1. Software

Software is a collection of programs, data, and instructions that tell a computer how to perform a task. Without software, a computer or mobile device is just hardware that cannot function. Software acts as a bridge between the user and the hardware.

Real-Life Examples:

- **System Software:** Android in your mobile, Windows in your PC → like the “manager” that controls all other apps.
- **Application Software:** WhatsApp, YouTube, PUBG, MS Word → apps you use daily to chat, watch videos, play games, or write documents.
  👉 Without software, your mobile is just like a box of plastic and circuits — it becomes “smart” only because of software.

---

2. Programming

Programming is the process of writing instructions (called code) for a computer to follow and complete a task. A program is a sequence of steps written in a logical order, just like a recipe for cooking or step-by-step instructions given to a robot.

🔶 **Why do we code?**
We code to make computers do work for us — from simple tasks like adding numbers to complex tasks like running social media, online shopping, and artificial intelligence.

🔶 **When do we code?**
We write code when we want to **solve a problem**, **automate a process**, or **create new applications** like websites, games, or apps.

🔶 **Use of coding (with examples):**

- **Solving problems:** Google Maps calculates the fastest route (coding).
- **Building apps & websites:** YouTube, Amazon, Swiggy — all powered by code.
- **Controlling machines:** ATMs give you money, washing machines follow programs, self-driving cars use sensors + code.
- **Analyzing data:** Netflix recommends movies, Instagram suggests reels using AI — all with coding.

☞ **Simple Example for Students:** Imagine you want to send 1000 “Happy Birthday” messages on WhatsApp. Doing it manually takes hours. With coding, you can write a program that sends them automatically in seconds.

---

3. Programming Language

A programming language is a special language used to write instructions that a computer can understand and execute. Since computers only understand **0s and 1s (binary)**, programming languages act as a **translator** between humans and computers.

Real-Life Example:

- If you visit Japan and only know English, you need a translator to talk to locals.
- Similarly, programming languages (like Python, Java, C) translate our human-like instructions into binary so the computer can understand.

---

4. Syntax

Syntax in programming means the **set of rules** that define how code must be written in a programming language. Just like grammar in English, syntax ensures that the computer can understand your instructions correctly.

Real-Life Example:

- In English, you must write *“I am eating food”* (✔).
- If you write *“Food eating am I”* (✕), people get confused.
- In the same way, wrong syntax confuses the computer.

Python Example:

```
print("Hello")  # ✔ Correct syntax

Print("Hello")  # ✕ Error (Python is case-sensitive)
```

☞ If syntax is wrong, the computer will stop and show an error — just like a teacher correcting bad grammar.

💠 Low-Level and High-Level Languages

2. Low-Level Languages

📖 **Definition:**
Low-level languages are programming languages that are **close to machine code** (binary language). They are **hard for humans to understand** but **fast for computers** to execute.

Types of Low-Level Languages:

1. **Machine Language (1st Generation Language)**
   - Written only in 0s and 1s (binary).
   - Directly understood by the computer.
   - Example: 10110000 01100001
   - Very difficult for humans.
2. **Assembly Language (2nd Generation Language)**
   - Uses short codes called **mnemonics** instead of binary.
   - Example: MOV A, 5 (move value 5 into register A).
   - Easier than machine language but still not beginner-friendly.
   - Needs an **assembler** to convert into machine code.

✅ Advantages of Low-Level Languages

- Very fast and efficient (directly controls hardware).
- Best for **system programming** (like operating systems, device drivers).

❌ Disadvantages of Low-Level Languages

- Hard to learn and understand.
- Code is not portable (machine-dependent).
- Writing long programs is very difficult.

👉 Real-Life Example:

- Low-level is like **talking directly in signals/morse code** with a machine.
- Only a few experts can do it, and it takes a lot of effort.

3. High-Level Languages

📖 **Definition:**
High-level languages are programming languages that are **closer to human language (English-like syntax)**. They are **easy for humans to write and understand**, but need a **compiler or interpreter** to convert into machine code.

Examples of High-Level Languages:

- Python
- Java
- C
- C++
- JavaScript
- Ruby

✅ **Advantages of High-Level Languages**

- Easy to learn and use (looks like English).
- Portable (can run on different machines).
- Reduces development time.
- Large libraries and support.

❌ **Disadvantages of High-Level Languages**

- Slower than low-level languages (because of translation).
- Less control over hardware.

👉 **Real-Life Example:**

- High-level is like **talking to your friend in English**.
- Both can easily understand, and communication is quick.

4. Code Comparison Example

Machine Language (Low-Level):

```
10110000 01100001
```

👉 Not readable for humans.

Assembly Language (Low-Level):

```
MOV A, 5

ADD A, B
```

👉 Still hard for beginners.

Python (High-Level):

```
a = 5

b = 10

print(a + b)
```

👉 Easy to understand → "Take 5, add 10, print result."

� Compiler vs Interpreter

- **Compiler** → Translates entire program at once → Fast execution.
  - Example: C, C++, and Java (with JIT) mainly use compilers.
- **Interpreter** → Translates program line by line → Easy debugging.
- It doesn't create a separate executable file.
  - Example: Python, JavaScript, Ruby are interpreter-based languages.

� Object-Oriented Programming (OOP)

Definition:

- **Object-Oriented Programming** is a way of programming where we organize code around **objects** (real-world things) instead of just functions and logic.
- Objects have **properties (data)** and **behaviors (methods/functions)**.

👉 **Example in Real Life:**
Think about a **Car**:

- Properties → color, brand, speed
- Behaviors → start(), stop(), accelerate()

In OOP, we model real-world things in the same way.

5. Python

Python is a **high-level, interpreted, object-oriented programming language** that is simple and powerful. It uses English-like commands, making it easier for beginners to learn. Python requires fewer lines of code compared to other languages, which means you can write programs faster and more clearly.

Why Python? (with real-life touch):

- **Easy to learn:** Looks almost like English, so beginners pick it up fast.
- **Less code:** What takes 5–6 lines in Java takes 1–2 lines in Python → saves time.
- **Powerful:** Used in Google (search engine), Netflix (movie recommendations), Instagram (filters), YouTube (video suggestions), Tesla (self-driving cars).
- **Beginner-friendly:** That's why it is chosen as the **first language** for students.

Hello World Example:
👉 In Python:

```
print("Hello, World!")
```

👉 In Java:

```
public class Main {

   public static void main(String[] args) {

      System.out.println("Hello, World!");

   }

}
```

Clearly, Python is **shorter, cleaner, and friendlier**

3. Why Python?

1. Easy to learn for beginners.

2. Code looks like English.
3. Can be used in almost every field.
4. Works on all platforms (Windows, Mac, Linux).
5. Has huge community support.

☞ **Real-Life Example:**

- If coding languages were vehicles:
    - **C language** = Bicycle (basic, needs more effort).
    - **Java** = Car (powerful but requires more setup).
    - **Python** = Bike or Scooter (easy, fast, and anyone can learn quickly).

4. Features of Python

- Simple and readable.
- Free and open source.
- Interpreted (runs line by line).
- Portable (runs everywhere).
- Object-oriented.
- Huge standard library.
- Supports GUI, AI, ML, Web, Games.

5. Where is Python Used?

Python is **everywhere** in the modern world:

1. **Artificial Intelligence (AI)** → Virtual assistants like Siri, Alexa, ChatGPT.
2. **Web Development** → Websites like Instagram, YouTube (back-end built using Python).
3. **Data Science** → Used by companies like Netflix, Amazon to recommend movies/products.
4. **Automation** → Auto-fill forms, rename 1000 files at once, generate reports.
5. **Cyber Security** → Ethical hackers use Python scripts to test systems.
6. **Game Development** → Pygame for making simple games.

7. **Robotics & IoT** → Controlling devices and machines.

👉 **Real-Life Example:**

- Netflix uses Python to recommend movies to you.
- Google uses Python in search engines.
- NASA uses Python for scientific research.

6. What Python Can Do (Better than Other Languages)

- Easier syntax (code looks like English).
- Fewer lines of code needed.
- Large number of ready-to-use libraries.
- Faster development speed.

7. Python Overview

- File extension: .py
- Python interactive prompt: >>>
- Run Python program:
  - Write code in program.py
  - Run → python program.py

▯ What is an Algorithm?

👉 An **Algorithm** is just a **step-by-step plan** to solve a problem.
Like a **recipe** for cooking food.

Example: If you want to make **Maggi** 🍜
Algorithm would be:

1. Take water in a pan.
2. Boil the water.
3. Add Maggi noodles.
4. Add masala packet.
5. Cook for 2 minutes.

6. Eat and enjoy.

✅ This is an algorithm — simple steps to solve a problem.

▯ What is a Flowchart?

👉 A **Flowchart** is like a **picture** of an algorithm.
Instead of writing steps, we **draw shapes and arrows** to show what happens first, next, and last.

📌 Flowchart uses some shapes:

- ■ **Oval (ellipse)** → Start/End
- □ **Rectangle** → Process / Step (like "Boil water")
- ◈ **Diamond** → Decision (Yes/No questions)
- ➡ **Arrows** → Show the flow

First Python Program

```
print("Hello, World!")
```

👉 Output:

```
Hello, World!
```

📌 **Explain to Students:**

- print() displays output on the screen.
- Text inside " " is called a **string**.

Comments in Python

- Comments = Notes in code (ignored by Python).

Single-line Comment:

```
# This is a single-line comment
```

Multi-line Comment:

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

☞ **Real-Life Example:**

- Comments are like **sticky notes in your textbook** – useful for you, but ignored in exams.

□ Input and Output in Python

☞ First, let’s understand:

- **Input** means **giving something to the computer**.
- **Output** means **computer giving something back to us**.

Think of it like **talking with your friend**:

- You ask your friend: *“What’s your name?”* → That is **Input**.
- Your friend replies: *“My name is Ravi.”* → That is **Output**.

So, **input is like asking** and **output is like answering**.

🔹 Example 1: Output

```
print("My name is Nandan")
```

☞ Here, we are telling the computer to **say something**.
The computer will answer:

```
My name is Nandan
```

So print() is used when **we want the computer to talk to us**.

