

PRACTICAL COURSE • CLASSROOM READY • 30 CHAPTERS



30-DAY PYTHON

LEARNING PLAN

Learn a little every day • build a skill for life

2026 STUDENT EDITION

```
day_1.py

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

Fundamentals

Control Flow

Data & Functions

Files & OOP

5 Projects

✓ Zero to Confident Coder ✓ 150+ Practice Exercises ✓ Real Mini Projects

AUTHOR

Gaurav Malviya



CONSISTENCY > MOTIVATION

Index

Welcome to the Course	3
------------------------------	----------

WEEK 1 • Python Fundamentals

Day 1	Introduction to Python	6
Day 2	Setup & First Program	8
Day 3	Variables & Data Types	10
Day 4	Basic Operators	13
Day 5	Input & Output	16
Day 6	Type Casting	19
Day 7	Mini Project & Review	22

WEEK 2 • Control Flow

Day 8	If Statements	27
Day 9	If-Else & Elif Statements	30
Day 10	Loops (For, While)	34
Day 11	Break, Continue & Pass	38
Day 12	Nested Loops	41
Day 13	Practice Problems	44
Day 14	Mini Project & Review	47

WEEK 3 • Data Structures & Functions

Day 15	Lists	51
Day 16	Tuples	55
Day 17	Sets	58
Day 18	Dictionaries	61
Day 19	Functions Basics	64
Day 20	Return Statement	67
Day 21	Mini Project & Review	71

WEEK 4 • Modules, Files & OOP

Day 22	Modules & Import	76
Day 23	File Handling (Basics)	79
Day 24	Errors & Exceptions	82
Day 25	OOP Introduction	86
Day 26	Classes & Objects	90
Day 27	Practice Problems	94
Day 28	Mini Project & Review	97

FINAL PUSH • Projects

Day 29	Build a Small Project	102
Day 30	Final Project & Review	106

APPENDICES

App. A	Quick Reference Card	111
App. B	Common Errors & Fixes	113
App. C	30-Day Plan at a Glance	114

Welcome to the Course

This book takes an absolute beginner to a confident Python programmer in one month. Each day is one chapter, and every chapter follows the same rhythm: **Understanding** (the idea in plain language), **Definition** (exam-ready wording), **Syntax**, **Examples with output**, **Practice Exercises**, and **Key Points** for revision.

How the Course Is Divided



Your 30-Day Learning Journey



For Teachers

Teach one chapter per day. Every 7th day is a **Mini Project & Review** day with a complete solution, extension challenges, and a marking rubric. The recommended session plan:

Recommended 60-Minute Class Plan (every chapter)

Concept & Definitions 20 min	Live Coding Examples 20 min	Student Practice 15 min	Recap & Key Points 5 min
---------------------------------	--------------------------------	----------------------------	-----------------------------

For Students

Do not skip days — each chapter builds on the last. Type every example yourself and run it before reading the explanation. Reading code is not learning code; typing it, breaking it, and fixing it is. All you need is a computer with **Python 3.10+** (free from python.org), an editor such as IDLE, Thonny, or VS Code, and a notebook.

Days 1-7

WEEK 1: Python Fundamentals

Install Python, write your first programs, and master variables, data types, operators, input/output, and type casting. By Day 7 students will build their first mini project.

Understanding

Imagine you want to give instructions to a very obedient but very literal servant. He will do *exactly* what you say — nothing more, nothing less. A computer is that servant, and a **programming language** is the language you use to give it instructions.

Python is one of the easiest programming languages for humans to read and write. Its instructions look almost like English sentences. That is why Python is the world's most popular first language — and also why it powers Google, YouTube, Instagram, NASA projects, and modern Artificial Intelligence.

Why learn Python?

- **Simple syntax** — less code to do more work
- **Free and open source** — anyone can download and use it
- **Huge job demand** — data science, web development, automation, AI/ML
- **Massive community** — millions of learners and free resources

Definition

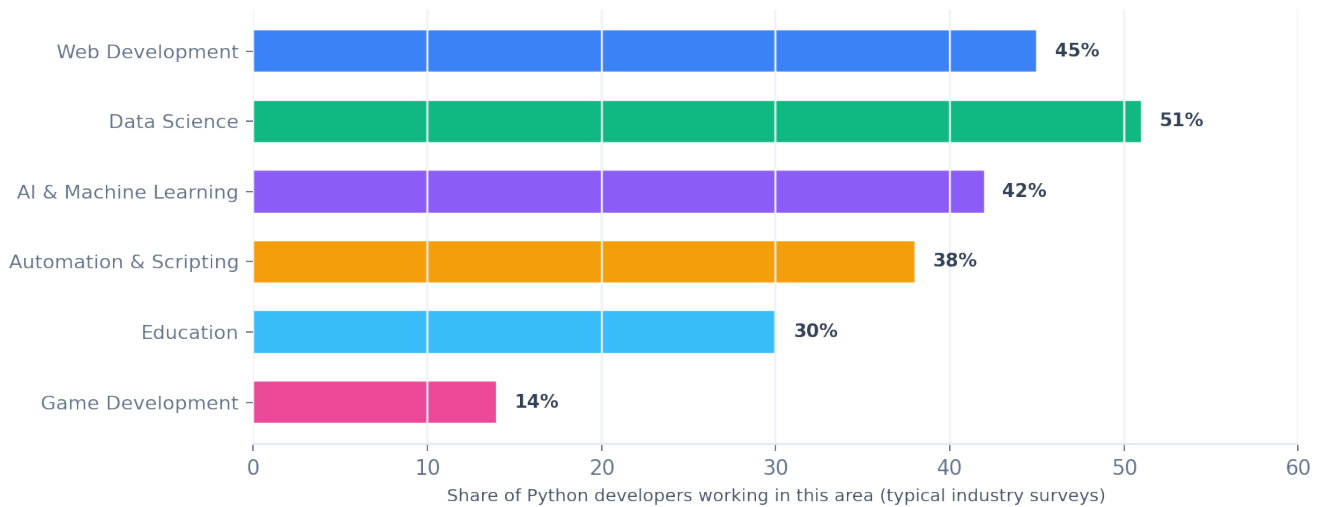
Python is a high-level, interpreted, general-purpose programming language created by **Guido van Rossum** and first released in **1991**. “High-level” means it is close to human language; “interpreted” means the code runs line by line without needing to be compiled first.

Key Terms

- **Program** — a set of instructions given to a computer to perform a task.
- **Interpreter** — the software that reads Python code line by line and executes it.
- **IDE (Integrated Development Environment)** — software where you write, run, and debug code (e.g., IDLE, VS Code, Thonny).

Where Python Is Used

Where Python Is Used in the Real World



Field	Real Example
Web development	Instagram, YouTube backends
Data science	Analyzing sales, weather, cricket statistics
AI & Machine Learning	ChatGPT-style models, face recognition
Automation	Auto-sending emails, renaming 1,000 files at once
Game development	Simple 2D games with Pygame
App scripting	Blender, GIS software

Classroom Activity

1. Discuss: name 5 apps you used today. Which ones might use Python?
2. Visit **python.org** and explore the official website.
3. Write in your notebook: three reasons *you* want to learn Python.

Key Points

- Python is high-level, interpreted, and beginner-friendly.
- Created by Guido van Rossum in 1991.
- Used in web, data science, AI, automation, and more.
- Python code is executed line by line by the interpreter.

Setup & First Program

Understanding

Before a carpenter builds furniture, he sets up his workshop. Today we set up our programming workshop: we install Python, meet the editor, and write our first program.

The tradition in programming is that your first program prints the words *Hello, World!* on the screen. In many languages this takes 5-10 lines. In Python, it takes **one**.

Installing Python (Windows)

1. Go to **python.org** → **Downloads** and download the latest Python 3 version.
2. Run the installer. **Important:** tick the checkbox **"Add Python to PATH"** before clicking Install.
3. Verify: open Command Prompt and type `python --version`. You should see something like `Python 3.12.4`.

On most Linux systems and macOS, Python 3 is pre-installed — check with `python3 --version`.

Two Ways to Run Python

1. **Interactive mode (the shell/REPL)** — type `python` in the terminal. You get a `>>>` prompt. Each line runs immediately. Great for quick experiments.
2. **Script mode** — write code in a file ending with `.py` (e.g., `hello.py`) and run it with `python hello.py`. This is how real programs are written.

Your First Program

```
# hello.py - my first Python program
print("Hello, World!")
print("Welcome to the 30-Day Python Course")
```

Output:

```
Hello, World!
Welcome to the 30-Day Python Course
```

Understanding the Code

- `print()` is a **built-in function** — a ready-made command that displays text on the screen.
- The text inside quotes is called a **string**.

- The line starting with `#` is a **comment** — Python ignores it. Comments are notes for humans.

```
# This is a single-line comment  
print("Comments help explain code") # comment can follow code too
```

Common Beginner Errors

```
print("Hello)      # Error: missing closing quote  
Print("Hello")    # Error: Python is case-sensitive; 'Print' is not defined  
print "Hello"      # Error: parentheses are required in Python 3
```

Read error messages carefully — they tell you the line number and the problem. Errors are teachers, not enemies.

Practice Exercises

1. Write a program that prints your name, your city, and your goal in three separate lines.
2. Write a program that prints a triangle using `*` characters.
3. Deliberately make each of the three errors above, run them, and write down what error message Python shows.
4. Print the following exactly: `She said, "Python is easy!"` (Hint: use single quotes outside.)

Key Points

- Install Python from python.org; always tick “Add to PATH” on Windows.
- `print()` displays output on screen.
- `#` starts a comment; Python ignores it.
- Python is case-sensitive: `print` works, `Print` does not.

Understanding

Think of a variable as a **labeled container**. In your kitchen you have containers labeled “Sugar”, “Rice”, “Tea”. You can look inside them, refill them, or replace the contents — the label stays.

In Python, a variable is a name that refers to a value stored in memory:

```
age = 15
```

Here `age` is the label, `15` is the value, and `=` means “store this value under this name” (it is called the **assignment operator**, not “equals”).

Definition

A **variable** is a named reference to a value stored in the computer’s memory. A **data type** describes what kind of value a variable holds (number, text, true/false, etc.), which decides what operations can be performed on it.

Rules for Naming Variables

- Must start with a letter or underscore: `name`, `_count` ✓
- Cannot start with a digit: `2name` ✗
- Can contain letters, digits, underscore: `student_1` ✓
- No spaces or symbols: `my name`, `roll-no` ✗
- Case-sensitive: `Age` and `age` are different variables
- Cannot be a Python keyword: `if`, `for`, `class`, `print` ✗

Good practice: use meaningful names — `total_marks` is better than `tm`.

Basic Data Types

Type	Name	Example	Use
<code>int</code>	Integer	<code>25</code> , <code>-3</code> , <code>0</code>	Whole numbers
<code>float</code>	Floating point	<code>3.14</code> , <code>-0.5</code>	Decimal numbers
<code>str</code>	String	<code>"Ravi"</code> , <code>'A'</code>	Text
<code>bool</code>	Boolean	<code>True</code> , <code>False</code>	Yes/No logic
<code>NoneType</code>	None	<code>None</code>	"No value yet"

Examples

```
name = "Aarav"      # str
age = 14             # int
height = 5.2         # float
is_student = True    # bool

print(name, age, height, is_student)
print(type(name))
print(type(age))
print(type(height))
print(type(is_student))
```

Output:

```
Aarav 14 5.2 True
<class 'str'>
<class 'int'>
<class 'float'>
<class 'bool'>
```

The `type()` function tells you the data type of any value — very useful for learning and debugging.

Variables can change value (and even type):

```
score = 10
print(score)      # 10
score = 25
print(score)      # 25
score = "high"    # allowed in Python (dynamic typing)
print(score)      # high
```

Multiple assignment:

```
x, y, z = 1, 2, 3
a = b = c = 0
print(x, y, z, a, b, c)  # 1 2 3 0 0 0
```

Practice Exercises

1. Create variables for your name, class, roll number, and percentage; print them with `type()` of each.
2. Which of these are valid names? `1st_rank`, `first_rank`, `_temp`, `my var`, `for`
3. Swap the values of two variables `a = 5` and `b = 10` (Hint: `a, b = b, a`).
4. Create a variable, print it, change its value twice, and print after each change.

Key Points

- A variable is a named container for a value; `=` assigns a value.
- Core types: `int`, `float`, `str`, `bool`, `None`.
- `type()` reveals the data type.
- Python variables can change type — this is called dynamic typing.

Basic Operators

Understanding

Operators are the **action symbols** of programming — just like $+$ and $-$ in mathematics. If variables are the nouns of a program, operators are the verbs: they *do* things to values.

Definition

An **operator** is a symbol that performs an operation on one or more values (called **operands**).
In $5 + 3$, the operator is $+$ and the operands are 5 and 3 .

1. Arithmetic Operators

