# **PYTHON**

## **What is Programming ?**

Programming is the process of **writing instructions that a computer can understand and execute** to perform specific tasks.

1. A **computer** doesn’t understand human language directly.
2. We use **programming languages** (like Python, C, Java, JavaScript, etc.) to give step-by-step instructions.
3. These instructions are called a **program**, and when run, the computer follows them to solve problems, automate tasks, or create applications.

## **What is Python ?**

Python is a popular programming language. It was created by Guido van Rossum, and released in 1991.

**It is used for:**

* web development (server-side)
* software development
* mathematics
* system scripting

## **Syntax in Python**

Python syntax refers to the set of rules that define how Python code must be written and structured so that the interpreter can understand and execute it. It acts like grammar in a language, ensuring code is valid and meaningful.

**Key Rules of Python Syntax**

* **Indentation:** Mandatory; used to define code blocks instead of {}.
* **Case-sensitive:** Identifiers like name, Name, and NAME are treated as different.
* **Statements end with newline**: Each new line represents the end of a statement (semicolon not required).
* **Comments:** Written using #.
* **Variables:** No need for type declaration; types are assigned automatically.
* **Functions & Classes:** Defined using def and class keywords.
* **Keywords:** Must be used exactly as defined (e.g., if, else, while).

## **Data Types**

Data types in Python specify the kind of values a variable can hold and determine what operations can be performed on that data. Python is dynamically typed, meaning you don’t need to declare the type explicitly; it is assigned automatically.

* **Numeric Types**
  1. int: Whole numbers (e.g., 10, -5)
  2. float: Decimal numbers (e.g., 3.14, -2.5)
  3. complex: Numbers with real and imaginary parts (e.g., 2 + 3j)
* **Text Type** 
  1. str: String (sequence of characters, e.g., "Hello")
* **Sequence Types**
  1. list: Ordered, mutable collection (e.g., [1, 2, 3])
  2. tuple: Ordered, immutable collection (e.g., (1, 2, 3))
  3. range: Sequence of numbers, often used in loops (e.g., range(5))
* **Set Types**
  1. set: Unordered, unique elements (e.g., {1, 2, 3})
  2. frozenset: Immutable version of set
* **Mapping Type**
  1. dict: Key-value pairs (e.g., {"name": "Alice", "age": 20})
* **Boolean Type**
  1. bool: True or False
* **Binary Types**
  1. bytes: Immutable sequence of bytes
  2. bytearray: Mutable sequence of bytes
  3. memoryview: View object for memory access

## **Conditional Statements**

Conditional statements in Python are used to make decisions in a program. They allow code to execute different blocks based on whether a condition is **True** or **False**

* **if statement**: Executes a block if the condition is true.

*age = 18*

*if age >= 18:*

*print("You are an adult")*

* **if-else statement**: Executes one block if condition is true, another if false.

*num = 5*

*if num % 2 == 0:*

*print("Even")*

*else:*

*print("Odd")*

* **if-elif-else statement**: Checks multiple conditions in sequence.

*marks = 75*

*if marks >= 90:*

*print("Grade A")*

*elif marks >= 75:*

*print("Grade B")*

*else:*

*print("Grade C")*

* **Nested if statement**: if inside another if.

*x = 10*

*if x > 0:*

*if x % 2 == 0:*

*print("Positive Even")*

## **Loops**

Loops in Python are control structures that let you **execute a block of code multiple times**, either for a fixed number of iterations or until a condition is met.

* **for Loop :** Iterates over sequences like lists, tuples, strings, or ranges.

*for i in range(5):*

*print(i)*

* **while Loop :** Repeats code as long as a condition is True.

*count = 0*

*while count < 5:*

*print(count)*

*count += 1*

**Loop Control Statements**

* **break:** Exit the loop completely.

*for i in range(5):*

*if i == 3:*

*break*

*print(i) # Output: 0 1 2*

* **continue** : Skip the current iteration, go to next.

*for i in range(5):*

*if i == 2:*

*continue*

*print(i) # Output: 0 1 3 4*

## **Functions**

A function in Python is a reusable block of code that performs a specific task. Functions help organize code, avoid repetition, and make programs easier to read and maintain.

**Types of Functions**

* **Built-in Functions:** Already provided by Python (e.g., print(), len(), type()).
* **User-defined Functions:** Created by the programmer using the def keyword.