❖ **Example 2: Input**

```
name = input("Enter your name: ")

print("Hello", name)
```

☞ Here, first the computer **asks us a question**:

```
Enter your name:
```

If we type:

```
Ravi
```

The computer will reply:

```
Hello Ravi
```

So, input() is used when **we want to talk to the computer and give it something**.

Input and Output

❖ **Output**

```
print("My name is Nandan")
```

☞ Output:

```
My name is Nandan
```

❖ **Input**

```
name = input("Enter your name: ")

print("Hello", name)
```

☞ Example Run:

```
Enter your name: Ravi

Hello Ravi
```

☞ **Real-Life Example:**
When you log in to Instagram and type your username, that is **input**.
When Instagram welcomes you, that is **output**.

🔷 1. What are Tokens?

👉 In Python, **tokens** are the smallest building blocks of a program.
They are like **words** in English. Just as sentences are formed by combining words, Python programs are formed by combining tokens.

📌 **Definition:**
A token is the smallest individual unit in a Python program that has a meaningful role in the execution of code.

👉 Example:

x = 10 + 5

Here:

- x → Identifier
- = → Operator
- 10, 5 → Constants
- + → Operator

So, this one line contains **tokens**!

📌 **Types of Tokens in Python:**

1. Keywords
2. Identifiers
3. Constants
4. Variables
5. Operators

🔷Keywords

👉 Keywords are **reserved words** in Python.
They have a predefined meaning and **cannot** be used as identifiers.

📌 Examples of Keywords:
if, else, while, for, break, continue, def, class, return, import, try, except, finally, with, lambda, is, in, global, nonlocal, assert, pass.

👉 Python has **35 keywords** (in Python 3.10+).

📌 Example Code:

```
# ❌ Wrong: Using keyword as a variable

if = 5     # Error

for = 10   # Error

# ✅ Correct

age = 5

count = 10
```

💠 Identifiers

👉 Identifiers are **names** given to variables, functions, classes, or objects.
They help us **identify** different parts of the program.

📌 **Rules for Identifiers:**

1. Must start with a letter (a-z, A-Z) or underscore (_).
2. Can contain letters, digits, and underscores.
3. Cannot start with a digit.
4. Case-sensitive (Age and age are different).
5. Cannot use keywords as identifiers.

📌 Example Code:

```
student_name = "John"  # ✅ Valid

roll123 = 45         # ✅ Valid

_number = 50         # ✅ Valid

2marks = 90          # ❌ Invalid (cannot start with number)

for = "test"         # ❌ Invalid (keyword)
```

◆ **Constants**

👉 A **constant** is a value that does **not change** during program execution.
Python does not have a special keyword like const (as in C++/Java), but by convention, we use **capital letters** for constants.

📌 Example:

```
PI = 3.14159

MAX_STUDENTS = 60

print(PI, MAX_STUDENTS)
```

👉 Even though we can reassign values, we **shouldn't change constants**.

◆ **Variables**

👉 Variables are **containers** that store data values.
Unlike constants, their values can **change** during execution.

📌 Example Code:

```
name = "Alice"

age = 21

height = 5.6


print("Name:", name)

print("Age:", age)

print("Height:", height)
```

7. Valid and Invalid Variable Names

Rules :

- Must start with a **letter** or **underscore** (_).
- Cannot start with a **digit**.
- Can only contain letters, digits, and underscores.

- Are **case-sensitive**.
- Cannot be a Python **keyword**.

Examples:

```
myvar = "John"     # valid
my_var = "John"    # valid
_my_var = "John"   # valid
myVar = "John"     # valid
MYVAR = "John"     # valid
myvar2 = "John"    # valid

2myvar = "John"    # invalid (starts with digit)
my-var = "John"    # invalid (contains dash)
my var = "John"    # invalid (contains space)
```

---

9. Assigning Multiple Variables

- You can assign multiple variables in one line:
- x, y, z = "Orange", "Banana", "Cherry"
- print(x, y, z)

Or assign the same value to multiple variables:

- x = y = z = "Orange"
- print(x, y, z)

Unpacking a list into variables:

- fruits = ["apple", "banana", "cherry"]
- x, y, z = fruits

print(x, y, z)

Data types

Basic Data Types

Data types classify the type of value a variable can hold. Python automatically assigns a data type based on the value you give it.

int: Represents integer numbers (whole numbers, positive or negative) without a decimal point. Examples: 10, -500, 0.

float: Represents real numbers with a decimal point. Examples: 3.14, -0.01, 2.0.

boolean (bool): Represents one of two values: True or False. Used for logical operations. Examples: is_valid = True, is_empty = False.

✅ **Key Points**

Booleans always start with capital T (True) and capital F (False).

- ❌ true or false → Error
- ✅ True or False → Correct

Booleans are the result of comparison operators:

```
print(10 > 5)    # True
print(10 == 5)   # False
print(5 < 3)     # False
```

In Python, many values have a Boolean equivalent (truthy / falsy):

- **Falsy values:** 0, "" (empty string), [] (empty list), None → False
- **Truthy values:** Any non-empty value or non-zero number → True

```
print(bool(0))       # False
print(bool(""))      # False
print(bool("hello"))  # True
print(bool(25))      # True
```

complex: Represents complex numbers, which have a real part and an imaginary part, written with a j or J.

💠 What is a Complex Number?

- A complex number is a number that has two parts:
    - Real part → a normal integer or float
    - Imaginary part → represented with j (in Python, not *i* like in mathematics).

📌 Syntax:

```
z = a + bj
```

Where:

- a → real part (int or float)
- b → imaginary part (int or float)
- j → imaginary unit

💠 Accessing Real and Imaginary Parts

Python provides attributes to get the real and imaginary parts:

```
z = 7 + 5j

print(z.real)   # 7.0

print(z.imag)   # 5.0
```

---

string (str): Represents a sequence of characters enclosed in single quotes ('...') or double quotes ("..."). You can perform various operations on strings, such as concatenation (joining them together) with the + operator, and slicing (extracting a part of the string). Examples: greeting = "Hello", full_name = "John Doe".

---

3. Casting (Changing Data Types)

To force a variable into a certain data type, you use casting functions:

- x = str(3)   # x is now "3"

- y = int(3)   # y is now 3
- z = float(3) # z is now 3.0

Task: Casting Practice

- Take number "7" as input. Convert it to int and float. Show results with type().

---

4. Getting the Type of a Variable(Type Check)

Use the type() function to check a variable’s data type:

- x = 5
- y = "John"
- print(type(x)) # <class 'int'>
- print(type(y)) # <class 'str'>

Task: Type Check

- Assign x = 10, then x = "Ten". Use print(type(x)) after each assignment.

---

5. Single or Double Quotes for Strings

- Both single ' ' and double " " quotes work for string variables:
- x = "John"
- x = 'John'

These two are identical.

Task: Quote Variations

- Create two string variables: one with single quotes, one with double quotes. Print both.

📘 Operators in Python

🔹 1. What are Operators?

- Operators are **special symbols** in Python that are used to perform operations on values and variables. They tell the computer what kind of task we want to do, such as adding numbers, comparing values, checking conditions, or working with data in memory.
- Operators make programming easier because they allow us to write simple expressions instead of long statements.
- Example: +, -, *, /, ==, and, etc.
- **Operands** → the values on which operators work.

👉 Example:

```
x = 10

y = 5

print(x + y)  # 15   → '+' is operator, x & y are operands
```

🔹 2. Types of Operators in Python

Python has many operators, grouped into categories:

1. **Arithmetic Operators**
2. **Comparison (Relational) Operators**
3. **Logical Operators**
4. **Assignment Operators**
5. **Bitwise Operators**
6. **Identity Operators**
7. **Membership Operators**

✨ 2.1 Arithmetic Operators

Arithmetic operators are used to perform **basic mathematical calculations** such as addition, subtraction, multiplication, division, etc. They allow us to work with numbers in the same way we do in mathematics. These operators are mostly used in solving equations, calculations, and performing numerical operations in a program.

Operator	Meaning	Example	Output
+	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
**	Exponent (power)	2 ** 3	8

👉 Example:

```
a = 9

b = 4

print(a + b)   # 13

print(a - b)   # 5

print(a * b)   # 36

print(a / b)   # 2.25

print(a // b)  # 2

print(a % b)   # 1

print(a ** b)  # 6561
```

✨ 2.2 Comparison (Relational) Operators

Comparison operators are used to compare two values. They check the relationship between the values and always return a result in the form of True or False. These operators are useful when making decisions in programs, like checking if two numbers are equal or finding out which number is bigger or smaller.

Used to compare values → returns **True** or **False**.

Operator	Meaning	Example	Output
==	Equal to	5 == 5	True
!=	Not equal to	5 != 3	True
>	Greater than	5 > 3	True
<	Less than	5 < 3	False
>=	Greater than or equal	5 >= 5	True
<=	Less than or equal	3 <= 5	True

👉 Example:

```
x = 10

y = 20

print(x == y)  # False

print(x != y)  # True

print(x > y)   # False

print(x < y)   # True

print(x >= 10) # True

print(y <= 15) # False
```

✨ 2.3 Logical Operators

Logical operators are used to **combine multiple conditions** or statements and decide whether the overall result is **True or False**. They help in building complex decision-making expressions. Logical operators are mostly used in conditional statements and loops where we need to check more than one condition at a time.

Operator	Meaning	Example	Output
and	True if **both** conditions true	(5 > 2) and (4 > 1)	True
or	True if **at least one** true	(5 > 2) or (4 < 1)	True
not	Negates (reverse) condition	not(5 > 2)	False

👉 Example:

```
a = 5

b = 10

print(a > 2 and b > 5)  # True

print(a > 10 or b > 5)  # True

print(not(a > 2))       # False
```

✨ 2.4 Assignment Operators

Assignment operators are used to **assign values to variables**. They can also perform mathematical operations while assigning the value. For example, instead of writing a separate equation, we can use assignment operators to update the value of a variable quickly. These operators make programs shorter and easier to write.