```
a, b = 17, 5
print(a + b)    # 22    addition
print(a - b)    # 12    subtraction
print(a * b)    # 85    multiplication
print(a / b)    # 3.4    division (always gives float)
print(a // b)   # 3     floor division (quotient, no decimals)
print(a % b)    # 2     modulus (remainder)
print(a ** b)   # 1419857 exponent (17 to the power 5)
```

Real use of `%`: check even/odd — a number is even if `number % 2 == 0`.

2. Comparison Operators (result is always True/False)

```
x, y = 10, 20
print(x == y)   # False    equal to
print(x != y)   # True     not equal to
print(x > y)    # False    greater than
print(x < y)    # True     less than
print(x >= 10)  # True     greater than or equal
print(y <= 15)  # False    less than or equal
```

3. Logical Operators

```
age = 16
marks = 85
print(age > 15 and marks > 80)    # True (both must be True)
print(age > 18 or marks > 80)     # True (any one True is enough)
print(not (age > 15))              # False (reverses the result)
```

4. Assignment Operators (shortcuts)

```
score = 10
score += 5    # same as score = score + 5 → 15
score -= 3    # → 12
score *= 2    # → 24
score /= 5    # → 4
print(score)  # 4
```

Operator Precedence (who goes first)

Just like BODMAS in maths: `**` first, then `*` / `//` `%`, then `+` `-`, then comparisons, then `not`, `and`, `or`. Use parentheses `()` to make intent clear.

```
print(2 + 3 * 4)    # 14, not 20
print((2 + 3) * 4)  # 20
```

Worked Example: Marks Calculator

```
maths = 78
science = 85
english = 90

total = maths + science + english
percentage = total / 3

print("Total:", total)
print("Percentage:", percentage)
print("Passed all:", maths >= 33 and science >= 33 and english >= 33)
```

Output:

```
Total: 253
Percentage: 84.33333333333333
Passed all: True
```

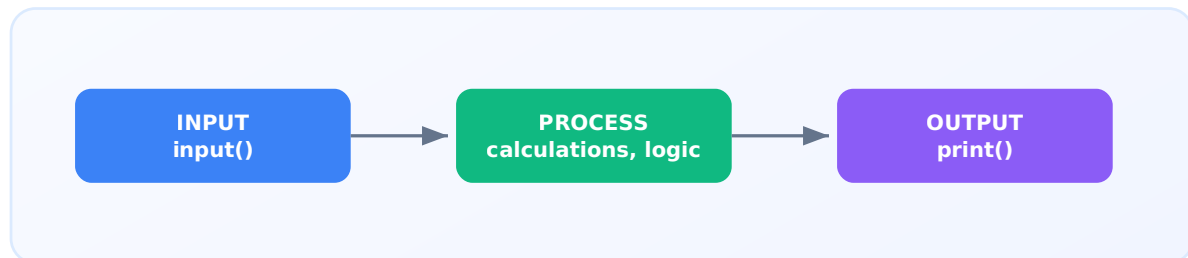
Practice Exercises

1. Take two numbers and print their sum, difference, product, quotient, and remainder.
2. Predict without running: `10 + 4 * 2` , `(10 + 4) * 2` , `10 % 3` , `2 ** 3 ** 2` . Then verify.
3. A shop gives 10% discount. For price = 450, calculate the final amount using operators.
4. Write expressions to check: is `n` divisible by both 3 and 5? Is `n` divisible by 3 or 5 but not both?

Key Points

- Arithmetic: `+` `-` `*` `/` `//` `%` `**` ; `/` always returns float.
- Comparison operators return `True` / `False` .
- Logical: `and` , `or` , `not` combine conditions.
- `x += 1` is shorthand for `x = x + 1` .
- Use parentheses to control order of evaluation.

Input & Output



Every program follows the same rhythm: read data, work on it, show the result

Understanding

A program that cannot talk to the user is like a shopkeeper who never listens to customers. **Input** is how a program listens; **output** is how it speaks. Today your programs become interactive.

Definition

`input()` is a built-in function that pauses the program and reads a line of text typed by the user. `print()` is a built-in function that displays values on the screen. `input()` **always returns a string**, even if the user types a number.

Taking Input

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

Sample run:

```
Enter your name: Priya
Hello, Priya
```

The String Trap (very important!)

```
age = input("Enter your age: ")    # user types 15
print(age + 1)                    # ERROR! Cannot add str and int
```

The fix — convert the input (full details tomorrow):

```
age = int(input("Enter your age: "))
print("Next year you will be", age + 1)
```

Better Output with f-strings

An **f-string** lets you place variables directly inside text using `{ }`:

```
name = "Rohan"
marks = 92
print(f"{name} scored {marks} marks.")
print(f"Half of marks = {marks / 2}")
```

Output:

```
Rohan scored 92 marks.
Half of marks = 46.0
```

f-strings are the modern, recommended way to format output.

print() Extras

```
print("A", "B", "C")           # A B C      (space by default)
print("A", "B", "C", sep="-")  # A-B-C     (custom separator)
print("Loading", end="...")    # stays on same line
print("Done")                  # Loading...Done
print(f"Pi is {3.14159:.2f}")  # Pi is 3.14 (2 decimal places)
```

Worked Example: Simple Interest Calculator

```
p = float(input("Enter principal amount: "))
r = float(input("Enter rate of interest: "))
t = float(input("Enter time in years: "))

si = (p * r * t) / 100
print(f"Simple Interest = {si:.2f}")
print(f"Total Amount    = {p + si:.2f}")
```

Sample run:

```
Enter principal amount: 5000
Enter rate of interest: 8
Enter time in years: 2
Simple Interest = 800.00
Total Amount    = 5800.00
```

Practice Exercises

1. Ask the user's name and city; print: "`<name> lives in <city>`." using an f-string.
2. Input two numbers and print their sum. (Careful — convert them first!)
3. Input the radius of a circle and print its area (`3.14159 * r * r`) rounded to 2 decimals.
4. Ask for a student's three subject marks and print total and percentage neatly formatted.
5. Print `1-2-3-4-5` using a single `print()` with `sep` .

Key Points

- `input()` reads user text; it **always** returns a string.
- Convert with `int()` or `float()` before doing maths.
- f-strings `f"...{variable}..."` are the best way to format output.
- `sep=` and `end=` customize `print()` ; `:.2f` controls decimal places.

Understanding

Sometimes you have money in dollars but the shop accepts rupees — you must **convert** it first. Similarly, data often arrives in the wrong type (remember: `input()` gives strings), and we must convert it before use. This conversion is called **type casting**.

Definition

Type casting (type conversion) is converting a value from one data type to another. **Implicit casting** is done automatically by Python; **explicit casting** is done by the programmer using functions like `int()`, `float()`, `str()`, and `bool()`.

Implicit Casting (automatic)

```
result = 5 + 2.0      # int + float
print(result)         # 7.0
print(type(result))   # <class 'float'>
```

Python automatically upgraded `5` to `5.0` so no information is lost.

Explicit Casting (manual)

```
# str → int
marks = int("85")
print(marks + 5)          # 90

# str → float
price = float("99.50")
print(price * 2)          # 199.0

# int → str (needed for joining with text using +)
age = 16
print("Age is " + str(age)) # Age is 16

# float → int (decimal part is CUT, not rounded)
print(int(7.9))           # 7

# number → bool
print(bool(0))             # False
print(bool(5))             # True
print(bool(""))            # False (empty string)
print(bool("hi"))          # True
```

What Cannot Be Cast

```
int("hello")      # ValueError: invalid literal
int("12.5")       # ValueError (use float("12.5") first)
float("abc")      # ValueError
```

Rule: the string must *look like* the target number.

Worked Example: Age Calculator

```
birth_year = int(input("Enter your birth year: "))
current_year = 2026
age = current_year - birth_year
print(f"You are approximately {age} years old.")
print(f"In 2050 you will be {2050 - birth_year}.")
```

Sample run:

```
Enter your birth year: 2010
You are approximately 16 years old.
In 2050 you will be 40.
```

round() vs int()

```
print(int(7.99))      # 7    (chops off decimals)
print(round(7.99))    # 8    (proper rounding)
print(round(7.456, 2)) # 7.46 (round to 2 decimals)
```

Practice Exercises

1. Convert and print: "250" to int, "3.75" to float, 100 to str, 0 and "0" to bool. Explain the bool results.
2. Ask the user for two numbers and print their average as a float rounded to 1 decimal.
3. Predict the output: `int(9.99)`, `round(9.99)`, `float(7)`, `str(3.14) + "!"`. Verify.
4. Ask for a price as text like "1499", cast it, add 18% GST, and print the final bill.

Key Points

- Casting converts a value's type: `int()`, `float()`, `str()`, `bool()`.
- `input()` results must be cast before arithmetic.
- `int(7.9)` gives 7 (truncation); `round(7.9)` gives 8.
- `bool()` — zero and empty values are `False`; everything else is `True`.

Review of Week 1

This week you learned: what Python is, how to run programs, `print()` and comments, variables and data types, arithmetic/comparison/logical operators, `input()`, f-strings, and type casting. Today you combine all of it.

Rapid Revision Quiz (answer orally or in notebook)

1. Who created Python, and in which year?
2. What is the output of `print(type(3.0))` ?
3. What is the difference between `/` and `//` ?
4. Why does `input("Enter age: ") + 1` cause an error?
5. What does `bool("")` return?
6. Write an f-string that prints `Total = 95.50` from a variable `t = 95.5` .

Mini Project 1: Student Bio-Data & Report Card Generator

Goal: Build a program that collects a student's details, computes results, and prints a formatted report card.

Requirements:

1. Ask for: name, class, roll number, and marks in 3 subjects (out of 100).
2. Calculate total, percentage, and whether the student passed every subject (pass mark 33).
3. Print a neatly formatted report card using f-strings.

Complete Solution:

```

# report_card.py - Week 1 Mini Project
print("==== STUDENT REPORT CARD GENERATOR =====")

name = input("Student name      : ")
s_class = input("Class          : ")
roll = input("Roll number      : ")

maths = float(input("Marks in Maths      : "))
science = float(input("Marks in Science : "))
english = float(input("Marks in English : "))

total = maths + science + english
percentage = total / 3
all_passed = maths >= 33 and science >= 33 and english >= 33

print()
print("=" * 40)
print(f"      REPORT CARD - {name.upper()}")
print("=" * 40)
print(f"Class      : {s_class}")
print(f"Roll No.   : {roll}")
print("-" * 40)
print(f"Maths      : {maths:6.1f} / 100")
print(f"Science    : {science:6.1f} / 100")
print(f"English    : {english:6.1f} / 100")
print("-" * 40)
print(f"Total      : {total:6.1f} / 300")
print(f"Percent    : {percentage:6.2f} %")
print(f"Result     : {'PASS' if all_passed else 'FAIL'}")
print("=" * 40)

```

Sample run:

===== STUDENT REPORT CARD GENERATOR =====

Student name : Ananya Sharma
Class : 9-B
Roll number : 21
Marks in Maths : 88
Marks in Science : 91
Marks in English : 79

=====

REPORT CARD - ANANYA SHARMA

=====

Class : 9-B
Roll No. : 21

Maths : 88.0 / 100
Science : 91.0 / 100
English : 79.0 / 100

Total : 258.0 / 300
Percent : 86.00 %
Result : PASS

=====

Extension Challenges (for fast finishers)

1. Add a fourth and fifth subject.
2. Show a grade: 90+ → A+, 75+ → A, 60+ → B, 33+ → C, else F. (Preview of Week 2's if-else!)
3. Ask for the maximum marks per subject instead of assuming 100.

Teacher's Assessment Rubric

Criterion	Marks
Program runs without errors	4
Correct calculations	3
Proper type casting of inputs	2
Neat, formatted output	1
Total	10

Key Points

- Real programs combine input, casting, calculation, and formatted output.
- Breaking a problem into steps (input → process → output) is the heart of programming.
- Next week: making programs *decide* and *repeat* — control flow!

Days 8-14

WEEK 2: Control Flow

Teach programs to make decisions (`if` , `if-else` , `elif`) and to repeat work (`for` , `while` loops), including `break` , `continue` , `pass` , and nested loops. By Day 14 students build a menu-driven mini project.

If Statements

Understanding

Life is full of decisions: *If it rains, take an umbrella. If marks ≥ 33 , you pass.* Until now our programs ran every line, top to bottom, no matter what. The `if` statement gives programs the power to **choose** whether to run a block of code.

Definition

The **if statement** is a conditional statement that executes a block of code **only when its** condition evaluates to `True`. The block is marked by a colon `:` and **indentation** (spaces at the start of the line).

Syntax