**Syntax of a Function**

* **Function with Parameters**

Parameters allow you to pass values into a function.

*def greet(name):*

*print("Hello,", name)*

*greet("Pranjal") # Output: Hello, Pranjal*

* **Function with Return Value**

A function can return a value using the return statement.

*def add(a, b):*

*return a + b*

*result = add(5, 3)*

*print(result) # Output: 8*

* **Function with Default Arguments**

Parameters can have default values if no argument is provided**.**

*def greet(name="Guest"):*

*print("Hello,", name)*

*greet() # Output: Hello, Guest*

*greet("Alice") # Output: Hello, Alice*

* **Function with \*args (Arbitrary Positional Arguments)**

Allows passing multiple arguments without specifying theexact number.

Inside the function, \*args is treated as a **tuple**.

*def add\_all(\*args):*

*return sum(args)*

*print(add\_all(1, 2, 3, 4)) # Output: 10*

* **Function with \*\*kwargs (Arbitrary Keyword Arguments)**

Allows passing multiple key=value pairs**.**

Inside the function, \*\*kwargs is treated as a **dictionary**.

*def show\_details(\*\*kwargs):*

*for key, value in kwargs.items():*

*print(key, ":", value)*

*show\_details(name="Pranjal", age=2o, city="Lonar")*

## **Exception Handling**

Exception handling in Python is a mechanism to **handle runtime errors** gracefully without stopping the entire program. It ensures that the program can continue or fail safely when unexpected situations occur.

**Keywords Used in Exception Handling**

* **try :** Block of code that may raise an error.
* **except :** Block of code to handle the error.
* **finally :** Executes always (cleanup actions).
* **else :** Executes if no exception occurs.

*try:*

*num = int("20")*

*except ValueError:*

*print("Invalid conversion")*

*else:*

*print("Conversion successful") # Runs if no error*

*finally:*

*print("Execution finished") # Always runs*

## **Decorators**

A **decorator** in Python is a special function that modifies or extends the behavior of another function or method **without changing its code**. Decorators are often used for logging, authentication, performance measurement, and more.

*def decorator(func):*

*def wrapper():*

*print("Welcome")*

*func()*

*print("Thanks for using this Function")*

*return wrapper*

*@decorator*

*def say\_hello():*

*print("Hello!")*

*say\_hello()*

## **OOP**

Object-Oriented Programming (OOP) is a programming paradigm that organizes code into **objects** containing both data (**attributes**) and behavior (**methods**). Python fully supports OOP, making code more reusable, modular, and easier to maintain.

**Core OOP Concepts**

* **Class**

A blueprint for creating objects (defines attributes and methods).

* **Object**

An instance of a class (real-world entity created from a class).

* **Encapsulation**

Binding data (attributes) and methods together, controlling access.

Achieved using access specifiers: **Public**, **Protected**, **Private**

* **Inheritance**

A class child can inherit properties from another class (parent).

* **Polymorphism**

Same function/method behaves differently depending on the object.

* **Abstraction**

Hiding implementation details and showing only the necessary features.

## **List Comprehension**

List comprehension provides a concise way to create lists in a single line, using a loop and optional condition inside square brackets.

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

*print(squares) # Output: [0, 1, 4, 9, 16]*

## **Dictionary Comprehension**

Dictionary comprehension provides a **concise way to create dictionaries** in a single line, using a loop and optional condition inside curly braces.

*squares\_dict = {x: x\*\*2 for x in range(5)}*

*print(squares\_dict) # Output: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}*

## **Iterators**

An **iterator** is an object that allows sequential access to elements of a collection (like lists, tuples, strings) without using indexing. It implements two methods:

* \_\_iter\_\_() → returns the iterator object itself
* \_\_next\_\_() → returns the next value, raises StopIteration when no items left

*my\_list = [1, 2, 3]*

*it = iter(my\_list) # Get iterator*

*print(next(it)) # Output: 1*

*print(next(it)) # Output: 2*

*print(next(it)) # Output: 3*

## **Generators**

A **generator** is a special type of iterator that is created using a **function with yield** instead of return. Generators are memory-efficient because they produce values **one at a time**

*def count\_up\_to(n):*

*i = 1*

*while i <= n:*

*yield i*

*i += 1*

*gen = count\_up\_to(3)*

*print(next(gen)) # Output: 1*

*print(next(gen)) # Output: 2*

*print(next(gen)) # Output: 3*

## **Virtual Environment**

A virtual environment is an isolated environment that allows you to install and manage Python packages separately for each project, without affecting the system-wide Python installation.