Operator	Example	Equivalent
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3

Operator	Example	Equivalent
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
//=	x //= 3	x = x // 3
%=	x %= 3	x = x % 3
**=	x **= 3	x = x ** 3

☞ Example:

```
x = 10
x += 5   # 15
x *= 2   # 30
print(x)
```

✨ 2.5 Bitwise Operators

Bitwise operators are used to perform operations at the **binary (bit) level** of data. Every number in the computer is stored in binary (0s and 1s), and bitwise operators allow us to work directly with those binary digits. These operators are very powerful and are often used in low-level programming, networking, and performance-based tasks.

Operator	Meaning	Example (binary)	Result
&	AND	5 & 3 → 101 & 011	1
|	OR	5 | 3 -> 101 | 011	7
^	XOR	5 ^ 3 → 101 ^ 011	6
~	NOT	~5 → -(5+1)	-6
<<	Left shift	5 << 1	10
>>	Right shift	5 >> 1	2

☞ Example:

```
a = 5  # 101
b = 3  # 011


print(a & b)  # 1
print(a | b)  # 7
print(a ^ b)  # 6
print(~a)     # -6
print(a << 1) # 10
print(a >> 1) # 2
```

✨ 2.6 Identity Operators

Identity operators are used to **check whether two objects are the same in memory**. They do not compare the values of the objects, but instead check if both variables point to the **same memory location**. This is useful when we want to know if two variables are exactly the same object or not.

Operator	Meaning	Example	Output
is	True if both point to same object	x is y	True/False
is not	True if not same object	x is not y	True/False

👉 Example:

```
x = [1,2,3]
y = [1,2,3]
z = x
print(x is z)     # True (same object)
print(x is y)     # False (different objects, same values)
print(x == y)     # True (values are equal)
print(x is not y)  # True
```

✨ 2.7 Membership Operators

Membership operators are used to check whether a value exists inside a sequence such as a list, string, or tuple. Instead of searching through the data manually, we can use these operators to directly test if an element is present or not. This makes checking for items in collections much easier.

Used to check if a value is present in a sequence (list, string, tuple).

Operator	Meaning	Example	Output
in	True if value exists	'a' in 'apple'	True
not in	True if value does not exist	'x' not in 'apple'	True

👉 Example:

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

print("apple" in fruits)     # True

print("grapes" not in fruits) # True
```

📝 Conditional Statements in Python

What are Conditional Statements?

Conditional statements are used to make decisions in a program. They allow the computer to choose different paths of execution depending on whether a condition is True or False.

In simple words, they help the program decide what to do next based on certain rules or situations.

Indentation

Indentation in Python means giving spaces at the beginning of a line of code to indicate a block. Unlike other languages that use { } or keywords like begin and end, Python uses **indentation** to define blocks of code.
It is **mandatory** in Python, and without correct indentation, the program will show an error.

☑ **Syntax:**

```
if condition:
   # indented block
   print("This is inside the if block")
print("This is outside the if block")
```

☑ **Example:**

```
x = 10
if x > 5:
   print("x is greater than 5")  # Indented block
print("End of program")          # Not indented (outside if)
```

In Python, the **main types of conditional statements** are:

1. **if statement** – checks a single condition.
2. **if-else statement** – provides two possible paths (True/False).
3. **if-elif-else statement** – checks multiple conditions in order.
4. **nested if-else statement** – an if-else statement inside another if-else.

if Statement: The if statement is used to check a condition. If the condition is **True**, the block of code inside if runs. If the condition is **False**, the block is skipped. It is useful for **checking a single condition**.

☑ **Syntax:**

```
if condition:
   # block of code if condition is true
```

☑ **Example:** age = 20

```
if age >= 18:
   print("You are eligible to vote")
```

if-else Statement:

The **if-else statement** in Python is a conditional statement that allows the program to **choose between two paths**.

- If the condition is **True**, the code inside the **if block** is executed.
- If the condition is **False**, the code inside the **else block** is executed.

It ensures that **one of the two blocks always runs**, making it useful when we want to handle both possible outcomes of a condition.

☑ **Syntax:**

```
if condition:
   # code if true
else:
   # code if false
```

☑ **Example:**

```
marks = 40
if marks >= 35:
   print("Pass")
else:
   print("Fail")
```

Nested if-else Statement

A **nested if-else statement** means placing one **if-else statement inside another if-else statement**.

It allows a program to **check multiple conditions step by step**, where the decision inside one block depends on the result of a previous condition.

In simple words, it is used when we need to make a decision **within another decision**.

☑ **Syntax:**

```
if condition1:
   if condition2:
       # code if both conditions are true
   else:
       # code if condition1 true but condition2 false
else:
   # code if condition1 false
```

✅ **Example:**

```
num = 15
if num > 0:
    if num % 2 == 0:
        print("Positive Even Number")
    else:
        print("Positive Odd Number")
else:
    print("Negative Number")
```

Chained Conditional (if-elif-else Statement)

The **if-elif-else statement** is a conditional statement used when there are **multiple conditions to check**.

- The program tests each condition one by one in order.
- As soon as one condition is **True**, its block of code is executed, and the remaining conditions are skipped.
- If none of the conditions are True, the **else block** is executed.

It is useful when there are **many possible choices or outcomes** in a program.

✅ **Syntax:**

```
if condition1:
    # code if condition1 true
elif condition2:
    # code if condition2 true
elif condition3:
    # code if condition3 true
else:
    # code if all conditions false
```

✅ **Example:** score = 85

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

◆ Loop

A **loop** is a programming concept that allows us to repeat a set of instructions multiple times without writing them again and again. Instead of repeating the same code, we use a loop to make the computer do the task repeatedly until a certain condition is met.

In programming, a loop is used when we want to repeat a set of instructions multiple times without writing the code again and again.

◆ Iteration

Iteration means one complete execution of the loop body. In simple words, each time the loop runs, it is called **one iteration**. If a loop runs 5 times, it means the loop has **5 iterations**.

◆ Types of Loops in Python

In Python, there are mainly **two types of loops**:

1□**For Loop ->** Used when the number of repetitions is known.

2□**While Loop ->** Used when the number of repetitions is unknown (depends on a condition).

1️⃣For Loop

- A for loop is a type of loop that repeats a block of code for a specific number of times. It is mostly used when we know how many times we want to repeat something.
- It automatically goes through items one by one.

👉 **Syntax**:

```
for variable in sequence:
  # code to be executed
```

- **variable** → takes each value from the sequence one by one.
- **sequence** → can be a range of numbers, list, string, or any iterable object.
- **Indentation** (spaces) is very important in Python.

Why do we use for loop?

1. To **avoid repetition** of code.
2. To **process collections of data** (like going through all numbers in a list, or letters in a string).
3. To perform actions a **fixed number of times**.

The range() function with for loop

Most commonly, we use for loop with the range() function.

👉 **Syntax of range():**

```
range(start, stop, step)
```

- **start** → from where the loop begins (default = 0)
- **stop** → loop will stop *before this number*
- **step** → difference between numbers (default = 1)

Example 1: Print numbers from 1 to 5

```
for i in range(1, 6):
  print(i)
```

- **Output:**
- 1

- 2
- 3
- 4
- 5
- 👉 Here, i starts from 1, goes up to 5, and stops before 6.

Example 2: Print numbers from 0 to 9

```
for i in range(10):

    print(i)
```

- **Output:**
- 0
- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9

Example 3: Print even numbers between 2 to 10

```
for i in range(2, 11, 2):

    print(i)
```

- **Output:**
- 2
- 4
- 6
- 8
- 10

5. Using for loop with a list

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

for fruit in fruits:

    print(fruit)
```

- **Output:**
- apple
- banana
- mango

6. Using for loop with a string

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

- **Output:**
- P
- y
- t
- h
- o
- n

2□While Loop

- A **while loop** is a type of loop that keeps repeating a block of code as long as a given condition is **true**. It is mostly used when we **do not know exactly how many times** the code should repeat, and we want it to continue until a condition becomes **false**.
- Unlike a for loop (which runs for a fixed number of times), a while loop continues until the condition becomes **false**.

Syntax

```
while condition:
   # code to be executed
```

- **condition** → checked before each iteration.
- If **True** → code runs.
- If **False** → loop stops.
- Be careful → if condition never becomes false, loop will run **forever** (infinite loop).

4. Example Programs

✅ Example 1: Print numbers from 1 to 5

```
i = 1
while i <= 5:
   print(i)
   i = i + 1
```

Output:

```
1
2
3
4
5
```

👉 Explanation:

- Start with i = 1.
- Loop runs while i <= 5.
- After each step, increase i by 1.

✅ Example 2: Countdown from 10 to 1

```
i = 10
while i >= 1:
   print(i)
   i = i - 1
```

✅ Example 3: Print even numbers from 2 to 20

```
i = 2
while i <= 20:
    print(i)

    i = i + 2
```

✅ Example 4: Sum of numbers from 1 to 10

```
i = 1

total = 0

while i <= 10:

    total = total + i

    i = i + 1

print("Sum is:", total)
```

✅ Example 5: Print characters of a word

```
word = "Python"

i = 0

while i < len(word):

    print(word[i])

    i = i + 1
```

� 5. Flow of a While Loop (Step by Step)

1. Check the condition.
2. If true → run the code inside.
3. Update the variable (so loop doesn’t run forever).
4. Go back and check condition again.
5. Stop when condition is false.

📓 Break and Continue in Python Loops

Break

The **break** statement is used to **stop a loop immediately**, even if the loop condition is still true or the loop has not finished all its iterations. Once break is executed, the loop ends and the program continues after the loop.

1. The break Statement

- **Used to exit (stop) the loop completely.**
- As soon as Python sees break, it comes out of the loop — even if the condition is still true or numbers are left.

✅ **Example with for loop**

```
for i in range(1, 11):
    if i == 5:
        break
    print(i)
```

- **Output:**
- 1
- 2
- 3
- 4
- Loop stops as soon as i == 5.