```
if condition:
    statement 1    # runs only if condition is True
    statement 2
statement 3        # always runs (outside the if)
```

Indentation is grammar in Python. Use exactly 4 spaces. Wrong indentation = `IndentationError` or wrong logic.

Examples

```
age = int(input("Enter your age: "))

if age >= 18:
    print("You are eligible to vote.")
    print("Please carry your ID card.")

print("Thank you!")    # runs in every case
```

Sample run 1:

```
Enter your age: 20
You are eligible to vote.
Please carry your ID card.
Thank you!
```

Sample run 2:

```
Enter your age: 15
Thank you!
```

Multiple independent ifs:

```
temp = 41
if temp > 40:
    print("Heat wave alert!")
if temp > 35:
    print("Drink plenty of water.")
```

Output:

```
Heat wave alert!
Drink plenty of water.
```

Both conditions were checked separately; both were `True`, so both blocks ran.

Conditions with logical operators:

```
marks = 95
attendance = 80
if marks >= 90 and attendance >= 75:
    print("Eligible for scholarship")
```

Truthiness in conditions

Any value can act as a condition: non-zero numbers and non-empty strings count as `True`.

```
name = input("Enter name: ")
if name:                               # True if user typed something
    print("Hello,", name)
if not name:
    print("You entered nothing!")
```

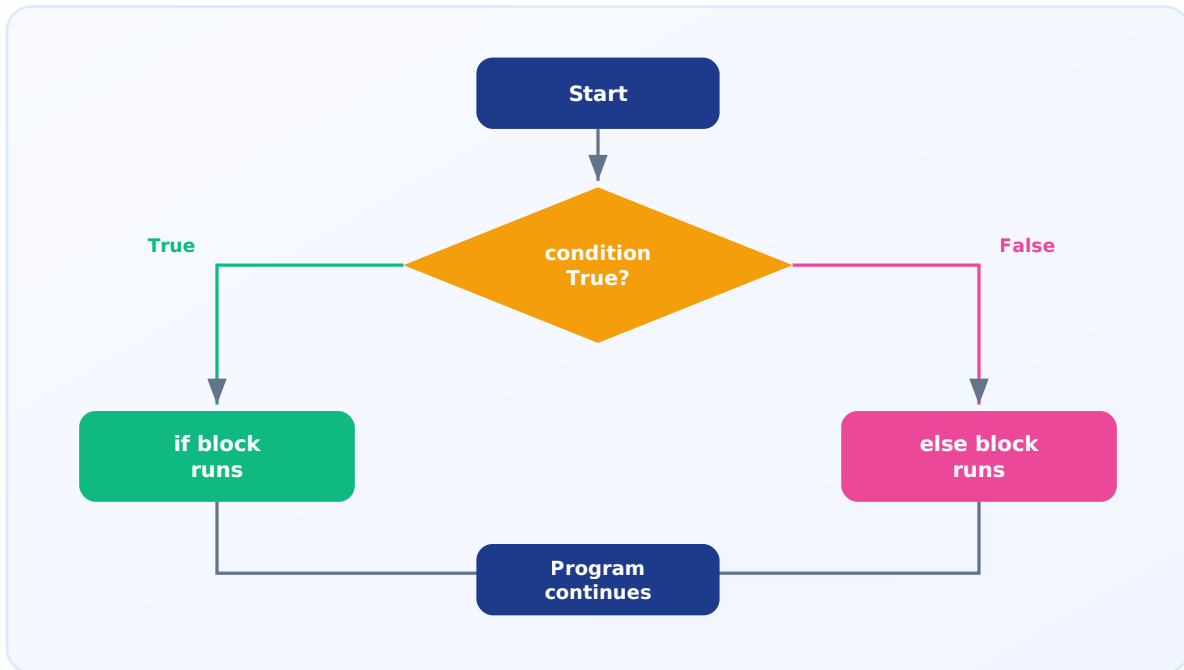
Practice Exercises

1. Ask for a number; if it is positive, print `"Positive"`.
2. Ask for a number; if it is even, print `"Even"` (use `%`).
3. Ask for marks; if marks ≥ 33 print `"Pass"`, and if marks ≥ 90 additionally print `"Excellent!"`.
4. Ask for a password; if it equals `"python123"`, print `"Access granted"`.
5. Find the error:

```
if age >= 18
    print("Adult")
```

Key Points

- `if` runs a block only when the condition is `True`.
- The colon `:` and 4-space indentation define the block.
- Conditions use comparison (`==` , `>` ...) and logical (`and` , `or` , `not`) operators.
- `=` assigns; `==` compares. Mixing them is a classic beginner bug.



The if-else decision: exactly one of the two paths runs

Understanding

Yesterday's programs could act when a condition was true — but stayed silent otherwise. Real decisions have **two or more paths**: pass *or* fail; positive, negative, *or* zero. `else` handles “otherwise”; `elif` (short for *else if*) handles “or maybe this instead”.

Definition

`if-else` provides two branches: the `if` block runs when the condition is `True`; the `else` block runs when it is `False`. `elif` allows checking multiple conditions in sequence; the **first** true condition's block runs and the rest are skipped.

Syntax

```
if condition1:
    # runs if condition1 is True
elif condition2:
    # runs if condition1 False and condition2 True
elif condition3:
    # ...
else:
    # runs if ALL conditions above are False
```

Examples

Two-way decision:

```
num = int(input("Enter a number: "))
if num % 2 == 0:
    print(f"{num} is Even")
else:
    print(f"{num} is Odd")
```

Multi-way decision — grading system:

```
marks = float(input("Enter percentage: "))

if marks >= 90:
    grade = "A+"
elif marks >= 75:
    grade = "A"
elif marks >= 60:
    grade = "B"
elif marks >= 33:
    grade = "C"
else:
    grade = "F"

print(f"Grade: {grade}")
```

Sample runs:

```
Enter percentage: 82
Grade: A

Enter percentage: 30
Grade: F
```

Notice: for 82, Python checked `82 >= 90` (False), then `82 >= 75` (True) — printed A and **skipped** the rest. Order matters! If you check `>= 33` first, everyone above 33 gets a C.

Nested if (an if inside an if):

```
age = int(input("Age: "))
has_id = input("Do you have ID? (yes/no): ")

if age >= 18:
    if has_id == "yes":
        print("Entry allowed")
    else:
        print("Please bring your ID")
else:
    print("Sorry, adults only")
```

Conditional expression (one-line if-else):

```
result = "Pass" if marks >= 33 else "Fail"
print(result)
```

Worked Example: Simple Calculator

```
a = float(input("First number : "))
b = float(input("Second number: "))
op = input("Operation (+ - * /): ")

if op == "+":
    print("Result:", a + b)
elif op == "-":
    print("Result:", a - b)
elif op == "*":
    print("Result:", a * b)
elif op == "/":
    if b != 0:
        print("Result:", a / b)
    else:
        print("Error: cannot divide by zero")
else:
    print("Invalid operation")
```

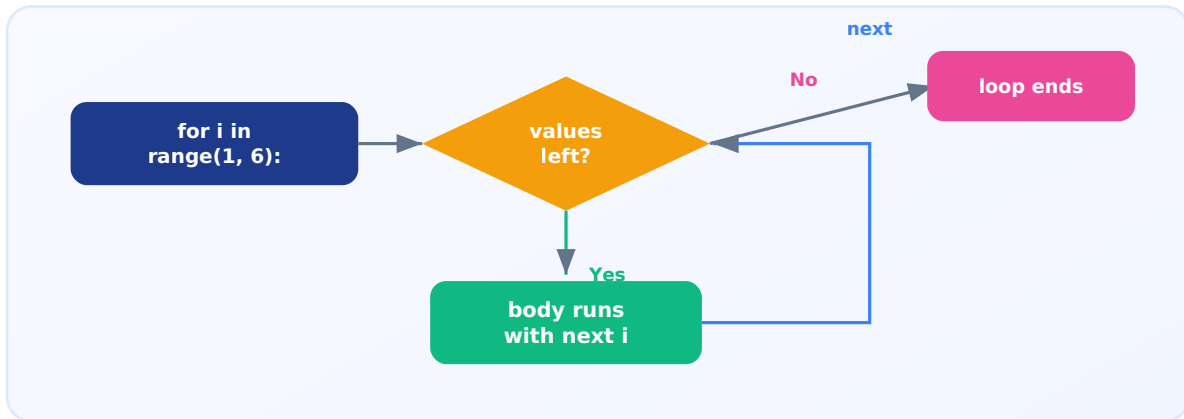
Practice Exercises

1. Input a number and print whether it is Positive, Negative, or Zero.
2. Input three numbers and print the largest (use if-elif-else).
3. Input a year and check leap year: divisible by 4 and (not by 100 unless also by 400).
4. BMI calculator: input weight (kg) and height (m); BMI = weight / height². Classify: <18.5 Underweight, <25 Normal, <30 Overweight, else Obese.
5. Rewrite exercise 1 using nested ifs, then discuss which version is cleaner.

Key Points

- `else` = “in all other cases”; it needs no condition.
- `elif` chains conditions; only the **first true** branch runs.
- Order your conditions from most specific to least specific.
- Nested ifs work but deep nesting is hard to read — prefer `elif` or logical operators.

Loops (For, While)



The for loop takes each value from the sequence, one per round

Understanding

Suppose your teacher says: “Write *I will practice Python* 100 times.” Would you write 100 `print()` lines? Loops let the computer do the repetition. There are two kinds:

- **for loop** — when you know *how many times* to repeat (count-controlled). Like doing exactly 20 push-ups.
- **while loop** — when you repeat *as long as* a condition holds (condition-controlled). Like stirring the pot *until* the milk boils.

Definition

A **loop** repeatedly executes a block of code. A **for loop** iterates over a sequence (like `range()`, a string, or a list). A **while loop** repeats as long as its condition remains `True`.

The range() Function

`range()` generates a sequence of numbers:

```

range(5)           # 0, 1, 2, 3, 4           (start=0, stop before 5)
range(1, 6)        # 1, 2, 3, 4, 5          (start, stop)
range(2, 11, 2)     # 2, 4, 6, 8, 10        (start, stop, step)
range(10, 0, -1)    # 10, 9, ..., 1         (counting down)
  
```

For Loop Examples

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

Output:

```
Round 1  
Round 2  
Round 3  
Round 4  
Round 5
```

Multiplication table — the classic:

```
n = int(input("Which table? "))  
for i in range(1, 11):  
    print(f"{n} x {i} = {n * i}")
```

Sample run (n = 7):

