# 7. String Formatting

### (a) f-strings (recommended):

name = "Meet"

age = 20

print(f"My name is {name}, I am {age} years old.")

### (b) .format():

print("My name is {}, I am {} years old.".format(name, age))

# 8. Immutability of Strings

s = "Hello"

# s[0] = "h" # ❌ Error (strings cannot be changed directly)

s = "h" + s[1:] # ✅ Create new string

print(s) # "hello"

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

# 1. Concatenation (Joining Strings)

Use the + operator to join strings.

a = "Hello"

b = "World"

result = a + " " + b

print(result)

👉 Output:

Hello World

# 2. Repetition

Use the \* operator to repeat a string.

word = "Hi! "

print(word \* 3)

👉 Output:

Hi! Hi! Hi!

# 3. String Methods

Python provides many built-in methods for strings (strings are immutable → methods return a new string).

### (a) Changing Case

text = "Python Programming"

print(text.upper()) # "PYTHON PROGRAMMING"

print(text.lower()) # "python programming"

print(text.title()) # "Python Programming"

print(text.capitalize()) # "Python programming"

### (b) Removing Spaces

s = " hello world "

print(s.strip()) # "hello world" (removes both ends)

print(s.lstrip()) # "hello world " (removes left spaces)

print(s.rstrip()) # " hello world" (removes right spaces)

### (c) Replace & Split

msg = "I like Java"

print(msg.replace("Java", "Python")) # "I like Python"

text = "apple,banana,cherry"

print(text.split(",")) # ['apple', 'banana', 'cherry']

### (d) Joining Strings

words = ["Python", "is", "fun"]

print(" ".join(words)) # "Python is fun"

### (e) Checking Strings

name = "Meet123"

print(name.isalpha()) # False (has numbers)

print("Meet".isalpha()) # True

print("123".isdigit()) # True

print("python".islower()) # True

print("PYTHON".isupper()) # True

# 4. Quick Example

s = " hello python "

print(s.upper()) # " HELLO PYTHON "

print(s.strip().capitalize()) # "Hello python"

print((s.strip() + "!!") \* 2) # "hello python!!hello python!!"

### *String slicing*

# 1. What is Slicing?

* **Slicing** means extracting a **portion of a string** using the format:

string[start:end:step]

* start → index to begin (inclusive).
* end → index to stop (exclusive).
* step → jump size (default = 1).

# 2. Basic Examples

text = "Python Programming"

print(text[0:6]) # "Python" (from 0 to 5)

print(text[7:18]) # "Programming" (from 7 to 17)

print(text[:6]) # "Python" (start defaults to 0)

print(text[7:]) # "Programming" (end defaults to length)

# 3. Negative Indexing

* Python allows **negative indices** (counting from the end).

print(text[-11:-1]) # "Programmin"

print(text[-11:]) # "Programming"

print(text[:-9]) # "Python"

# 4. Using Step

print(text[::2]) # "Pto rgamn" (every 2nd char)

print(text[::3]) # "Ph rgm" (every 3rd char)

print(text[::-1]) # "gnimmargorP nohtyP" (reverse string)

# 5. Real Examples

s = "HelloWorld"

print(s[0:5]) # "Hello"

print(s[5:]) # "World"

print(s[-5:]) # "World"

print(s[:-5]) # "Hello"

print(s[::-1]) # "dlroWolleH"

### Theory: 10 Advanced Python (map(), reduce(), filter(), Closures and Decorators)

### *How functional programming works in Python.*

# 1. What is Functional Programming?

Functional programming (FP) is a **programming paradigm** where you build programs by **composing functions** instead of focusing on changing state or using loops.

Key principles:

* Functions are **first-class citizens** (they can be passed around like variables).
* Avoid **changing state** (immutability).
* Focus on **what to do**, not **how to do it**.

# 2. Functions as First-Class Objects

In Python, functions can be assigned to variables, passed as arguments, or returned from other functions.

def greet(name):

return f"Hello, {name}!"

say\_hello = greet # assign function

print(say\_hello("Meet"))

👉 Output:

Hello, Meet!

# 3. Higher-Order Functions

Functions that take other functions as arguments or return them.