✅ **Example with while loop**

```
i = 1
while i <= 10:
    if i == 5:
        break
    print(i)
    i = i + 1
```

- **Output:**

- 1
- 2
- 3
- 4

Continue: The **continue** statement is used to **skip the current iteration** of the loop and move to the next iteration. It does not stop the entire loop, it only skips the remaining code for the current cycle and continues with the next one.

The continue Statement

- **Used to skip the current iteration** and move to the next one.
- The loop doesn’t stop completely — it just skips that turn.

✅ **Example with for loop**

```
for i in range(1, 6):
  if i == 3:
    continue
  print(i)
```

Output:

```
1
2
4
5
```

☞ 3 is skipped.

✅ **Example with while loop**

```
i = 1
while i <= 5:
  if i == 3:
    i = i + 1   # important: update before continue
    continue
```

```
   print(i)
   i = i + 1
```

Output:

```
1
2
4
5
```

👉 Again, 3 is skipped.

◆3. pass in Python

In Python, the **pass statement** is a **null statement**.
It is used as a **placeholder** when you want to write code later but don’t want Python to throw an error for leaving the block empty.

✅ **Why we use pass?**

- Python does **not allow empty code blocks** (like inside functions, classes, loops, or conditionals).
- If you leave them empty, Python will give an **IndentationError** or **SyntaxError**.
- To avoid this, we use **pass** as a temporary statement.
- It does **nothing** when executed.

✅ Example 1: Using pass in if statement

```
x = 10


if x > 5:
   pass  # do nothing for now
else:
   print("x is less than or equal to 5") # Here, the program won’t give an error, even though the if block is empty.
```

✅ Example 2: Using pass in a function

```
def my_function():
    pass  # function will be written later
```

🔹 Without pass, Python would give an error because the function body is empty.

✅ Example 3: Using pass in a class

```
class MyClass:
    pass  # class definition will come later
```

✅ Example 4: Using pass in loops

```
for i in range(5):
    pass  # loop logic will be added later
```

🔹 3. Difference Between break and continue

Statement	What it Does	Example Behavior
break	Stops the loop completely	You leave the classroom immediately
continue	Skips the current step, moves to next	You skip one question but continue with the rest
pass	Does nothing, just a placeholder	You sit quietly in class and do nothing

Collections Data Types in Python

In Python, **collection data types** are used to **store multiple values in a single variable**.
They allow us to group data together, access them, and perform different operations easily.

Types of Collection Data Types in Python

Python has **4 built-in collection data types**:

1️⃣ **List**

2️⃣ **Tuple**

3️⃣ **Set**

4️⃣ **Dictionary**

List in Python

Definition:
Creating a List

```
# Empty list

my_list = []


# List with numbers

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

# List with strings
# List with boolean

# List with mixed data types

mixed = [10, "Hello", 3.5, True]


# Nested list

nested = [[1, 2], [3, 4]]
```

Operations on Lists

1. Accessing Elements

Accessing elements in a list means retrieving a value stored at a particular position.

- In Python, every element in a list is stored with a position number called an **index**.
- Indexing starts from **0** (first element) and goes up to **n-1** (last element).
- We can also use **negative indexing** to access elements from the end. Here, -1 refers to the last element, -2 to the second last, and so on.

By **index** (index starts from 0).

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

print(fruits[0])   # apple

print(fruits[2])   # cherry
```

- **Negative Indexing** (from end).

```
print(fruits[-1])  # cherry

print(fruits[-2])  # banana
```

---

2. Slicing Lists

Slicing means taking out a part (or sublist) from the main list, it’s extracting a part of the list using [start:end].

- It is done by specifying a starting index and an ending index.
- The slice includes the starting index but excludes the ending index.
- We can also leave the start or end blank to include all elements up to that side.
- Negative indexes can also be used to slice from the end.

```
numbers = [10, 20, 30, 40, 50]

print(numbers[1:4])   # [20, 30, 40]

print(numbers[:3])    # [10, 20, 30]
print(numbers[2:])    # [30, 40, 50]

print(numbers[-3:])   # [30, 40, 50]
```

---

3. Adding Elements

We can add new values to a list in different ways:

- **Appending: Adds a single element at the end.**
- **Inserting: Adds an element at a specific position (index) inside the list.**
- **Extending: Adds multiple elements at once by combining another list or sequence.**

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


# Add single element at end

fruits.append("cherry")

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


# Add element at specific index

fruits.insert(1, "orange")

print(fruits)  # ['apple', 'orange', 'banana', 'cherry']


# Add multiple elements

fruits.extend(["mango", "grapes"])

print(fruits)  # ['apple', 'orange', 'banana', 'cherry', 'mango', 'grapes']
```

---

4. Removing Elements

Removing means deleting elements from a list. Different methods are available:

- Remove(): Deletes the first occurrence of the specified element.

- Pop(): Deletes an element at a specific index (if index is not given, it removes the last element).
- Clear(): Deletes all elements, making the list empty.

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


fruits.remove("banana")    # removes first occurrence

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


fruits.pop(1)   # removes element at index 1

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


fruits.pop()    # removes last element

print(fruits)  # ['apple']


fruits.clear()  # removes all elements

print(fruits)  # []
```

5. Changing Elements

Lists are mutable, meaning we can change the value of elements after creating the list.

- We can update or replace an element by directly assigning a new value at a specific index.

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

fruits[1] = "orange"

print(fruits)  # ['apple', 'orange', 'cherry']
```

6. List Operations (Mathematical)

Lists support some mathematical-like operations:

- Concatenation (+): Joins two lists together into one.
- Repetition (*): Repeats the elements of a list multiple times.

```
# Concatenation
a = [1, 2]
b = [3, 4]
print(a + b)   # [1, 2, 3, 4]

# Repetition
print(a * 3)   # [1, 2, 1, 2, 1, 2]
```

---

7. Searching and Checking

We can check if an element exists in a list or not:

- in operator: Returns True if the element is found.
- not in operator: Returns True if the element is not present.
- index(): Gives the position of the first occurrence of an element.
- count(): Tells how many times a particular element appears.

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

print("banana" in fruits)   # True
print("mango" not in fruits) # True

print(fruits.index("cherry")) # 2
print(fruits.count("apple"))  # 1
```

8. Sorting and Reversing

Lists can be rearranged in order:

- Sorting: Arranges elements either in ascending order (smallest to largest) or descending order (largest to smallest).
- Reversing: Flips the list so the last element becomes the first and vice versa.

```
numbers = [4, 2, 9, 1]

numbers.sort()      # Ascending order
print(numbers)  # [1, 2, 4, 9]

numbers.sort(reverse=True)  # Descending order
print(numbers)  # [9, 4, 2, 1]

numbers.reverse()   # Reverse list
print(numbers)  # [1, 2, 4, 9] (after reverse)
```

9. Copying a List

Sometimes, we need a separate copy of a list to work with.

- Copying creates a new list with the same elements, so changes in the new list do not affect the original one.

```
fruits = ["apple", "banana"]
copy_list = fruits.copy()
print(copy_list)  # ['apple', 'banana']
```

10. Built-in Functions with Lists

Python provides many ready-made functions to work with lists:

- len(): Returns the number of elements in the list.
- max(): Returns the largest element.
- min(): Returns the smallest element.
- sum(): Returns the total sum of all numeric elements.

```
numbers = [10, 20, 30, 40, 50]


print(len(numbers))   # 5

print(max(numbers))   # 50

print(min(numbers))   # 10

print(sum(numbers))   # 150


# Program to find maximum and minimum in a list without built-in functions


# Example list

numbers = [12, 45, 7, 89, 34, 22, 90, 3]


# Assume first element is both max and min

maximum = numbers[0]

minimum = numbers[0]


# Loop through the list

for num in numbers:

    if num > maximum:
        maximum = num
    if num < minimum:
        minimum = num

print("Maximum number in the list:", maximum)
print("Minimum number in the list:", minimum)
```

Take multiple inputs in a single line

We can use input().split() to take space-separated values.

```
numbers = input("Enter numbers separated by space: ").split()
print(numbers)
```

◈ Example run:

```
Enter numbers separated by space: 10 20 30 40
['10', '20', '30', '40']
```

☞ Note: These are **strings**. To convert into **integers**:

```
numbers = list(map(int, input("Enter numbers: ").split()))
print(numbers)   # [10, 20, 30, 40]
```

Take list input one by one (using loop)

```
size = int(input("Enter number of elements: "))
numbers = []

for i in range(size):
    num = int(input(f"Enter element {i+1}: "))
    numbers.append(num)

print(numbers)
```

List Operations with Loops in Python

Practice Problems – Lists with Loops in Python

---

1. Print all elements of a list

Write a program to take a list of fruits as input and print each fruit using a loop.

```
fruits = input("Enter fruits separated by space: ").split()

print("Fruits in the list:")
for fruit in fruits:
   print(fruit)
```

---

2. Sum of all elements in a list

Take n numbers from the user, store them in a list, and find their sum using a loop.

```
numbers = list(map(int, input("Enter numbers separated by space: ").split()))

total = 0
for num in numbers:
   total += num

print("Sum of all numbers:", total)
```

---

3. Find the maximum and minimum number

Take numbers in a list and find the **largest** and **smallest** number using a loop.

```
numbers = list(map(int, input("Enter numbers separated by space: ").split()))

maximum = numbers[0]
```

```
minimum = numbers[0]


for num in numbers:
    if num > maximum:
        maximum = num
    if num < minimum:
        minimum = num


print("Maximum number:", maximum)
print("Minimum number:", minimum)
```

---

4. Search for an element in the list

Take a list of names from the user and search whether a given name exists in the list or not (using a loop, not in keyword).

```
names = input("Enter names separated by space: ").split()
search_name = input("Enter name to search: ")


found = False
for name in names:
    if name == search_name:
        found = True
        break


if found:
    print(search_name, "is in the list.")
else:
    print(search_name, "is not in the list.")
```

---

5. Count even and odd numbers

Take a list of numbers and count how many are even and how many are odd.