```
7 x 1 = 7  
7 x 2 = 14  
...  
7 x 10 = 70
```

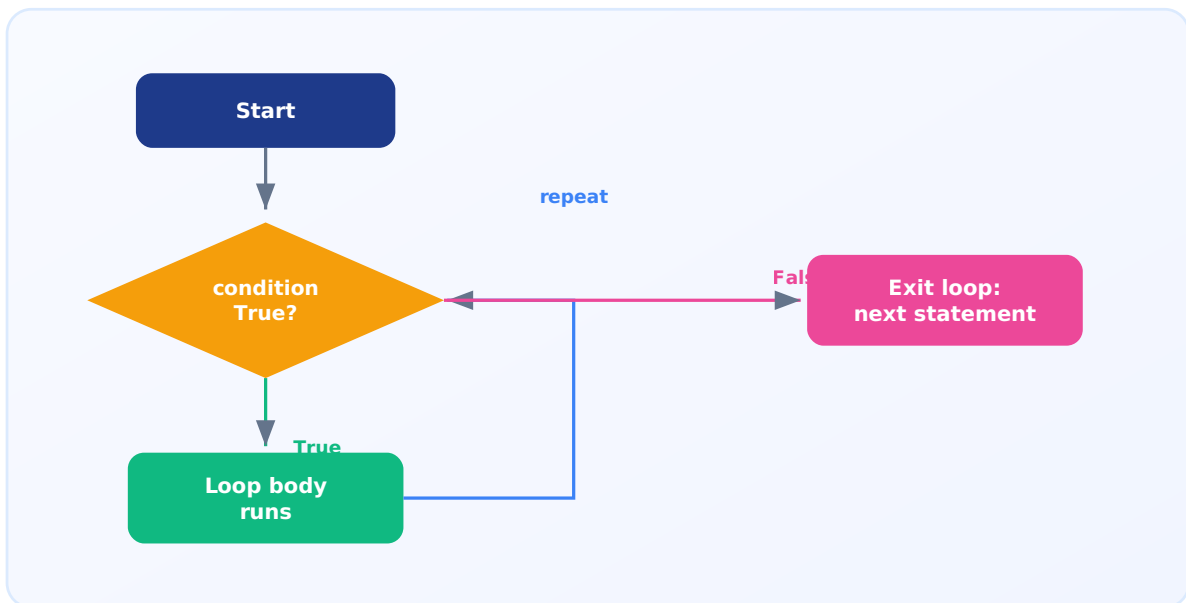
Looping over a string:

```
for ch in "PYTHON":  
    print(ch, end=" ")  
# Output: P Y T H O N
```

Sum of first N numbers (accumulator pattern — very important):

```
n = int(input("N = "))  
total = 0  
for i in range(1, n + 1):  
    total += i  
print("Sum =", total)    # for N=10 → 55
```

While Loop Examples



The while loop: check, run, and come back — until the condition turns False

```
count = 1
while count <= 5:
    print("Count is", count)
    count += 1      # WITHOUT this line → infinite loop!
print("Loop finished")
```

While for input validation (repeat until correct):

```
password = input("Enter password: ")
while password != "python123":
    print("Wrong! Try again.")
    password = input("Enter password: ")
print("Welcome!")
```

Digit sum using while:

```
num = int(input("Number: "))
digit_sum = 0
while num > 0:
    digit_sum += num % 10  # take last digit
    num //= 10            # remove last digit
print("Sum of digits:", digit_sum)  # 345 → 12
```

For vs While — Choosing

Situation	Use
Print table of 7 (fixed 10 times)	<code>for</code>
Keep asking until password correct	<code>while</code>
Process each character of a string	<code>for</code>
Game loop: run until player quits	<code>while</code>

Practice Exercises

1. Print numbers 1–20, then 20–1, then all even numbers from 2–50 (three loops).
2. Print the multiplication table of any number in the format `n x i = result`.
3. Input N; print the sum of squares $1^2 + 2^2 + \dots + N^2$.
4. Use a while loop to find how many times 2 divides 96 evenly ($96 \rightarrow 48 \rightarrow 24 \rightarrow \dots$).
5. Input a number and count its digits using a while loop.
6. Factorial: input N and compute $N! = 1 \times 2 \times \dots \times N$.

Key Points

- `for` = known number of repetitions; `while` = repeat while a condition is true.
- `range(start, stop, step)` — stop is **not included**.
- Accumulator pattern: initialize before the loop, update inside the loop.
- A `while` loop must change something each round, or it never ends.

Break, Continue & Pass

Understanding

Sometimes during repetition you need special controls — like a TV remote for loops:

- **break** — the power button: stop the loop immediately.
- **continue** — the skip button: skip the rest of *this* round, jump to the next.
- **pass** — a blank placeholder: “do nothing, but the syntax needs something here.”

Definition

break terminates the nearest enclosing loop at once. **continue** skips the remaining statements of the current iteration and moves to the next iteration. **pass** is a null statement used as a placeholder where a statement is syntactically required.

break Examples

```
for i in range(1, 11):
    if i == 6:
        break
    print(i, end=" ")
# Output: 1 2 3 4 5
```

Searching (stop at first find):

```
secret = 7
for guess in range(1, 101):
    if guess == secret:
        print("Found it at", guess)
        break
```

Menu loop — run until user quits:

```
while True:
    # deliberately infinite
    cmd = input("Type 'quit' to exit: ")
    if cmd == "quit":
        print("Goodbye!")
        break
    print("You typed:", cmd)
```

`while True` + `break` is one of the most common real-world patterns.

continue Examples

```
for i in range(1, 11):
    if i % 2 == 0:
        continue          # skip even numbers
    print(i, end=" ")
# Output: 1 3 5 7 9
```

Skipping invalid data:

```
marks_list = "45,abc,88,72,xyz,90"
for item in marks_list.split(","):
    if not item.isdigit():
        continue
    print("Valid mark:", item)
```

pass Examples

```
for i in range(5):
    pass          # loop body will be written later; no error

if score > 100:
    pass          # TODO: handle bonus later
else:
    print("Normal score")
```

Without `pass`, an empty block causes `IndentationError`. Teachers' tip: use `pass` when sketching program structure before filling in logic.

The else Clause on Loops (bonus)

A loop can have an `else` that runs **only if the loop finished without `break`**:

```
n = int(input("Number: "))
for i in range(2, n):
    if n % i == 0:
        print("Not prime, divisible by", i)
        break
else:
    print(n, "is prime")
```

Practice Exercises

1. Print 1-50 but stop completely when you reach the first number divisible by both 6 and 7.
2. Print 1-20 skipping all multiples of 3.

3. Keep asking the user for numbers; add them to a running total; stop when they enter `0` ; then print the total.
4. Ask for numbers; ignore negatives using `continue` ; stop on `0` using `break` ; print the sum of the accepted numbers.
5. Use the loop-`else` pattern to check whether an input number is prime.

Key Points

- `break` exits the loop entirely; `continue` skips to the next iteration.
- `while True:` with `break` is the standard “menu/game loop” pattern.
- `pass` does nothing — a legal placeholder for empty blocks.
- Loop `else` runs only when the loop was not broken.

Understanding

A clock has a minute hand and an hour hand. For each single movement of the hour hand, the minute hand completes a *full round*. That is a **nested loop**: a loop inside another loop. For every one iteration of the **outer** loop, the **inner** loop runs completely.

Nested loops are the key to grids, tables, patterns, and comparing every item with every other item.

Definition

A **nested loop** is a loop placed inside the body of another loop. If the outer loop runs m times and the inner loop runs n times per outer iteration, the inner body executes $m \times n$ times in total.

Watch It Work

```
for i in range(1, 4):           # outer: rows
    for j in range(1, 4):       # inner: columns
        print(f"i={i}, j={j}")
```

Output:

```
i=1, j=1
i=1, j=2
i=1, j=3
i=2, j=1
i=2, j=2
i=2, j=3
i=3, j=1
i=3, j=2
i=3, j=3
```

See the pattern: `i` stays fixed while `j` completes its full run.

Star Patterns (the classic teaching tool)

Square:

```
for i in range(4):
    for j in range(5):
        print("*", end=" ")
    print()           # newline after each row
```

```
*****
*****
*****
*****
```

Right triangle:

```
n = 5
for i in range(1, n + 1):
    print("*" * i)
```

```
*
**
***
****
*****
```

Number pyramid:

```
n = 5
for i in range(1, n + 1):
    print(" " * (n - i), end=" ")
    for j in range(1, i + 1):
        print(j, end=" ")
    print()
```

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

All Multiplication Tables at Once

```
for n in range(2, 6):
    print(f"--- Table of {n} ---")
    for i in range(1, 11):
        print(f"{n} x {i} = {n * i}")
    print()
```

break in Nested Loops

`break` only exits the **innermost** loop:

```
for i in range(1, 4):
    for j in range(1, 6):
        if j == 3:
            break          # exits inner loop only
        print(i, j)
```

Practice Exercises

1. Print a 6 × 6 square of `#` characters.
2. Print an inverted right triangle (5 stars, then 4, then 3...).
3. Print this pattern for $n = 4$:

```
1
2 2
3 3 3
4 4 4 4
```

1. Print all pairs `(i, j)` with $1 \leq i, j \leq 4$ where $i \neq j$.
2. Print tables of 12 to 15, each labeled, using nested loops.
3. Challenge: print a full pyramid of stars with $n = 5$ rows.

Key Points

- Inner loop completes fully for **each** outer iteration.
- Total inner-body runs = outer count × inner count.
- Patterns: outer loop = rows, inner loop = columns, `print()` for the newline.
- `break` affects only the loop it is directly inside.

Practice Problems (Control Flow)

Understanding

Today is a **lab day**. No new concepts — instead, students solve graded problems that mix conditions and loops. Struggling and debugging today is the real learning. Teachers: let students attempt each problem for 10–15 minutes before discussing the solution.

Level 1: Warm-Up

Problem 1 — Odd/Even counter. Input 5 numbers (use a loop); count how many are odd and how many even.

```
odd = even = 0
for i in range(5):
    n = int(input(f"Number {i+1}: "))
    if n % 2 == 0:
        even += 1
    else:
        odd += 1
print(f"Even: {even}, Odd: {odd}")
```

Problem 2 — FizzBuzz (a famous interview question). Print 1–30, but print **Fizz** for multiples of 3, **Buzz** for multiples of 5, and **FizzBuzz** for multiples of both.

```
for i in range(1, 31):
    if i % 15 == 0:
        print("FizzBuzz")
    elif i % 3 == 0:
        print("Fizz")
    elif i % 5 == 0:
        print("Buzz")
    else:
        print(i)
```

Why check 15 first? Order of conditions matters — discuss in class.

Level 2: Core Practice

Problem 3 — Reverse a number. 1234 → 4321.

```

num = int(input("Number: "))
rev = 0
while num > 0:
    rev = rev * 10 + num % 10
    num //= 10
print("Reversed:", rev)

```

Problem 4 — Palindrome number. A number that reads the same backwards (121, 1331). Combine Problem 3 with a comparison of the original and reversed values.

Problem 5 — Prime check.

```

n = int(input("Number: "))
if n < 2:
    print("Not prime")
else:
    for i in range(2, int(n ** 0.5) + 1):
        if n % i == 0:
            print("Not prime")
            break
    else:
        print("Prime")

```

Problem 6 — Fibonacci series (each term = sum of previous two): 0 1 1 2 3 5 8 ...

```

n = int(input("How many terms? "))
a, b = 0, 1
for _ in range(n):
    print(a, end=" ")
    a, b = b, a + b

```

Level 3: Challenge

Problem 7 — Number guessing game.