### Example: map(), filter(), reduce()

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

# map: apply function to each element

squares = list(map(lambda x: x\*\*2, nums))

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

# filter: select elements based on condition

evens = list(filter(lambda x: x % 2 == 0, nums))

print(evens) # [2, 4]

# reduce: apply rolling computation

from functools import reduce

product = reduce(lambda a, b: a \* b, nums)

print(product) # 120

# 4. Immutability in FP

Functional style avoids modifying data. Instead, create new values.

nums = [1, 2, 3]

new\_nums = list(map(lambda x: x+1, nums)) # does not change nums

print(nums) # [1, 2, 3]

print(new\_nums) # [2, 3, 4]

# 5. Recursion Instead of Loops

FP often prefers **recursion** instead of iterative loops.

def factorial(n):

if n == 0:

return 1

return n \* factorial(n-1)

print(factorial(5)) # 120

# 6. Pure Functions

A **pure function** depends only on its input and produces no side effects (like modifying global variables or printing).

# Pure function

def add(a, b):

return a + b

# 7. Built-in Tools for FP

* map(), filter(), reduce()
* lambda functions (anonymous functions)
* functools module (e.g., reduce, partial)
* itertools module (functional-style looping)

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

# 1. map()

* **Purpose**: Apply a function to **each element** of an iterable (like list, tuple).
* **Syntax**:
* map(function, iterable)
* Returns a **map object** (needs list() or tuple() to view).

Example:

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

squares = list(map(lambda x: x\*\*2, nums))

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

# 2. filter()

* **Purpose**: Select items from an iterable based on a condition.
* **Syntax**:
* filter(function, iterable)
* The function must return **True/False**.

✅ Example:

nums = [10, 15, 20, 25, 30]

evens = list(filter(lambda x: x % 2 == 0, nums))

print(evens) # [10, 20, 30]

# 3. reduce()

* **Purpose**: Apply a function **cumulatively** to the elements of an iterable, reducing it to a single value.
* **Syntax**:
* reduce(function, iterable)
* Comes from functools.

Example:

from functools import reduce

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

product = reduce(lambda a, b: a \* b, nums)

print(product) # 120 (1\*2\*3\*4\*5)

# 4. Combined Example

Let’s use all three on one dataset:

from functools import reduce

nums = [1, 2, 3, 4, 5, 6]

# Step 1: Square each number

squares = list(map(lambda x: x\*\*2, nums))

print("Squares:", squares) # [1, 4, 9, 16, 25, 36]

# Step 2: Keep only even squares

even\_squares = list(filter(lambda x: x % 2 == 0, squares))

print("Even Squares:", even\_squares) # [4, 16, 36]

# Step 3: Find the product of even squares

result = reduce(lambda a, b: a \* b, even\_squares)

print("Product:", result) # 2304

### *Introduction to closures and decorators.*

# 1. Closures in Python

### What is a Closure?

A **closure** is a function that:

1. Has an **inner function**.
2. The inner function refers to variables from the **outer function’s scope**.
3. The outer function returns the inner function.

Example:

def outer\_function(msg):

def inner\_function():

print("Message:", msg) # inner uses outer variable

return inner\_function

closure = outer\_function("Hello Closure!")

closure() # Message: Hello Closure!

Here:

* msg is from the **outer scope**, but still accessible by inner\_function.
* This is a **closure**.

Uses:

* Encapsulation
* Function factories
* Maintaining state without classes

# 2. Decorators in Python

### What is a Decorator?

A **decorator** is a special function that **takes another function as input and extends/modifies its behavior without changing its code**.

It uses closures internally!

Basic Example:

def my\_decorator(func):

def wrapper():

print("Before function runs")

func()

print("After function runs")

return wrapper

@my\_decorator

def say\_hello():

print("Hello!")

say\_hello()

👉 Output:

Before function runs

Hello!

After function runs

* @my\_decorator is shorthand for say\_hello = my\_decorator(say\_hello).

# 3. Decorators with Arguments

def repeat(n):

def decorator(func):

def wrapper(\*args, \*\*kwargs):

for \_ in range(n):

func(\*args, \*\*kwargs)

return wrapper

return decorator

@repeat(3)

def greet(name):

print("Hello,", name)

greet("Meet")

👉 Output:

Hello, Meet

Hello, Meet

Hello, Meet

# 4. Built-in Decorators

Python has many built-in decorators:

* @staticmethod
* @classmethod
* @property
* @functools.lru\_cache (for caching results)