```
numbers = list(map(int, input("Enter numbers separated by space: ").split()))


even_count = 0
odd_count = 0


for num in numbers:
    if num % 2 == 0:
        even_count += 1
    else:
        odd_count += 1


print("Even numbers count:", even_count)
print("Odd numbers count:", odd_count)
```

---

6. Reverse a list (without using reverse function)

Take numbers in a list and print them in reverse order using a loop.

```
numbers = list(map(int, input("Enter numbers separated by space: ").split()))


print("Reversed list:")
for i in range(len(numbers) - 1, -1, -1):
    print(numbers[i], end=" ")
```

---

7. Remove all negative numbers

Take a list of integers (positive + negative) and create a new list that contains only non-negative numbers.

```
numbers = list(map(int, input("Enter numbers separated by space: ").split()))


positive_numbers = []

for num in numbers:

    if num >= 0:

        positive_numbers.append(num)


print("List without negative numbers:", positive_numbers)
```

---

8. Multiply each element by 2

Take a list of numbers and update each element by multiplying it by 2 using a loop.

```
numbers = list(map(int, input("Enter numbers separated by space: ").split()))


for i in range(len(numbers)):

    numbers[i] = numbers[i] * 2


print("Updated list:", numbers)
```

� Practice Problems

Students can try these independently.

1. Write a program to **find the average** of all numbers in a list.

   ```
   numbers = [10, 20, 30, 40, 50]   # sample list
   total = 0
   ```

```
for num in numbers:
    total += num   # add each number

average = total / len(numbers)   # sum ÷ count
print("Average =", average)
```

2. Count how many **positive, negative, and zero** values are in a list.

```
numbers = [5, -3, 0, 7, -1, 0]

positive = 0
negative = 0
zero = 0

for num in numbers:
    if num > 0:
        positive += 1
    elif num < 0:
        negative += 1
    else:
        zero += 1

print("Positive =", positive)
print("Negative =", negative)
print("Zero =", zero)
```

3. Remove **duplicate elements** from a list.

```
numbers = [2, 3, 2, 5, 3, 7, 5]
unique_list = []

for num in numbers:
    if num not in unique_list:
        unique_list.append(num)

print("List without duplicates:", unique_list)
```

4. Write a program to **separate even and odd numbers** into two new lists.

```
numbers = [10, 15, 22, 33, 40, 55]
even = []
odd = []

for num in numbers:
    if num % 2 == 0:
        even.append(num)
    else:
        odd.append(num)

print("Even numbers =", even)
print("Odd numbers =", odd)
```

5. Take a list of names and **print the longest name**.

```
names = ["Tom", "Alexander", "Bob", "Christina"]

longest = names[0]   # assume first name is longest

for name in names:
    if len(name) > len(longest):
        longest = name

print("Longest name =", longest)
```

Tuples and Their Operations in Python

What is a Tuple?

- A tuple is a collection in Python used to store multiple values in a single variable.
- Tuples are:
  1. Ordered – Items are stored in sequence.
  2. Immutable – Cannot be changed after creation.
  3. Allow duplicates – Same values can repeat.
  4. Can store mixed data types – Numbers, strings, floats, etc.

👉 Syntax:

```
my_tuple = (10, 20, 30, 40)
```

🛠 Tuple Operations

1. Creating Tuples

```
# Empty tuple
t1 = ()


# Tuple with numbers
t2 = (1, 2, 3, 4)


# Tuple with mixed data
t3 = ("apple", 10, 5.6, True)


# Tuple with one element (comma is required)
t4 = (10,)   # Correct
t5 = (10)    # Wrong → this is just an integer
```

2. Accessing Elements

- Tuples use **indexing** (starts from 0).

```
fruits = ("apple", "banana", "cherry")

print(fruits[0])   # apple

print(fruits[2])   # cherry

print(fruits[-1])  # cherry (negative index)
```

3. Slicing Tuples

- Extract part of tuple using [start:end].

```
numbers = (10, 20, 30, 40, 50)

print(numbers[1:4])   # (20, 30, 40)

print(numbers[:3])    # (10, 20, 30)

print(numbers[2:])    # (30, 40, 50)
```

4. Concatenation & Repetition

```
t1 = (1, 2, 3)

t2 = (4, 5)



# Concatenation

print(t1 + t2)   # (1, 2, 3, 4, 5)



# Repetition

print(t1 * 3)    # (1, 2, 3, 1, 2, 3, 1, 2, 3)
```

5. Membership Test

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

print("red" in colors)     # True

print("yellow" not in colors)  # True
```

---

6. Iteration (Looping)

```
animals = ("cat", "dog", "lion")

for a in animals:

   print(a)
```

---

7. Tuple Functions

```
nums = (10, 20, 30, 20, 40)


print(len(nums))    # 5 → length of tuple

print(max(nums))    # 40 → maximum value

print(min(nums))    # 10 → minimum value

print(nums.count(20)) # 2 → count occurrences of 20

print(nums.index(30)) # 2 → index of first 30
```

---

8. Nested Tuples

- Tuples inside tuples.

```
nested = (1, (2, 3), (4, 5, 6))

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

print(nested[1][0])  # 2
```

---

9. Tuple Packing & Unpacking

```
# Packing

student = ("John", 21, "CSE")


# Unpacking

name, age, branch = student
```

```
print(name)   # John

print(age)    # 21

print(branch) # CSE
```

10. **Highlight immutability**:

```
my_tuple = (10, 20, 30)

my_tuple[0] = 100   # ❌ Error
```

📓 Sets in Python – Detailed Notes

✅ Definition

- **A set is a collection data type in Python that stores multiple items in a single variable.**
- **Sets are unordered, unindexed, mutable, and they do not allow duplicate values.**
- **They are useful when you want to store unique elements and perform mathematical operations like union, intersection, and difference.**

👉 Syntax:

```
my_set = {element1, element2, element3}
```

✅ Characteristics of Sets

1. **Unordered**
   - **Elements do not have a fixed order.**
   - **Example: {1, 2, 3} and {3, 2, 1} are considered the same set.**
2. **Unindexed**
   - **You cannot access set elements by index like lists or tuples.**
   - **Example: my_set[0] ❌ will give an error.**
3. **Unique values only**
   - **Duplicate values are automatically removed.**

4. **s = {1, 2, 2, 3, 3}**
5. **print(s) # {1, 2, 3}**
6. **Mutable (changeable)**
   - **You can add or remove elements after creating a set.**
7. **Heterogeneous values**
   - **A set can store different data types (integers, strings, floats, booleans, etc.).**
8. **s = {10, "apple", 3.5, True}**
9. **print(s) # {True, 10, 'apple', 3.5}**
10. **No duplicates + unordered = no guaranteed order**
    - **Since sets are unordered, the output order may not match the input order.**

✅ **Creating Sets**

# Set with integers

A set is a collection of unique items (no duplicates).

Sets are written using { } with elements separated by commas.

s1 = {1, 2, 3, 4}

# Mixed data types

s2 = {"apple", 42, 3.14, False}

# Empty set

s3 = set()   # Correct

s4 = {}   # Wrong → creates an empty dictionary

☑ Accessing Elements

- **Direct indexing is not possible.**
- **Use a loop to access items.**

```
fruits = {"apple", "banana", "cherry"}
for fruit in fruits:
    print(fruit)
```

☑ Adding and Removing Elements

❖ Adding

```
s = {1, 2, 3}
s.add(4)        # Add a single element
print(s)        # {1, 2, 3, 4}

s.update([5, 6]) # Add multiple elements
print(s)        # {1, 2, 3, 4, 5, 6}
```

❖ Removing

```
s.remove(2)     # Removes 2, error if not found
s.discard(10)   # No error if 10 not found
s.pop()         # Removes a random element
s.clear()       # Empties the set
```

☑ Mathematical Set Operations

```
A = {1, 2, 3, 4}
B = {3, 4, 5, 6}

# Union (all unique elements)
```

```
print(A | B)      # {1, 2, 3, 4, 5, 6}
print(A.union(B))  # Same as above



# Intersection (common elements)
print(A & B)       # {3, 4}
print(A.intersection(B))

# Difference (elements in A but not in B)
print(A - B)       # {1, 2}
print(A.difference(B))

# Symmetric Difference (elements not common)
print(A ^ B)       # {1, 2, 5, 6}
print(A.symmetric_difference(B))
```

✅ Set Functions

```
nums = {10, 20, 30, 40, 20}

print(len(nums))   # 4 (unique items only)
print(max(nums))   # 40
print(min(nums))   # 10
print(sum(nums))   # 100
```

✅ Frozen Set

- A frozen set is an immutable set (cannot be changed after creation).

```
frozen = frozenset([1, 2, 3, 4])
```

print(frozen)

# frozen.add(5) ❌ Error (cannot modify)

✅ Dictionary

- A dictionary is a built-in collection data type in Python.
- It stores data in the form of key–value pairs.
- Each key is unique and works as an index to access its corresponding value.
- Keys are immutable (can't be changed once created), while values can be anything (mutable or immutable).
- Dictionaries are:
    - Ordered (from Python 3.7+)
    - Mutable (can be modified)
    - Do not allow duplicate keys

👉 Syntax:

```
my_dict = {
   "key1": "value1",
   "key2": "value2"
}
```

✅ Characteristics of Dictionaries

1. Key–Value Pairs
    - Every entry in a dictionary is a pair: key: value.
      Example:

   {"name": "Alice", "age": 20}

2. Unique Keys

   - Keys must be unique. If the same key is repeated, the latest value overwrites the previous one.
     Example:

```
d = {"a": 1, "a": 2}

print(d)   # {'a': 2}
```

3. **Keys must be Immutable**
   - Valid keys: string, number, tuple (if elements inside are immutable).
   - Invalid keys: list, dictionary (since they are mutable).
4. **Values can be Any Data Type**
   - Numbers, strings, lists, tuples, or even another dictionary.
5. **Ordered (Python 3.7+)**
   - Dictionaries preserve the order of insertion.
6. **Mutable**
   - Items can be added, modified, or deleted.