```

import random
secret = random.randint(1, 50)
attempts = 0

while True:
    guess = int(input("Guess (1-50): "))
    attempts += 1
    if guess < secret:
        print("Too low!")
    elif guess > secret:
        print("Too high!")
    else:
        print(f"Correct! You took {attempts} attempts.")
        break

```

Problem 8 — Armstrong numbers. A 3-digit number equal to the sum of cubes of its digits ($153 = 1^3 + 5^3 + 3^3$). Print all Armstrong numbers from 100–999.

Problem 9 — GCD of two numbers using a while loop (Euclid’s method: repeatedly replace the larger with the remainder).

Homework

1. Sum of all numbers between two inputs `a` and `b`.
2. Count vowels in a string entered by the user (loop over the string).
3. Print all prime numbers between 1 and 100.
4. A shop billing loop: keep asking item price until user enters 0; then print total, and apply 5% discount if total > 1000.

Key Points

- Reading a problem → planning steps → coding → testing is the professional workflow.
- The accumulator, counter, and flag patterns solve most beginner problems.
- FizzBuzz, palindrome, prime, and Fibonacci are must-know classics.

Review of Week 2

You can now make programs decide (`if/elif/else`) and repeat (`for` , `while`), control loops (`break` , `continue` , `pass`), and combine loops (nesting). That is the complete toolkit of **control flow** — the logic muscles of programming.

Rapid Revision Quiz

1. What is the output of `for i in range(2, 10, 3): print(i)` ?
2. When does an infinite loop occur? How do you deliberately create a safe one?
3. Difference between `break` and `continue` in one line each.
4. In nested loops, how many times does the inner body run if outer runs 4 times and inner runs 6 times?
5. Why must `elif marks >= 90` come *before* `elif marks >= 60` in a grading chain?

Mini Project 2: ATM Machine Simulator

Goal: Simulate an ATM with a PIN check, a repeating menu, and balance operations — a real menu-driven program using everything from Weeks 1-2.

Requirements:

1. Ask for a 4-digit PIN; give maximum 3 attempts (loop + break).
2. Menu: 1) Check balance 2) Deposit 3) Withdraw 4) Exit — repeat until Exit.
3. Withdrawal must not exceed balance; deposits must be positive.

Complete Solution:

```

# atm.py - Week 2 Mini Project
CORRECT_PIN = "4321"
balance = 5000.0

# --- PIN verification: max 3 attempts ---
attempts = 0
logged_in = False
while attempts < 3:
    pin = input("Enter 4-digit PIN: ")
    if pin == CORRECT_PIN:
        logged_in = True
        break
    attempts += 1
    print(f"Wrong PIN. Attempts left: {3 - attempts}")

if not logged_in:
    print("Card blocked. Contact your bank.")
else:
    print("\nWelcome to PyBank ATM!")
    while True:
        print("\n----- MENU -----")
        print("1. Check Balance")
        print("2. Deposit")
        print("3. Withdraw")
        print("4. Exit")
        choice = input("Choose (1-4): ")

        if choice == "1":
            print(f"Your balance is Rs. {balance:.2f}")
        elif choice == "2":
            amt = float(input("Amount to deposit: "))
            if amt > 0:
                balance += amt
                print(f"Deposited. New balance: Rs. {balance:.2f}")
            else:
                print("Amount must be positive.")
        elif choice == "3":
            amt = float(input("Amount to withdraw: "))
            if amt <= 0:
                print("Amount must be positive.")
            elif amt > balance:
                print("Insufficient balance!")
            else:
                balance -= amt
                print(f>Please collect cash. Balance: Rs. {balance:.2f}")
        elif choice == "4":
            print("Thank you for banking with PyBank!")
            break
        else:
            print("Invalid choice. Try 1-4.")

```

Sample run (shortened):

```

Enter 4-digit PIN: 1111
Wrong PIN. Attempts left: 2
Enter 4-digit PIN: 4321

Welcome to PyBank ATM!

----- MENU -----
1. Check Balance
2. Deposit
3. Withdraw
4. Exit
Choose (1-4): 3
Amount to withdraw: 1200
Please collect cash. Balance: Rs. 3800.00

```

Extension Challenges

1. Add a “Change PIN” option (ask old PIN first).
2. Keep a count of transactions and show it on exit.
3. Add a mini statement: remember the last 5 actions as text (preview of lists — Week 3!).

Teacher’s Assessment Rubric

Criterion	Marks
PIN logic with 3 attempts works	3
Menu repeats correctly and exits cleanly	3
Balance validations (negative, insufficient)	3
Code readability (names, structure)	1
Total	10

Key Points

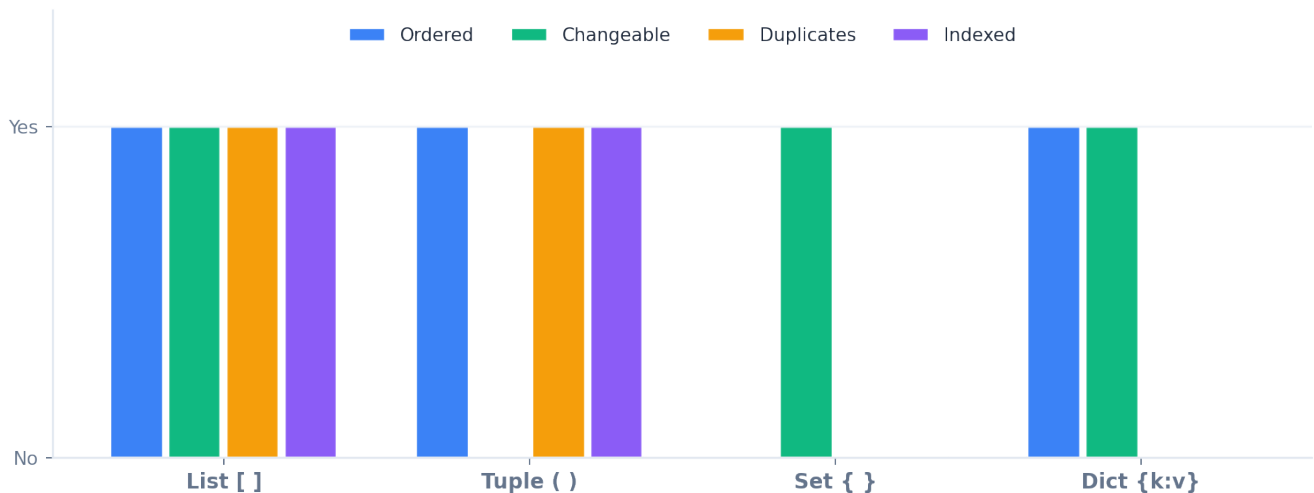
- Menu-driven programs = `while True` + `if/elif` on the user’s choice + `break` to exit.
- Validating input before acting on it is the mark of a professional program.
- Next week: storing many values together — lists, tuples, sets, dictionaries, and functions!

Days 15-21

WEEK 3: Data Structures & Functions

Store and organize collections of data using lists, tuples, sets, and dictionaries; then package reusable logic into functions with parameters and return values. Day 21 combines everything in a contact book mini project.

The Four Containers at a Glance



Understanding

So far, one variable holds one value. But real data comes in groups: marks of 40 students, items in a shopping cart, names in an attendance register. A **list** is Python's ordered, changeable container that holds many values under one name — like a train where each coach has a number (its **index**) and carries one passenger (its **value**).

Definition

A **list** is an ordered, mutable (changeable) collection of items, written inside square brackets `[ ]`, separated by commas. Items can be of any type and can repeat. Indexing starts at **0**.

Creating and Indexing

```
fruits = ["apple", "banana", "mango", "orange", "grapes"]
marks = [78, 92, 65, 88]
mixed = ["Ravi", 16, 5.4, True]
empty = []

print(fruits[0])    # apple      (first item)
print(fruits[2])    # mango
print(fruits[-1])   # grapes     (negative index: from the end)
print(fruits[-2])   # orange
print(len(fruits))  # 5          (number of items)
```

Index map for `fruits` :

Index	0	1	2	3	4
Value	apple	banana	mango	orange	grapes
Negative	-5	-4	-3	-2	-1

Slicing (taking a portion)

```
nums = [10, 20, 30, 40, 50, 60]
print(nums[1:4])    # [20, 30, 40]    index 1 to 3 (4 excluded)
print(nums[:3])     # [10, 20, 30]   from start
print(nums[3:])     # [40, 50, 60]   till end
print(nums[::2])    # [10, 30, 50]   every 2nd item
print(nums[::-1])   # [60, 50, 40, 30, 20, 10] reversed!
```

Changing Lists (mutability)

```
cart = ["pen", "book"]
cart[0] = "pencil"      # replace by index
cart.append("eraser")   # add at end
cart.insert(1, "scale") # add at position 1
print(cart)             # ['pencil', 'scale', 'book', 'eraser']

cart.remove("book")     # remove by value
last = cart.pop()       # remove & return last item
print(last)            # eraser
print(cart)            # ['pencil', 'scale']
```

Useful List Methods & Functions

```
marks = [78, 92, 65, 88, 92]
print(max(marks))      # 92
print(min(marks))      # 65
print(sum(marks))      # 415
print(marks.count(92)) # 2
print(marks.index(65)) # 2
marks.sort()           # [65, 78, 88, 92, 92]
marks.reverse()        # [92, 92, 88, 78, 65]
print(92 in marks)     # True (membership check)
```

Looping Over Lists

```
students = ["Asha", "Vikram", "Neha"]

for name in students:           # by value
    print("Hello,", name)

for i in range(len(students)):  # by index
    print(i + 1, students[i])

for i, name in enumerate(students, start=1): # both at once
    print(f"{i}. {name}")
```

Worked Example: Class Marks Analysis

```
marks = []
n = int(input("How many students? "))
for i in range(n):
    m = float(input(f"Marks of student {i+1}: "))
    marks.append(m)

print(f"Highest : {max(marks)}")
print(f"Lowest  : {min(marks)}")
print(f"Average : {sum(marks) / len(marks):.2f}")
above_avg = 0
for m in marks:
    if m > sum(marks) / len(marks):
        above_avg += 1
print(f"Above average: {above_avg} students")
```

Practice Exercises

1. Create a list of 5 favourite movies; print the first, last, and middle using indexes.
2. Take 5 numbers from the user into a list; print the list, its sum, and its average.
3. From `nums = [4, 9, 12, 7, 15, 20]` print only the even numbers using a loop.
4. Reverse a list without using `reverse()` or `[::-1]` (loop from the end).
5. Ask for a name and check if it exists in a given attendance list using `in`.
6. Remove all duplicates from `[1, 2, 2, 3, 4, 4, 5]` using a loop and a new list.

Key Points

- Lists: ordered, mutable, allow duplicates; created with `[]`.
- Indexing starts at 0; negative indexes count from the end.
- Slicing `list[start:stop:step]` — stop is excluded.
- Key methods: `append`, `insert`, `remove`, `pop`, `sort`, `reverse`, `count`, `index`.
- `len`, `sum`, `max`, `min`, and `in` work on lists.

Understanding

A list is like a whiteboard — you can erase and rewrite. A **tuple** is like a printed certificate — once created, it cannot be changed. Use tuples for data that should stay fixed: coordinates of a city, days of the week, a date of birth, RGB color values.

Definition

A **tuple** is an ordered, **immutable** (unchangeable) collection of items, written inside parentheses `( )`, separated by commas. It supports indexing and slicing like a list, but no item can be added, removed, or modified after creation.

Creating Tuples

```
point = (4, 5)
days = ("Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun")
person = ("Meera", 17, "Bhopal")
one_item = (5,)          # note the comma! (5) is just the number 5
empty = ()

print(type(one_item))    # <class 'tuple'>
print(type(5))           # <class 'int'> - the comma makes the tuple
```

Accessing (same as lists)

```
days = ("Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun")
print(days[0])           # Mon
print(days[-1])          # Sun
print(days[1:4])         # ('Tue', 'Wed', 'Thu')
print(len(days))         # 7
print("Fri" in days)     # True

for d in days[:5]:
    print(d, end=" ")    # Mon Tue Wed Thu Fri
```

Immutability in Action

```
point = (4, 5)
point[0] = 10      # TypeError: 'tuple' object does not support item assignment
```

You cannot change a tuple — but you can create a **new** one:

```
point = (4, 5)
point = (10, 5)    # allowed: this is a brand-new tuple
```

Tuple Packing & Unpacking (very useful!)

```
student = "Arjun", 16, 88.5      # packing (parentheses optional)
name, age, percent = student      # unpacking into variables
print(name)      # Arjun
print(age)       # 16

a, b = 5, 10
a, b = b, a      # the famous Python swap!
print(a, b)     # 10 5
```

Tuple Methods (only two!)