**Why use it?**

* Avoids conflicts between different project dependencies.
* Keeps projects independent.
* Ensures reproducibility of code.

## **pip**

pip stands for **"Pip Installs Packages"**. It is the default **package manager** in Python used to install, upgrade, and manage external libraries.

## **Standard libraries**

The **Python Standard Library** is a collection of built-in modules and packages that come pre-installed with Python. It provides **ready-to-use functions and tools** for tasks like file handling, math, system operations, networking, data manipulation, and more — without needing to install external packages.

**Commonly Used Standard Libraries**

* **math :** Mathematical functions

*import math*

*print(math.sqrt(16)) # 4.0*

*print(math.pi)*

* **random :** Generate random numbers

*import random*

*print(random.randint(1, 10))*

* **datetime**: Work with dates and times

*import datetime*

*today = datetime.date.today()*

*print(today)*

* **os** **:** Interact with the operating system

*import os*

*print(os.getcwd())*

* **json** **:** Work with JSON data

*import json*

*data = '{"name":"Pranjal","age":21}'*

*parsed = json.loads(data)*

*print(parsed["name"])*

* **collections :** Specialized data structures

*from collections import Counter*

*print(Counter("banana"))*

* **statistics** **:** Statistical calculations

*import statistics*

*data = [10, 20, 30, 40]*

*print(statistics.mean(data))*

## **Coding Stadandars**

**Coding standards** are a set of rules and best practices that define how code should be written, organized, and documented to ensure consistency, readability, maintainability, and quality across a project or team.

* **Naming Conventions** 
  1. **Variables and Functions:** Names should be written in snake\_case (all lowercase with words separated by underscores).

Example: total\_amount, calculate\_sum.

* 1. **Classes and Exceptions:** Class names should follow PascalCase (each word starts with a capital letter, no underscores).

Example: StudentRecord, PaymentProcessor, FileNotFoundError.

* 1. **Constants:** Constants should be written in UPPERCASE letters, with words separated by underscores.

Example: PI, MAX\_LIMIT, DEFAULT\_TIMEOUT.

* 1. **Private and Protected Members:** Members meant for internal use should start with an underscore (\_). This convention warns developers not to access them directly.

If stronger privacy is needed, use double underscores (\_\_), which invoke name mangling.

Example: \_balance (protected), \_\_password (private).

* 1. **Modules and Packages:** File names and package names should be in lowercase. Underscores can be used if the name is long or contains multiple words.

Example: math\_utils, stringtools.

* **Docstrings & Comments**

**Docstring:** Describes what a function, class, or module does.

Written using triple quotes """ ... """.

Example:

*def add(a, b):*

*"""Return the sum of two numbers a and b."""*

*return a + b*

**Inline comments:** Explain non-obvious code logic.

Use # before comment, keep them short and meaningful.

Example:

*result = factorial(n) # Recursive call*

* **Type of Testing**

1. **Unit Testing:** Tests small pieces (functions, classes) individually.
2. **Integration Testing:** Tests how modules work together.
3. **System Testing:** Tests the entire application end-to-end.
4. **Acceptance Testing:** Validates against business requirements.
5. **Regression Testing:** Ensures new changes don’t break existing features.
6. **Smoke & Sanity Testing:** Quick checks before detailed testing.

* **PEP 8 (Python Enhancement Proposal 8)**

1. **Indentation:** 4 spaces (no tabs).
2. **Line length:** max 79 characters.
3. **Imports:** standard - third-party - local.
4. **Spaces around operators:** a = b + c not a=b+c.
5. Readable variable names, avoid single letters (except in loops).

* **SOLID Principles**

**S – Single Responsibility:** One class = one responsibility.

**O – Open/Closed:** Open for extension, closed for modification.

**L – Liskov Substitution:** Subclasses should replace parent classes without breaking code.

**I – Interface Segregation:** Many small interfaces better than one large one.

**D – Dependency Inversion:** Depend on abstractions, not concrete implementations.

* **DRY Principle (Don’t Repeat Yourself)**

Avoid duplicating code; use functions, classes, or modules.

Example: Instead of writing the same calculation in multiple places → put it in one function and call it.