✅ Creating Dictionaries

1. **Normal Dictionary**

```
student = {"name": "John", "age": 21, "branch": "CSE"}
```

2. **Using dict() Constructor**

```
student2 = dict(name="Alice", age=22, branch="ECE")
```

3. **Empty Dictionary**

```
empty = {}
```

✅ Accessing Values

- Use key names inside square brackets or get() method.

```
student = {"name": "John", "age": 21, "branch": "CSE"}


print(student["name"])       # John
print(student.get("age"))    # 21
print(student.get("grade"))  # None (safe, avoids error if key not found)
```

✅ Adding & Updating Items

- **Adding** → You can insert a new key–value pair into a dictionary by simply assigning it.
- **Updating** → If the key already exists, assigning a new value will **overwrite** the old one.

```
student = {"name": "John", "age": 21}


# Adding a new key-value pair
student["branch"] = "CSE"   # branch key added


# Updating an existing key
student["age"] = 22        # age key updated from 21 → 22


print(student)
```

◈ **Output:**

```
{'name': 'John', 'age': 22, 'branch': 'CSE'}
```

✅ Removing Items

Python gives multiple ways to **delete items** from a dictionary:

```
student = {"name": "John", "age": 21, "branch": "CSE"}


student.pop("age")       # Removes the key 'age'
student.popitem()        # Removes the LAST inserted item (branch here)

del student["name"]       # Deletes the key 'name'

student.clear()          # Removes ALL items (dictionary becomes empty)
```

📌 **Explanation:**

- .pop(key) → Removes a specific key and **returns its value**. If the key doesn't exist → error.
- .popitem() → Removes the **last inserted key-value pair** (since Python 3.7, dicts are ordered).
- del dict[key] → Deletes a key directly.
- .clear() → Empties the entire dictionary.

✅ Dictionary Methods

Methods allow you to **access dictionary data** easily.

```
person = {"name": "Alice", "age": 25, "city": "Hyderabad"}

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

print(person.values())   # dict_values(['Alice', 25, 'Hyderabad'])

print(person.items())    # dict_items([('name', 'Alice'), ('age', 25), ('city', 'Hyderabad')])

# Update dictionary

person.update({"age": 26, "country": "India"})

print(person)
```

Output:

```
dict_keys(['name', 'age', 'city'])

dict_values(['Alice', 25, 'Hyderabad'])

dict_items([('name', 'Alice', 'age', 25), ('city', 'Hyderabad')])


{'name': 'Alice', 'age': 26, 'city': 'Hyderabad', 'country': 'India'}
```

✅ Looping through Dictionary

You can loop over **keys, values, or both**.

```
person = {"name": "Alice", "age": 25}


# Loop through keys

for key in person:

    print(key)


# Loop through values

for value in person.values():

    print(value)


# Loop through both key & value

for key, value in person.items():

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

🔹 **Output:**

```
name

age

Alice

25

name : Alice

age : 25
```

📌 **Explanation:**

- By default, looping through a dictionary iterates **keys**.
- .values() → loops through only values.
- .items() → gives **both key and value** (best way to loop).

✅ Nested Dictionary

- A **dictionary inside another dictionary**.
- Useful for storing **structured data** (like student info).

```
students = {
  "s1": {"name": "John", "age": 20},
  "s2": {"name": "Alice", "age": 22}
}


print(students["s1"]["name"])  # John
```

📌 Explanation:

- "s1" is the key for the **inner dictionary**.
- Inside "s1", "name" is another key.
- Access is done like: outer_dict["s1"]["name"].

✅ Dictionary Functions

Python has some **built-in functions** that work on dictionaries.

```
marks = {"math": 90, "science": 85, "english": 88}


print(len(marks))         # 3 → number of key-value pairs
print(sum(marks.values())) # 263 → sum of values
print(max(marks.values())) # 90 → highest value
print(min(marks.values())) # 85 → lowest value
```

✅ Dictionary Comprehension

- A **shortcut** for creating dictionaries using a single line.

- Similar to list comprehension but with {}.

```
squares = {x: x*x for x in range(1, 6)}

print(squares)
```

🔹 **Output:**

```
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

📌 **Explanation:**

- For each number x in range 1–5 → key is x, value is x*x.
- Output is a dictionary mapping numbers → their squares.

📝 Practice Problems with Solutions

🔹 Basic Level Problems

---

1. Create a dictionary with your details (name, age, city). Print values.

```
# Creating dictionary

my_details = {"name": "Nandan", "age": 21, "city": "Hyderabad"}


# Printing values

print(my_details["name"])  # Accessing using key

print(my_details["age"])

print(my_details["city"])
```

👉 **Explanation:**

- A dictionary stores information in key–value form.
- Access values using dict[key].

---

2. Add a new key "college" and update "age".

```
student = {"name": "Nandan", "age": 21, "city": "Hyderabad"}


# Adding new key
student["college"] = "ABC University"


# Updating existing key
student["age"] = 22


print(student)
```

☞ **Output:**

```
{'name': 'Nandan', 'age': 22, 'city': 'Hyderabad', 'college': 'ABC University'}
```

---

3. Write a program to print all keys of a dictionary.

```
person = {"name": "Alice", "age": 25, "city": "Delhi"}


print("Keys are:")
for key in person.keys():
   print(key)
```

☞ **Output:**

```
name
age
city
```

---

4. Remove a key using pop() and print the dictionary.

```
car = {"brand": "Toyota", "model": "Innova", "year": 2020}
```

```
car.pop("year")   # removes 'year' key
print(car)
```

☞ **Output:**

```
{'brand': 'Toyota', 'model': 'Innova'}
```

---

5. Create a dictionary with 3 fruits and their prices. Print only values.

```
fruits = {"apple": 100, "banana": 50, "mango": 120}

print("Fruit prices:")
for price in fruits.values():
   print(price)
```

☞ **Output:**

```
100
50
120
```

◆ Medium Level Problems

---

6. Count frequency of characters in "banana".

```
word = "banana"
freq = {}   # empty dictionary

for ch in word:
   if ch in freq:
      freq[ch] += 1   # if character already exists, increase count
   else:
        freq[ch] = 1    # if not, add with value 1


print(freq)
```

☞ **Output:**

```
{'b': 1, 'a': 3, 'n': 2}
```

---

7. Merge two dictionaries into one.

```
dict1 = {"a": 1, "b": 2}

dict2 = {"c": 3, "d": 4}


# Method 1: using update()

dict1.update(dict2)

print(dict1)
```

☞ **Output:**

```
{'a': 1, 'b': 2, 'c': 3, 'd': 4}
```

---

8. Write a program to print all students with marks above 50.

```
marks = {"John": 45, "Alice": 75, "Bob": 60, "Mike": 40}


print("Students with marks > 50:")

for name, score in marks.items():

    if score > 50:

        print(name, ":", score)
```

☞ **Output:**

```
Alice : 75

Bob : 60
```

---

9. Find the student with the highest marks in a dictionary.

```
marks = {"John": 45, "Alice": 75, "Bob": 60}

# max() with key parameter
topper = max(marks, key=marks.get)

print("Topper is:", topper, "with marks", marks[topper])
```

👉 **Output:**

```
Topper is: Alice with marks 75
```

10. Write a program to create a dictionary that stores words as keys and their lengths as values using dictionary comprehension. Finally, print the dictionary.

✅ **Solution**

```
# List of words
words = ["apple", "banana", "cherry"]
# Dictionary comprehension to store word : length
word_length = {word: len(word) for word in words}

print(word_length)
```

📌 **Output**

```
{'apple': 5, 'banana': 6, 'cherry': 6}
```

📝 Arrays in Python

---

1. What is an Array? (Definition)

- An **array** is a collection of elements of the **same data type** stored in a continuous memory location.
- Arrays are used to store multiple values in a single variable.
- Example: Storing 10 numbers in a single array instead of creating 10 separate variables.

👉 In simple words: An array is like a **list of similar items** kept together.

---

2. Why Arrays? (Importance)

- Saves memory.
- Easy to manage multiple values.
- Allows faster operations like searching and sorting.
- Useful in mathematical and scientific problems.

---

3. Arrays vs Lists in Python

- In Python, we often use **lists** instead of arrays because lists are more flexible.
- But Python also provides an **array module** for creating arrays when all elements must be of the same type.

Feature	**List**	**Array**
Data Types	Can store multiple data types together (e.g., int, float, string, boolean).	Can only store values of the **same data type**.
Flexibility	Very flexible – you can add any type of item.	Less flexible – restricted to one type.
Module Requirement	Built-in, no need to import.	Requires the array module (or NumPy for advanced arrays).

Feature	List	Array
Usage	General-purpose programming.	More useful in numeric and scientific calculations.
Speed	Slightly slower for numeric operations.	Faster and more memory efficient for numbers.
Example	list1 = [1, "hello", 3.5, True]	arr.array('i', [1, 2, 3, 4, 5])

---

4. Creating Arrays in Python

Using array module

```
import array as arr


# Create an integer array

numbers = arr.array('i', [1, 2, 3, 4, 5])

print(numbers)
```

Explanation:

- 'i' → type code (for integer).
- Other type codes:
  - 'i' → int
  - 'f' → float
  - 'd' → double

---

5. Accessing Array Elements

Like lists, use **indexing** (starts from 0).

```
print(numbers[0])  # First element

print(numbers[2])  # Third element
```

---

6. Adding Elements

- Use .append() to add one element.
- Use .extend() to add multiple elements.

```
numbers.append(6)

numbers.extend([7, 8])

print(numbers)
```

---

7. Removing Elements

- .remove(value) → Removes first occurrence of value.
- .pop(index) → Removes element at given index.

```
numbers.remove(3)

numbers.pop(1)

print(numbers)
```

---

8. Updating Elements

```
numbers[0] = 10   # Change first element

print(numbers)
```

---

9. Looping through Arrays

```
for num in numbers:

   print(num)
```

---

10. Basic Operations on Arrays

```
print(len(numbers))   # Length

print(sum(numbers))   # Sum of elements
print(max(numbers))   # Maximum element

print(min(numbers))   # Minimum element
```

📓 NumPy and NumPy Arrays

✅ What is NumPy?