```
scores = (10, 20, 10, 30, 10)
print(scores.count(10))  # 3
print(scores.index(30))  # 3
print(max(scores), min(scores), sum(scores))  # 30 10 80
```

List vs Tuple

Feature	List	Tuple
Brackets	[]	()
Mutable	Yes	No
Methods	Many	2 (<code>count</code> , <code>index</code>)
Speed / memory	Slower, larger	Faster, smaller
Use for	Data that changes	Fixed data
Can be dict key	No	Yes

Convert between them: `list((1,2,3))` and `tuple([1,2,3])`.

Practice Exercises

1. Create a tuple of 6 subjects; print the first three using slicing.
2. Try changing an element of a tuple; note the exact error message.
3. Store `(name, roll, percentage)` for a student; unpack into three variables and print an f-string sentence.
4. A tuple has `(5, 8, 5, 2, 8, 5)`. Count how many times 5 appears, and find the index of 2.
5. Given two variables `x = 3` and `y = 9`, swap them in one line.
6. Convert the tuple `(3, 1, 2)` to a list, sort it, and convert back to a tuple.

Key Points

- Tuples: ordered, immutable, created with `()`; single-item tuples need a trailing comma.
- Indexing, slicing, `len`, `in`, and loops work exactly like lists.
- Packing/unpacking gives clean multi-assignment and the one-line swap.
- Choose a tuple when the data must never change.

Understanding

Imagine collecting signatures for a petition — each person may sign **only once**, and the order of signatures does not matter. That is a **set**: no duplicates, no fixed order. Sets shine at removing duplicates and answering “what is common / different between two groups?” — just like the Venn diagrams from maths class.

Definition

A **set** is an unordered collection of **unique** items, written inside curly braces `{ }`. Duplicates are automatically removed. Sets are mutable (items can be added/removed), but they cannot be indexed because they have no order.

Creating Sets

```
colors = {"red", "green", "blue"}
nums = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3}
print(nums)           # {1, 2, 3, 4, 5, 6, 9} – duplicates gone, order arbitrary

empty_set = set()      # NOT {} – that creates an empty dictionary!
print(type({}))        # <class 'dict'>
print(type(set()))     # <class 'set'>

letters = set("banana")
print(letters)         # {'b', 'a', 'n'}
```

Basic Operations

```
skills = {"python", "excel"}
skills.add("sql")       # add one item
skills.update(["git", "linux"]) # add many
print(skills)

skills.remove("excel")  # error if not present
skills.discard("java")  # NO error if not present
print("python" in skills) # True – membership test is very fast
print(len(skills))
```

Sets cannot do `skills[0]` — no indexing! Loop instead:

```
for s in skills:
    print(s)
```

Set Mathematics (the superpower)

```
cricket = {"Ravi", "Asha", "Vikram", "Neha"}
football = {"Asha", "Karan", "Neha", "Pooja"}

print(cricket | football)  # union: plays either sport
# {'Ravi', 'Asha', 'Vikram', 'Neha', 'Karan', 'Pooja'}

print(cricket & football)  # intersection: plays BOTH
# {'Asha', 'Neha'}

print(cricket - football)  # difference: only cricket
# {'Ravi', 'Vikram'}

print(cricket ^ football)  # symmetric difference: exactly one sport
# {'Ravi', 'Vikram', 'Karan', 'Pooja'}
```

Method forms: `union()`, `intersection()`, `difference()`, `symmetric_difference()`.

Subset / superset:

```
toppers = {"Asha", "Neha"}
print(toppers.issubset(cricket))  # True
print(cricket.issuperset(toppers))  # True
```

Worked Example: Attendance Comparison

```
monday = {"Ravi", "Asha", "Neha", "Karan", "Pooja"}
tuesday = {"Asha", "Karan", "Vikram", "Pooja"}

print("Present both days :", monday & tuesday)
print("Absent on Tuesday :", monday - tuesday)
print("New on Tuesday    :", tuesday - monday)
print("Total students seen:", len(monday | tuesday))
```

Removing Duplicates from a List (classic trick)

```
marks = [88, 92, 88, 75, 92, 60]
unique = list(set(marks))
print(unique)  # duplicates removed (order may change)
```

Practice Exercises

1. Create a set from the list `[1, 2, 2, 3, 3, 3, 4]` and print it.
2. Two sets: your hobbies and your friend's hobbies. Print common hobbies and hobbies unique to you.
3. Ask the user for 5 numbers; store in a set; report how many *distinct* numbers were entered.
4. Given vowels = `{ 'a', 'e', 'i', 'o', 'u' }` and the letters of a word, find which vowels the word contains (intersection).
5. Explain with code why `{ }` is not an empty set.
6. From two class lists (with duplicates), find students appearing in both classes.

Key Points

- Sets: unordered, unique items, `{ }` ; empty set is `set()` .
- `add` , `update` , `remove` , `discard` modify a set; no indexing.
- Set maths: `|` union, `&` intersection, `-` difference, `^` symmetric difference.
- `list(set(x))` is the quickest duplicate remover.

Understanding

A real dictionary maps a *word* to its *meaning*. You never search page numbers — you search by word. A Python **dictionary** works the same way: it stores **key → value** pairs. Instead of asking “what is at index 3?”, you ask “what is the phone number of *Ravi*?”

Dictionaries are the most used data structure in real projects: contact books, JSON data from websites, game scores, product catalogs.

Definition

A **dictionary** is an unordered, mutable collection of **key-value pairs** written inside curly braces `{key: value}`. Keys must be unique and immutable (strings, numbers, tuples); values can be anything. (Since Python 3.7 insertion order is preserved.)

Creating and Accessing

```
student = {  
    "name": "Ishita",  
    "class": 10,  
    "marks": 91.5,  
    "is_prefect": True  
}  
  
print(student["name"])      # Ishita  
print(student["marks"])    # 91.5  
print(len(student))        # 4  
  
print(student["age"])       # KeyError! key does not exist  
print(student.get("age"))   # None (safe access – no error)  
print(student.get("age", 0)) # 0 (default value if missing)
```

Adding, Updating, Removing

```
student["age"] = 15           # add new pair
student["marks"] = 93.0      # update existing
print(student)

student.pop("is_prefect")    # remove by key
del student["age"]          # another way to remove
print(student)
```

Looping Over Dictionaries

```
prices = {"pen": 10, "notebook": 45, "eraser": 5}

for key in prices:           # keys
    print(key)

for value in prices.values(): # values
    print(value)

for item, cost in prices.items(): # both – most useful!
    print(f"{item} costs Rs. {cost}")
```

Output of the last loop:

```
pen costs Rs. 10
notebook costs Rs. 45
eraser costs Rs. 5
```

Useful Patterns

Membership & totals:

```
print("pen" in prices)      # True (checks KEYS)
print(sum(prices.values())) # 60
print(max(prices, key=prices.get)) # notebook (key with highest value)
```

Counting frequency (must-know pattern):

```
text = "banana"
freq = {}
for ch in text:
    freq[ch] = freq.get(ch, 0) + 1
print(freq)           # {'b': 1, 'a': 3, 'n': 2}
```

Nested dictionaries:

```

school = {
    "10A": {"strength": 42, "teacher": "Mrs. Rao"},
    "10B": {"strength": 39, "teacher": "Mr. Khan"},
}
print(school["10A"]["teacher"])    # Mrs. Rao

```

Worked Example: Marks Lookup System

```

marks = {}
for i in range(3):
    name = input("Student name: ")
    m = float(input("Marks: "))
    marks[name] = m

who = input("\nWhose marks do you want? ")
if who in marks:
    print(f"{who} scored {marks[who]}")
else:
    print("Student not found.")

```

Practice Exercises

1. Make a dictionary of 5 countries → capitals; ask the user for a country and print its capital (handle missing keys with `get`).
2. Store item→price for 4 shop items; print a bill line for each using `.items()` and the grand total.
3. Count the frequency of each word in the sentence "the quick brown fox jumps over the lazy dog the fox" (use `.split()`).
4. Update program 1: allow the user to *add* a new country and capital, then print all pairs.
5. From `marks = {"Ravi": 78, "Asha": 92, "Neha": 85}` find the topper's name.
6. Build a nested dictionary for 2 students, each with name, class, and 3 subject marks; print each student's total.

Key Points

- Dictionaries store key→value pairs; keys are unique.
- Access with `d[key]` ; safe access with `d.get(key, default)` .
- `keys()` , `values()` , `items()` power dictionary loops.
- `freq[k] = freq.get(k, 0) + 1` is the counting pattern.
- Dictionaries + lists together model most real-world data.

Understanding

In a kitchen, once you know the recipe for tea, you don't rewrite it every morning — you just say “*make tea*”. A **function** is a named recipe in code: define the steps once, then **call** it by name whenever needed, with different ingredients (**arguments**) each time.

Functions give us: **reusability** (write once, use many times), **organization** (big problems split into small named pieces), and **easier debugging** (fix the recipe in one place).

Definition

A **function** is a named, reusable block of code that performs a specific task. It is created with the **def** keyword, may accept inputs called **parameters**, and runs only when **called**. Values passed during the call are called **arguments**.

Syntax

```
def function_name(parameter1, parameter2):  
    """Docstring: what this function does."""  
    # body – the steps  
    ...
```

First Functions

```
def greet():  
    print("Hello! Welcome to Python.")  
  
greet()           # calling the function  
greet()           # reuse – no rewriting!
```

Output:

```
Hello! Welcome to Python.  
Hello! Welcome to Python.
```

With parameters:

```
def greet_student(name):
    print(f"Good morning, {name}!")

greet_student("Aisha")      # Good morning, Aisha!
greet_student("Rohit")     # Good morning, Rohit!
```

`name` is the **parameter** (the placeholder); `"Aisha"` is the **argument** (the actual value).

Multiple parameters:

```
def print_table(n, upto):
    for i in range(1, upto + 1):
        print(f"{n} x {i} = {n * i}")

print_table(7, 5)
print_table(12, 3)
```

Default and Keyword Arguments

```
def make_tea(cups=1, sugar="normal"):
    print(f"Making {cups} cup(s) of tea, {sugar} sugar.")

make_tea()                # Making 1 cup(s) of tea, normal sugar.
make_tea(3)               # Making 3 cup(s) of tea, normal sugar.
make_tea(2, "less")       # positional arguments
make_tea(sugar="no", cups=4) # keyword arguments – order free!
```

Docstrings — document your functions

```
def area_rectangle(length, width):
    """Return-free demo: prints the area of a rectangle."""
    print("Area =", length * width)

help(area_rectangle)      # shows the docstring
```

Variable Scope (local vs global)

```
def counter():
    x = 5          # local: exists only inside the function
    print(x)

counter()          # 5
print(x)           # NameError: x is not defined outside
```

Variables created inside a function are **local** — invisible outside. Variables created outside are **global** — readable inside, but avoid modifying them from inside functions (it makes bugs hard to trace).

Practice Exercises

1. Write `welcome()` that prints a 3-line welcome banner; call it twice.
2. Write `square(n)` that prints the square of `n`; call it for 4, 9, and 12.
3. Write `is_even(n)` that prints "Even" or "Odd".
4. Write `bill(amount, gst=18)` that prints the amount plus GST; call it with and without the second argument.
5. Write `pattern(rows, symbol="*")` printing a right triangle of the symbol.
6. Predict and verify: what happens when you call a function with too few arguments?

Key Points

- `def name(params):` defines; `name(args)` calls.
- Parameters receive arguments; defaults make arguments optional.
- Keyword arguments (`sugar="no"`) improve clarity and free the order.
- Local variables live only inside their function (scope).
- Docstrings describe what a function does.

Understanding

Yesterday's functions *printed* results — like a chef who cooks and immediately serves. But often you want the dish **handed back** so you can use it further: add a topping, combine with another dish, store it. The `return` statement hands the result back to the caller.

print vs return — the most important distinction of this week:

- `print()` only *shows* a value on screen. The program cannot use it further.
- `return` *gives back* a value to wherever the function was called. It can be stored, compared, passed to another function.

Definition

The `return` statement ends a function's execution and sends a value back to the caller. A function without a `return` (or with a bare `return`) returns `None`. A function may have multiple `return` statements, but only one executes per call.

Basic Returns

```
def add(a, b):  
    return a + b  
  
result = add(5, 3)           # value comes back and is STORED  
print(result)                # 8  
print(add(10, 20) * 2)       # 60 – returned values can be used in expressions  
  
big = add(add(1, 2), add(3, 4)) # functions feeding functions  
print(big)                   # 10
```

What print-only functions cannot do:

```
def add_print(a, b):  
    print(a + b)  
  
x = add_print(5, 3)          # prints 8, but...  
print(x)                     # None – nothing was returned!
```

return Ends the Function

```
def check_pass(marks):  
    if marks >= 33:  
        return "Pass"  
    return "Fail"           # runs only if the first return didn't  
  
print(check_pass(70))      # Pass  
print(check_pass(20))      # Fail
```

Code after a `return` never runs — Python leaves the function immediately.

Returning Multiple Values (as a tuple)

```
def stats(numbers):  
    return min(numbers), max(numbers), sum(numbers) / len(numbers)  
  
low, high, avg = stats([70, 85, 90, 65])  
print(f"Low={low}, High={high}, Avg={avg:.1f}")  
# Low=65, High=90, Avg=77.5
```

Building a Function Library

```
def area_circle(r):
    """Return area of a circle of radius r."""
    return 3.14159 * r * r

def is_prime(n):
    """Return True if n is prime."""
    if n < 2:
        return False
    for i in range(2, int(n ** 0.5) + 1):
        if n % i == 0:
            return False
    return True

def grade(percent):
    """Return letter grade for a percentage."""
    if percent >= 90: return "A+"
    if percent >= 75: return "A"
    if percent >= 60: return "B"
    if percent >= 33: return "C"
    return "F"

# Using the library
print(area_circle(5))           # 78.53975
print(is_prime(29))             # True
print(grade(81))                # A

primes = []
for n in range(2, 30):
    if is_prime(n):
        primes.append(n)
print(primes)                   # [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

See the power: `is_prime` was written once and reused inside a loop to build a whole list.

Practice Exercises

1. Write `cube(n)` returning n^3 ; use it to print cubes of 1–10.
2. Write `bigger(a, b)` returning the larger number; then `biggest3(a, b, c)` that uses `bigger` twice.
3. Write `total_and_percent(m1, m2, m3)` returning both total and percentage; unpack and print.
4. Write `count_vowels(text)` returning how many vowels a string has.
5. Write `is_leap(year)` returning True/False; use it to print all leap years 2000–2030.
6. Explain the output: `print(print("hi"))`.

Key Points

- `return` sends a value back; `print` only displays it.
- A function with no `return` gives `None`.
- `return` immediately exits the function.
- Multiple values return as a tuple — unpack them.
- Returned values can be stored, compared, and chained into other functions.

Review of Week 3

You now hold the four core containers — **list** (ordered, changeable), **tuple** (ordered, fixed), **set** (unique items), **dictionary** (key→value) — plus **functions** with parameters and return values. This is the toolkit that turns small scripts into real applications.

Rapid Revision Quiz

1. Which container would you pick for: (a) days of the week (b) unique visitor IDs (c) roll number → name mapping (d) a to-do list?
2. What is the output of `list(set([3, 1, 3, 2, 1]))` (any order)?
3. Difference between `d[key]` and `d.get(key)` when the key is missing?
4. What does a function return if it has no `return` statement?
5. Write a one-line swap of variables `a` and `b`.

Mini Project 3: Contact Book Application

Goal: A menu-driven contact book that stores names and phone numbers in a dictionary, with all operations written as functions.

Requirements:

1. Menu: Add, Search, Update, Delete, Show All, Exit.
2. Contacts stored in a dictionary `{name: phone}`.
3. Every operation is its own function; the menu loop only coordinates.

Complete Solution:

```
# contact_book.py – Week 3 Mini Project
```

```
contacts = {}
```

```
def add_contact(name, phone):  
    """Add a contact; return a status message."""  
    if name in contacts:  
        return f"{name} already exists. Use Update instead."  
    contacts[name] = phone  
    return f"Added {name}."
```

```
def search_contact(name):  
    """Return the phone number or a not-found message."""  
    return contacts.get(name, f"No contact named {name}.")
```

```
def update_contact(name, phone):  
    if name not in contacts:  
        return f"{name} not found."  
    contacts[name] = phone  
    return f"Updated {name}."
```

```
def delete_contact(name):  
    if name in contacts:  
        contacts.pop(name)  
        return f"Deleted {name}."  
    return f"{name} not found."
```

```
def show_all():  
    if not contacts:  
        return "Contact book is empty."  
    lines = []  
    for i, (name, phone) in enumerate(sorted(contacts.items()), start=1):  
        lines.append(f"{i}. {name:15} {phone}")  
    return "\n".join(lines)
```

```
# ----- main menu loop -----
```

```
print("===== PY CONTACT BOOK =====")
```

```
while True:
```

```
    print("\n1.Add  2.Search  3.Update  4.Delete  5.Show All  6.Exit")
```

```
    choice = input("Choose: ")
```

```
    if choice == "1":
```

```
        n = input("Name : ")
```

```
        p = input("Phone: ")
```

```
        print(add_contact(n, p))
```

```
    elif choice == "2":
```

```
        print(search_contact(input("Name to search: ")))
```

```
    elif choice == "3":
```

```
        n = input("Name : ")
```

```
        p = input("New phone: ")
```

```
        print(update_contact(n, p))
```

```
    elif choice == "4":
```

```
        print(delete_contact(input("Name to delete: ")))
```

```
    elif choice == "5":
```

```
        print(show_all())
```

```
elif choice == "6":
    print(f"Goodbye! You have {len(contacts)} contact(s) saved.")
    break
else:
    print("Invalid choice.")
```

Sample run (shortened):

```
===== PY CONTACT BOOK =====

1.Add 2.Search 3.Update 4.Delete 5.Show All 6.Exit
Choose: 1
Name : Asha
Phone: 98765xxxxx
Added Asha.

1.Add 2.Search 3.Update 4.Delete 5.Show All 6.Exit
Choose: 5
1. Asha          98765xxxxx
```

Notice the design: functions *return* messages; only the menu loop *prints*. This separation (logic vs display) is professional program structure.

Extension Challenges

1. Store each contact as a nested dictionary: `{"phone": ..., "email": ..., "city": ...}`.
2. Add “Find by city” — loop through values and collect matches in a list.
3. Prevent duplicate phone numbers using a set of used numbers.
4. Add input validation: phone must be 10 digits (`len` + `isdigit`).

Teacher’s Assessment Rubric

Criterion	Marks
All six menu options work	4
Each operation is a proper function with return values	3
Dictionary used correctly (get/pop/items)	2
Clean output formatting	1
Total	10

Key Points

- Choosing the right container is half the design of a program.
- Functions that return values (instead of printing) are reusable everywhere.
- Menu loop + functions + dictionary = the skeleton of countless real apps.
- Next week: modules, files, error handling, and object-oriented programming!

Days 22-28

WEEK 4: Modules, Files & OOP Basics

Use ready-made and self-written modules, read and write files so data survives after the program ends, handle errors gracefully, and take the first confident steps into Object-Oriented Programming. Day 28 builds a persistent quiz application.

Understanding

You don't build a phone from sand — you buy ready-made parts. Programming works the same way: a **module** is a ready-made file of Python code (functions, variables, classes) that you can plug into your program with `import`. Python ships with a huge **standard library** ("batteries included"), and you can also write your own modules.

Definition

A **module** is a file containing Python definitions (functions, variables, classes) that can be reused in other programs via the `import` statement. A **package** is a folder of related modules. The **Python Standard Library** is the collection of modules that comes pre-installed with Python.

Four Ways to Import

```
import math                # 1. whole module
print(math.sqrt(25))       # use with module name: math.____

import math as m           # 2. with an alias
print(m.pi)

from math import sqrt, pi  # 3. specific items only
print(sqrt(49), pi)       # no prefix needed

from math import *         # 4. everything (avoid – pollutes names)
```

Recommended styles are 1–3. Style 4 makes it unclear where names came from.

Star Modules of the Standard Library

math — mathematics:

```
import math
print(math.sqrt(144))      # 12.0
print(math.floor(7.8))    # 7
print(math.ceil(7.2))     # 8
print(math.pow(2, 10))    # 1024.0
print(math.factorial(5))  # 120
print(math.pi)           # 3.141592653589793
```

random — chance and games:

```
import random
print(random.randint(1, 6))      # dice roll: 1..6 inclusive
print(random.random())          # float 0.0..1.0
print(random.choice(["red", "green", "blue"])) # pick one
cards = [1, 2, 3, 4, 5]
random.shuffle(cards)           # shuffle in place
print(cards)
print(random.sample(range(1, 50), 6)) # 6 unique lottery numbers
```

datetime — dates and times:

```
import datetime
today = datetime.date.today()
now = datetime.datetime.now()
print(today)                # 2026-07-19
print(now.strftime("%d-%m-%Y %H:%M")) # 19-07-2026 14:30
birth = datetime.date(2010, 5, 14)
print((today - birth).days, "days old")
```

Others worth knowing: `os` (files/folders), `sys` (interpreter info), `statistics` (mean, median), `time` (`time.sleep(2)` pauses 2 seconds).

Writing Your Own Module

File 1 — `mymaths.py` :

```
# mymaths.py — my own module
def area_circle(r):
    return 3.14159 * r * r

def is_even(n):
    return n % 2 == 0

school = "Sunrise Public School"
```

File 2 — `main.py` (same folder):

```
import mymaths

print(mymaths.area_circle(7))    # 153.93791
print(mymaths.is_even(10))      # True
print(mymaths.school)           # Sunrise Public School
```

Your Week 3 function library can now live in one file and serve every future program!

Worked Example: Dice Game

```
import random

print("=== Dice Battle ===")
you = random.randint(1, 6)
computer = random.randint(1, 6)
print(f"You rolled      : {you}")
print(f"Computer rolled: {computer}")
if you > computer:
    print("You win!")
elif you < computer:
    print("Computer wins!")
else:
    print("It's a draw!")
```

Practice Exercises

1. Using `math`, print the square root, ceiling, and floor of a user-entered decimal.
2. Simulate tossing a coin 10 times with `random.choice(["Heads", "Tails"])`; count each side.
3. Print today's date in the format `19 July 2026` using `strftime("%d %B %Y")`.
4. Create your own module `strings_util.py` with `count_vowels(text)` and `reverse(text)`; import and use it.
5. Lucky draw: pick 3 unique winners from a list of 8 names using `random.sample`.
6. Guess-the-number game: computer picks `randint(1,100)`; user guesses with high/low hints (reuse Day 13 logic).

Key Points

- `import module`, `import module as alias`, `from module import name`.
- Standard library stars: `math`, `random`, `datetime`, `os`, `time`.
- Any `.py` file is a module — import your own utilities.
- Modules turn code into reusable building blocks across projects.

Understanding

Every program so far had amnesia — close it, and all data vanished, because variables live in RAM. **Files** live on the disk and survive shutdowns. File handling lets programs *remember*: save student records, read question banks, write reports.

Working with a file is like using a notebook: **open** it, **read or write**, then **close** it.

Definition

File handling is the process of creating, opening, reading, writing, and closing files from a program. Python's built-in `open(filename, mode)` returns a file object; the **mode** decides the operation: "r" read, "w" write (overwrites!), "a" append, "x" create-new.

File Modes

Mode	Meaning	If file exists	If missing
"r"	Read	reads it	Error
"w"	Write	erases & rewrites	creates it
"a"	Append	adds at end	creates it
"x"	Create	Error	creates it
"r+"	Read + write	reads/writes	Error

The dangerous one is "w" — it silently wipes existing content. Teach this early!

Writing to a File

```
f = open("notes.txt", "w")
f.write("Python is fun.\n")
f.write("Files remember data.\n")
f.close()
print("File written!")
```

A file `notes.txt` now exists next to your program with two lines.

The with Statement (best practice)

`with` opens the file and **closes it automatically**, even if an error occurs:

```
with open("notes.txt", "a") as f:
    f.write("This line was appended.\n")
# no f.close() needed – done automatically
```

Use `with` in all your programs from today.

Reading from a File

```
# 1. Read everything at once
with open("notes.txt", "r") as f:
    content = f.read()
print(content)

# 2. Read line by line (best for big files)
with open("notes.txt", "r") as f:
    for line in f:
        print("LINE:", line.strip()) # strip() removes the trailing \n

# 3. Read all lines into a list
with open("notes.txt", "r") as f:
    lines = f.readlines()
print(lines) # ['Python is fun.\n', 'Files remember data.\n', ...]
```

Worked Example: Student Marks Saver

```
# save marks
n = int(input("How many students? "))
with open("marks.txt", "w") as f:
    for i in range(n):
        name = input("Name : ")
        marks = input("Marks: ")
        f.write(f"{name},{marks}\n")

# read back and report
print("\n--- Saved Records ---")
total = 0
count = 0
with open("marks.txt", "r") as f:
    for line in f:
        name, marks = line.strip().split(",")
        print(f"{name:12} {marks}")
        total += float(marks)
        count += 1
print(f"Class average: {total / count:.2f}")
```

Sample run:

```
How many students? 2
Name : Asha
Marks: 92
Name : Vikram
Marks: 78

--- Saved Records ---
Asha          92
Vikram        78
Class average: 85.00
```

Run the program again with mode `"a"` in the writer, and old records stay — the program now *remembers*.