- **NumPy (Numerical Python)** is a **Python library** used for **scientific and mathematical computing**.
- It provides:
  - Powerful **array objects** (more efficient than Python lists).
  - **Vectorized operations** (fast element-wise calculations).
  - Support for **linear algebra, statistics, Fourier transforms, random numbers**, etc.

👉 To use NumPy, install it (if not installed):

```
pip install numpy
```

✅ Importing NumPy

```
import numpy as np
```

3. NumPy Arrays

✅ Creating a NumPy Array

```
import numpy as np


# 1D array

arr1 = np.array([1, 2, 3, 4, 5])

print(arr1)
```

```
# 2D array (matrix)

arr2 = np.array([[1, 2, 3], [4, 5, 6]])

print(arr2)
```

👉 ndarray means “n-dimensional array.”

- arr1 → 1D
- arr2 → 2D

---

4. Methods and Operations in NumPy Arrays

A. Basic Array Information

```
arr = np.array([1, 2, 3, 4, 5])


print(arr.ndim)      # Returns the number of dimensions (axes) of the array.

print(arr.shape)     # Returns a tuple showing the size of the array in each dimension (rows, columns, etc.).

print(arr.size)      # Returns the total number of elements in the array.

print(arr.dtype)     # Returns the data type of the elements in the array.
```

✅ **Output:**

```
1

(5,)

5

int32   # (on most systems, could also be int64 depending on your OS/processor)
```

Explanation of Output:

1. **arr.ndim → 1**
   Because this is a **1D array** (just one row of elements).
2. **arr.shape → (5,)**
   This means the array has **5 elements in one dimension**.

3. **arr.size → 5**
   There are exactly **5 total elements** in the array.
4. **arr.dtype → int32 / int64**
   Shows the **data type of elements** (here integers).
   - On **Windows (64-bit)** → usually int32
   - On **Linux/macOS (64-bit)** → usually int64

B. Creating Arrays with NumPy

```
import numpy as np


print(np.zeros(5))      # Creates an array of 5 zeros → [0. 0. 0. 0. 0.]

print(np.ones(5))       # Creates an array of 5 ones → [1. 1. 1. 1. 1.]

print(np.arange(1, 10, 2))  # Creates an array from 1 to 9 with step 2 → [1 3 5 7 9]

print(np.linspace(0, 1, 5)) # Creates 5 numbers evenly spaced between 0 and 1 → [0.   0.25 0.5 0.75 1.  ]
```

C. Indexing and Slicing

```
arr = np.array([10, 20, 30, 40, 50])


print(arr[0])      # First element → 10

print(arr[-1])     # Last element → 50

print(arr[1:4])    # Elements from index 1 to 3 → [20 30 40]
```

D. Mathematical Operations

```
arr = np.array([1, 2, 3, 4, 5])
```

```
print(arr + 5)    # Add 5 to every element → [ 6  7  8  9 10]

print(arr * 2)    # Multiply every element by 2 → [ 2  4  6  8 10]

print(arr ** 2)   # Square of each element → [ 1  4  9 16 25]
```

E. Aggregate Functions

```
arr = np.array([10, 20, 30, 40, 50])


print(np.sum(arr))     # Sum → 150

print(np.mean(arr))    # Average → 30

print(np.max(arr))     # Maximum → 50

print(np.min(arr))     # Minimum → 10

print(np.std(arr))     # Standard Deviation
```

G. Multi-Dimensional Arrays (Matrix Operations)

```
a = np.array([[1, 2], [3, 4]])

b = np.array([[5, 6], [7, 8]])


print(a + b)      # Matrix addition

print(a * b)      # Element-wise multiplication

print(np.dot(a, b))  # Matrix multiplication
```

📘 Functions in Python

1. Defining a Function

A **function** is a block of code that performs a specific task.
It allows us to group instructions together and reuse them whenever needed.
Instead of writing the same code again and again, we just **define a function once and call it whenever required**.

🔹 Defining a Function

Definition: Defining a function means creating it using the def keyword.

- We give the function a name.
- We can give it parameters (inputs).
- Inside it, we write the code that will run when the function is called.

Syntax:

```
def function_name(parameters):
    # function body
    return value   # (optional)
```

example:

```
def greet():
    print("Hello, Welcome to Python!")
```

---

2. Calling a Function

📘 Calling a Function – Definition

Calling a function means **telling Python to run the code inside a function** by using its name followed by parentheses ().

- If the function needs input (parameters), we provide values (arguments) inside the parentheses.
- When we call a function, Python jumps to the function, executes its code, and then comes back to continue the program.

```
def greet():
   print("Hello, Welcome to Python!")


greet()   # calling
# Output: Hello, Welcome to Python!
```

💡 **Key Point:** Function **must be defined before it is called**.

---

3. Passing Parameters & Arguments

- **Parameter →** A variable declared inside the function definition. It acts like an *empty container* waiting to receive a value.
- **Argument →** The actual value you pass into the function when calling it. It fills the container (parameter).

```
def greet(name):   # name is parameter
   print("Hello", name)


greet("Nandan")    # "Nandan" is argument
```

📘 Types of Function Arguments

A. Positional Arguments

When we pass arguments in the same order as the function parameters, they are called **positional arguments**.
Here, the position (order) is very important. If you change the order, the result will also change.

```
def student(name, age):
   print(name, "is", age, "years old")


student("Ravi", 20)   # Correct
# student(20, "Ravi") → Wrong order
```

B. Keyword Arguments

When we pass values by writing the **parameter name = value**, they are called **keyword arguments**.
Here, order doesn’t matter, because Python knows which value belongs to which parameter. def

```
def student(name, age):

    print(name, "is", age, "years old")



student(age=22, name="Anjali")
```

☞ Output: Anjali is 22 years old

C. Default Arguments

When a parameter has a default value in the function definition, it becomes a **default argument**.
If no argument is given during the call, the default value is used.
If an argument is given, it overrides the default.

```
def greet(name="Student"):

    print("Hello", name)



greet()         # Output: Hello Student

greet("Ravi")    # Output: Hello Ravi
```

D. Variable-Length Arguments

**Sometimes we don’t know how many arguments will be passed to a function.
Python provides two special ways to handle this:**

*args → **(Non-keyword variable-length arguments)**

- Collects multiple values into a **tuple**.

```
def add(*numbers):
    print(sum(numbers))


add(10, 20, 30)   # Output: 60
```

- **kwargs → **(Keyword variable-length arguments)**
- Collects multiple key=value pairs into a **dictionary**.

```
def details(**info):
    for key, value in info.items():
        print(key, ":", value)


details(name="Ravi", age=20, course="Python")
```

Can a Function Have Both *args and **kwargs?

✅ **Yes, a function in Python can have both *args and **kwargs together.**

- *args → collects **extra positional arguments** into a tuple.
- **kwargs → collects **extra keyword arguments** into a dictionary.

When both are used, the function can accept **any number of positional arguments** and **any number of keyword arguments** at the same time.

📘 Scope of Variables

🔹 Definition of Scope

The scope of a variable means the part of the program where that variable can be accessed or used.
In Python, variables can be local or global, depending on where they are created.

Local Variable

A variable declared **inside a function** is called a **local variable**.

- It exists only while the function is running.
- It cannot be used outside that function.

Global Variable

A variable declared **outside all functions** is called a **global variable**.

- It can be accessed anywhere in the program (inside or outside functions).

```
x = 10   # global variable


def test():
    y = 5   # local variable
    print("Inside function:", x, y)


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

📓 Recursive Functions

Recursion is when a function calls **itself** to solve a smaller part of the same problem.
Think: “I’ll solve this by asking a smaller version of the same question.”

Two parts every recursive function needs:

1. **Base case** — the simplest input you know the answer to (so recursion stops).
2. **Recursive case** — how to reduce the problem and call the same function on the smaller problem.

Short rule: *Check base case → otherwise break the problem into smaller part(s) and call the function on them → combine results and return.*

2) Why use recursion? When to use it

- Natural for problems defined in terms of smaller subproblems (factorials, tree traversals, divide-and-conquer like binary search, etc.).
- Makes code short and expressive for certain problems.
- But recursion can be slower or use more memory (call stack) for large inputs.

3) The best simple example: factorial

Definition: n! = n * (n-1) * (n-2) * ... * 1 and 0! = 1.

Recursive idea:

- **Base case:** 0! = 1 (or 1! = 1)
- **Recursive case:** n! = n * (n-1)!

Python code

```
def factorial(n):
    if n == 0:          # base case
        return 1
    else:
        return n * factorial(n - 1)   # recursive call
```

4) Step-by-step trace of factorial(4)

Explain by writing the calls and returns (call stack):

Call sequence (what happens first):

```
factorial(4)
 -> needs factorial(3)
    -> needs factorial(2)
       -> needs factorial(1)
          -> needs factorial(0)
             -> base case: return 1
          -> factorial(1) returns 1 * 1 = 1
       -> factorial(2) returns 2 * 1 = 2
    -> factorial(3) returns 3 * 2 = 6
 -> factorial(4) returns 4 * 6 = 24
```

Call stack view (top = current active call):

[active: factorial(4)]

```
[ factorial(3) ]

[ factorial(2) ]

[ factorial(1) ]

[ factorial(0) ]   <-- base case returns 1
```

Then return up the stack multiplying at each step.

📓 Anonymous (Lambda) Functions

One-line functions without def.

```
square = lambda x: x*x

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

📓 Higher Order Functions

Python provides some very powerful **built-in higher-order functions** that make it easier to process collections (like lists, tuples).
The three most common ones are:

✅ map()
✅ filter()
✅ reduce() (in functools module

map()

map() applies a **function** to each element of a sequence (list, tuple, etc.) and returns a new sequence (iterator).

👉 Think: *"Do this function to every element in the list."*

Syntax: map(function, iterable)

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

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

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

filter()

filter() selects elements from a sequence that satisfy a **condition** (True/False).

☞ Think: *“Pick only those elements that pass the test.”*

Syntax:

filter(function, iterable)

```
numbers = [10, 15, 20, 25, 30]

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

print(evens)  # [2, 4]
```

reduce()

reduce() is in the **functools module**. It applies a function **cumulatively** to the elements of a sequence, reducing it to a **single value**.

Syntax:

```
from functools import reduce

reduce(function, iterable)


from functools import reduce


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


# Add all numbers together

result = reduce(lambda x, y: x + y, numbers)

print(result)   # 15
```

🔹 Functions Practice Problems in Python

✅ 1. Basic Level (for very beginners)

👉 Goal: Learn to **define** and **call** a function.

1. Write a function to print **"Hello, World!"**.
2. Write a function that takes a **name** as input and prints "Hello, <name>".
3. Write a function that prints the **sum of two fixed numbers** (e.g., 10 + 20).
4. Write a function to check if a number is **even or odd**.
5. Write a function to **return** the square of a number.

✅ 2. Medium Level (parameters, return values, small logic)

👉 Goal: Learn **arguments, return, conditions, loops inside functions**.

1. Write a function that takes two numbers and **returns their sum**.
2. Write a function that takes a number n and returns the **factorial** (using a loop).
3. Write a function that takes a list of numbers and returns the **largest number**.
4. Write a function that takes a list of numbers and returns a **new list with only even numbers**.
5. Write a function that takes a string and returns the **number of vowels** in it.

💻 Example (Medium level – Factorial with loop):

```
def factorial(n):
    result = 1
    for i in range(1, n+1):
        result *= i
    return result


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

✅ 3. Hard Level (recursion, higher-order functions, nested logic)

👉 Goal: Learn **recursion, *args, **kwargs, higher-order functions, problem-solving**.

1. Write a recursive function to find the **factorial** of a number.
2. Write a recursive function to generate the **Fibonacci sequence** up to n terms.
3. Write a function that takes a list of numbers and uses **map()** to return their squares.
4. Write a function that uses **filter()** to keep only prime numbers from a list.
5. Write a function that uses **reduce()** to calculate the product of all numbers in a list.
6. Write a recursive function to calculate the **sum of digits** of a number.
7. Write a function that accepts **any number of arguments** (*args) and returns their sum.
8. Write a function that counts the frequency of each word in a sentence (use **dictionary inside function**).

💻 Example (Hard level – Fibonacci Recursion):

```
def fibonacci(n):
  if n <= 1:
    return n
  else:
    return fibonacci(n-1) + fibonacci(n-2)


for i in range(6):
  print(fibonacci(i), end=" ")
# Output: 0 1 1 2 3 5
```

📓 File Handling in Python

Introduction to Files

- A **file** is a collection of data stored on a computer's hard drive (permanent storage).
- When you run a program, data is stored in **RAM (temporary memory)** and lost when the program ends.
- Files allow us to **store data permanently** for future use.

File Handling

File handling in Python means **working with files** (creating, opening, reading, writing, updating, and closing) so that data is stored **permanently** instead of being lost when the program ends.

👉 Example:

- Notes app saves your text in a file.
- A game saves your progress in a file.
- Banking software stores account records in files/databases.

Types of Files

In Python, there are **two types of files**:

1. **Text Files (.txt, .csv, .log)**
   - Store data in human-readable format (characters).
   - Each line ends with \n.
   - Example:
   - Nandan - 90
   - Aarav - 85
   - Priya - 92
2. **Binary Files (.bin, .jpg, .exe, .mp3)**
   - Store data in **0s and 1s** (machine-readable).
   - Example: images, videos, executables.
   - Not directly readable in Notepad.

Explain the Basics

File Operations in Python:

There are **four main operations** on files:

1. **Open** → Open the file for use
2. **Read** → Get data from the file
3. **Write** → Put data into the file
4. **Close** → Close the file after work

📌 Python uses the open() function:

open("filename", "mode")

Access Modes in Python

When opening a file, we specify a mode (what we want to do with it).

📌 **Syntax:**

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

Mode	**Meaning**	**Example Use**
"r"	**Read (default)**	Opens file for reading. Error if file doesn’t exist.
"w"	**Write**	Creates a new file OR overwrites existing content.
"a"	**Append**	Opens file for writing. Adds new data at the end.
"rb"	**Read Binary**	For binary files (images, videos).
"wb"	**Write Binary**	Write to binary file.
"r+"	**Read + Write**	Can read and write, file must exist.
"w+"	**Write + Read**	Creates new file, can read/write. Old content deleted.

1. Opening a File

Opening a file means using the open() function to access a file in a specific **mode** (read, write, append, etc.) before working with it.

📌 Syntax:

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

- "filename" → Name of the file (e.g., "data.txt")
- "mode" → What you want to do with the file

2. Closing a File

- Closing a file means using the close() function to end the connection with the file and save changes safely.
- After working with a file, you must **close** it to save changes and free memory.

```
file.close()
```

---

3. Writing to a File

Writing to a file means storing new data into a file and removes the old data using write() or writelines().

```
file = open("students.txt", "w")   # open in write mode

file.write("Hello Students!\n")

file.write("Welcome to File Handling.\n")

file.close()

print("Data written successfully!")
```

3.1 Writing multiple lines with writelines()

```
# Writing multiple student names into a file

lines = ["Nandan\n", "Aarav\n", "Priya\n"]

file = open("students.txt", "w")

file.writelines(lines)

file.close()

print("Data written successfully!")
```

👉 This creates a file students.txt with the given content.
⚠ If file already exists, **old content will be erased**.

4. Appending to a File

Appending to a file means adding new data at the end of the file without deleting the existing content.

```
file = open("students.txt", "a")   # open in append mode

file.write("This line is added later.\n")

file.close()
```

☞ New content is added **without removing old data**.

5. Reading from a File

Reading from a file means getting data from a file into the program using read(), readline(), or readlines().

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

content = file.read()              # read whole file

print("File Content:\n", content)

file.close()
```

5.1 Different Read Methods

Python provides multiple ways to read a file:

a. readline() → Reads **one line at a time**.

```
f = open("students.txt", "r")

print(f.readline())   # first line

print(f.readline())   # second line

f.close()
```

b. readlines() → Reads all lines and returns a **list**.

```
f = open("students.txt", "r")

lines = f.readlines()

print(lines)

f.close()
```

6. Reading Line by Line

Reading line by line means accessing the file content **one line at a time**, usually using a loop.

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

☞ Useful when reading large files.

7. Using with Statement

- Normally, we must call close() after file operations.
- But with with, the file closes **automatically** after use.

```
with open("students.txt", "r") as f:
    data = f.read()
    print(data)
```

8. Practical Example

Storing Student Marks

```
# Writing student data into a file
with open("marks.txt", "w") as f:
    f.write("Nandan - 90\n")
    f.write("Aarav - 85\n")
    f.write("Anurag - 92\n")

# Reading back student data
with open("marks.txt", "r") as f:
    print("Student Marks:")
    for line in f:
        print(line.strip())
```

📌 Output:

```
Student Marks:

Nandan - 90

Aarav - 85

Anurag - 92
```

9. Why is File Handling Important?

- Data is **stored permanently**.
- Helps in creating **real-world applications**:
    - Saving notes in an app
    - Recording marks or employee details
    - Logging activities (log files)
    - Reading and analyzing large datasets

👉 Without file handling, data is lost when the program ends.

10. File Handling Practice Problems

1. Write a program to create a new file called student.txt.
2. Write a program to write the text "Hello Students!" into a file.
3. Write a program to read and display the contents of a file named info.txt.
4. Write a program to append "Good Luck!" to the existing file student.txt.
5. Write a program to read a file line by line and print each line.
6. Write a program to write 5 lines (entered by the user) into a file called notes.txt.
7. Write a program to count the number of lines in a file called data.txt.
8. Write a program to copy the contents of one file (source.txt) to another file (destination.txt).
9. Write a program to write a list of fruits ["Apple", "Banana", "Mango"] to a file using writelines().
10. Write a program to read the first 20 characters of a file using the read() method.

📘 Error Handling in Python – Notes

1. Introduction to Errors and Exceptions

- Errors are mistakes in a program that stop it from working correctly.
- **Exception** is a special type of error that occurs while the program is running.
- Python provides ways to **handle exceptions** so that the program doesn't crash suddenly.

---

2. Types of Errors

🔹 Compile-Time Errors

- Errors that happen when the code is not written correctly (wrong syntax).
- Example:

```
print("Hello"   # Missing closing bracket → SyntaxError
```

🔹 Logical Errors

- Errors where the program runs but gives **wrong output** because the logic is wrong.
- Example:

```
# Find average, but forgot to divide by 2

a, b = 10, 20

print("Average:", a + b)  # Wrong logic → 30 instead of 15
```

🔹 Runtime Errors

- Errors that happen **while the program is running**.
- Example:

```
x = 10 / 0   # ZeroDivisionError
```

---

3. Types of Exceptions in Python

Some common exceptions are:

- **ZeroDivisionError** → Dividing a number by zero.
- **ValueError** → Invalid value given (e.g., entering text instead of a number).

- **IndexError** → Accessing a list index that doesn’t exist.
- **KeyError** → Accessing a dictionary key that doesn’t exist.
- **TypeError** → Using the wrong type in an operation.
- **FileNotFoundError** → Trying to open a file that doesn’t exist.

---

4. Exception Handling in Python

Python uses **try-except-finally** statements:

◆ **try block**

- Code that may cause an error is written inside try.

◆ **except block**

- Code to handle the error is written inside except.

◆ **finally block**

- Code inside finally always runs, whether an error happens or not.

✅ Example:

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
    print("Result:", result)
except ZeroDivisionError:
    print("❌ You cannot divide by zero!")
except ValueError:
    print("❌ Invalid input. Please enter a number.")
finally:
    print("Program finished.")
```

Useful Python Links

- Official Python Documentation: https://docs.python.org/3/
- W3Schools Python Tutorial: https://www.w3schools.com/python/
- GeeksforGeeks Python: https://www.geeksforgeeks.org/python-programming-language/
- Programiz Python: https://www.programiz.com/python-programming

Useful Practice Platforms

- HackerRank Python: https://www.hackerrank.com/domains/tutorials/10-days-of-python
- LeetCode Python: https://leetcode.com/
- CodingBat Python: https://codingbat.com/python