Checking a File Exists

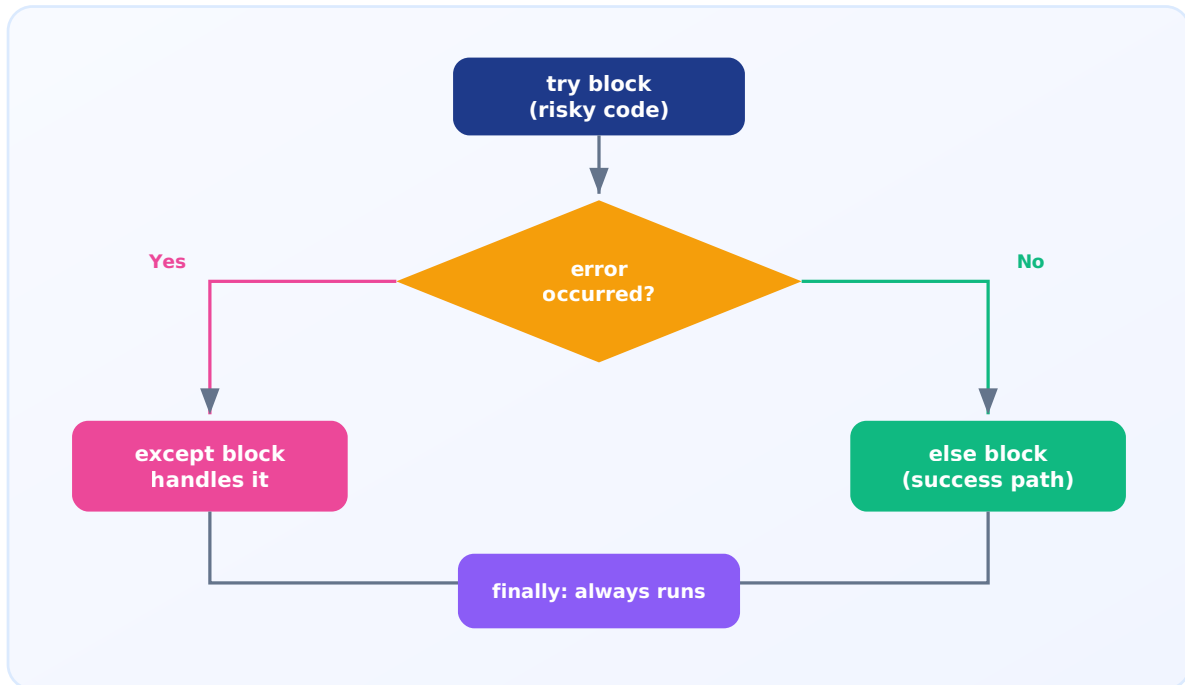
```
import os
if os.path.exists("marks.txt"):
    print("File found")
else:
    print("No data yet")
```

Practice Exercises

1. Write your daily timetable (5 lines) to `timetable.txt`, then read and display it.
2. Append today's date (`datetime`) to a `diary.txt` file each time the program runs.
3. Count the number of lines and words in any text file (`split()` helps).
4. Copy the contents of one file into another (read from A, write to B).
5. Search: ask for a name and print only the lines of `marks.txt` containing it.
6. Explain: what happens to existing content when a file is opened in `"w"` vs `"a"` ?

Key Points

- `open(name, mode)` → file object; modes `r/w/a/x` .
- `"w"` overwrites silently; `"a"` preserves and appends.
- Always prefer `with open(...)` as `f:` — auto-closing, error-safe.
- `read()` , `readlines()` , or `for line in f` for reading; `strip()` cleans newlines.
- Files give programs permanent memory.



Exception handling: the program survives errors instead of crashing

Understanding

A good shopkeeper doesn't shut the shop when a customer pays with a torn note — he politely asks for another. Similarly, a good program doesn't **crash** when the user types "abc" where a number was expected — it handles the problem and continues. Today we turn fragile programs into robust ones.

Three kinds of problems:

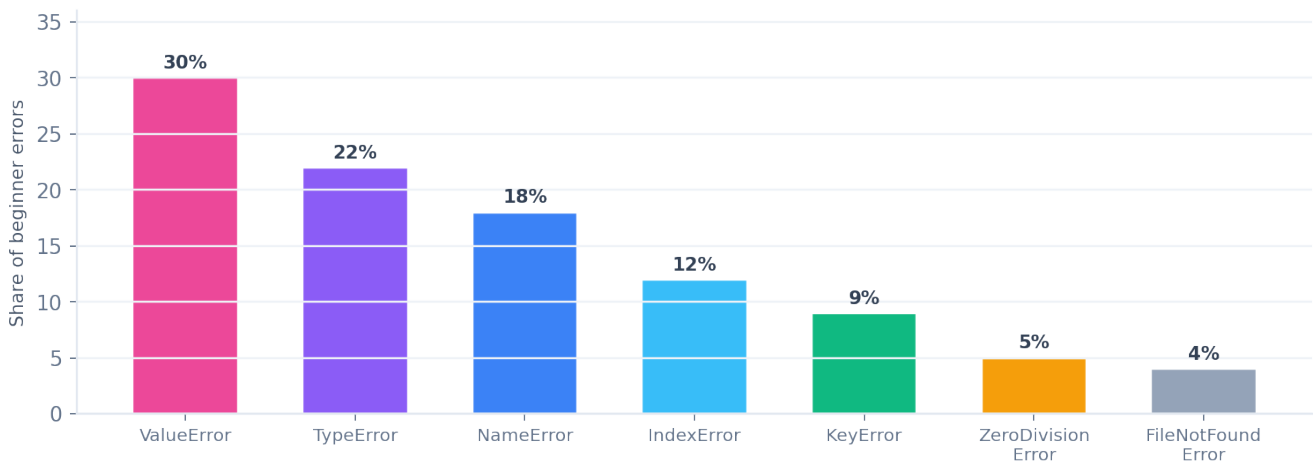
1. **Syntax errors** — grammar mistakes; the program won't even start (`print("hi"` — missing bracket).
2. **Runtime errors (exceptions)** — legal code that fails while running (`10 / 0`).
3. **Logical errors** — runs fine, wrong answer (used `+` instead of `*`). No error message — only testing finds these!

Definition

An **exception** is an error detected during execution that interrupts the normal flow of the program. **Exception handling** uses the **try-except** structure: risky code goes in `try` ; recovery code goes in `except` ; optional `else` runs if no error occurred; optional `finally` always runs.

Common Exceptions

Which Exceptions Will You Meet Most Often?



```
print(10 / 0)           # ZeroDivisionError
print(int("abc"))       # ValueError
print(undefined_var)    # NameError
print("2" + 2)          # TypeError
lst = [1, 2]; lst[5]     # IndexError
d = {}; d["x"]           # KeyError
open("nofile.txt")       # FileNotFoundError
```

Learning to *read* the error name and line number is a superpower — the message literally tells you what and where.

try-except Basics

```
try:
    age = int(input("Enter age: "))
    print("Next year:", age + 1)
except ValueError:
    print("That was not a number!")
print("Program continues...")
```

Sample run:

```
Enter age: hello
That was not a number!
Program continues...
```

Without `try`, the program would have crashed at line 2.

Multiple excepts, else, finally

```
try:
    a = int(input("Numerator : "))
    b = int(input("Denominator: "))
    result = a / b
except ValueError:
    print("Please enter whole numbers only.")
except ZeroDivisionError:
    print("Cannot divide by zero.")
else:
    print("Result:", result)          # runs only if NO exception
finally:
    print("Calculation attempt over.") # ALWAYS runs
```

Catch-all (use sparingly):

```
except Exception as e:
    print("Something went wrong:", e)
```

The Bullet-Proof Input Pattern (memorize this!)

```
while True:
    try:
        marks = float(input("Enter marks (0-100): "))
        if 0 <= marks <= 100:
            break
        print("Marks must be between 0 and 100.")
    except ValueError:
        print("Numbers only, please.")
    print("Accepted:", marks)
```

This loop simply cannot be broken by bad input — every serious program uses it.

raise — creating your own errors

```
def set_age(age):
    if age < 0:
        raise ValueError("Age cannot be negative")
    print("Age set to", age)

set_age(15)      # Age set to 15
set_age(-3)     # ValueError: Age cannot be negative
```

Worked Example: Safe File Reader

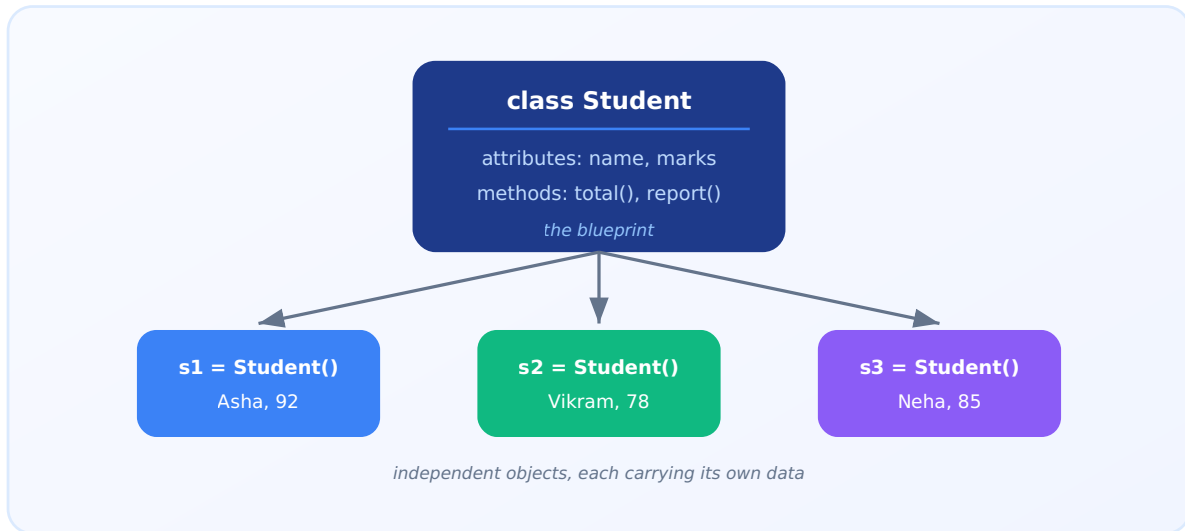
```
filename = input("File to open: ")
try:
    with open(filename, "r") as f:
        print(f.read())
except FileNotFoundError:
    print(f"Sorry, '{filename}' does not exist.")
```

Practice Exercises

1. Ask for two numbers and divide them; handle both `ValueError` and `ZeroDivisionError` with friendly messages.
2. Access index 10 of a 3-item list inside `try`; catch `IndexError`.
3. Wrap Day 18's marks-lookup: catch `KeyError` when using `marks[name]` directly.
4. Use the bullet-proof pattern to force a valid integer between 1 and 6 (a dice guess).
5. Write `withdraw(balance, amount)` that raises `ValueError` if amount exceeds balance; call it inside try-except.
6. Identify the type: (a) `while x > 0` never ending (b) `prnit("hi")` (c) `int("3.5")`.

Key Points

- Syntax errors stop the start; exceptions stop the run; logic errors hide.
- `try` holds risky code; `except ErrorType` handles it; `else` = success path; `finally` = always.
- Catch *specific* exceptions; use `Exception as e` only as a last resort.
- `while True` + `try` = unbreakable input validation.
- `raise` lets your own functions signal invalid situations.



One class (blueprint) creates many objects (instances)

Understanding

Until now we wrote programs as **actions** (functions) working on **separate data** (variables). But look at the real world: a **car** bundles its data (color, speed, fuel) with its actions (start, accelerate, brake) into one object. **Object-Oriented Programming (OOP)** brings that natural bundling into code.

Think of a **class** as a *blueprint* — the architect's drawing of a house. An **object** is an actual *house built* from that blueprint. One blueprint → many houses, each with its own color and furniture.

Definition

Object-Oriented Programming (OOP) is a programming style that organizes code into **classes** (blueprints defining data and behaviour together) and **objects** (individual instances created from a class). The data inside an object are its **attributes**; the functions inside a class are its **methods**.

Why OOP?

Without OOP	With OOP
<code>student1_name</code> , <code>student1_marks</code> , <code>student2_name</code> ... variable explosion	Each student is one object carrying its own data
Functions and data live apart	Data + behaviour travel together
Hard to model real things	Classes map naturally to real-world things

The four pillars (studied deeply in advanced courses): **Encapsulation** (bundle & protect data), **Inheritance** (child classes reuse parent code), **Polymorphism** (same action, different forms), **Abstraction** (hide inner complexity).

First Class and Objects

```
class Dog:
    pass                # empty class for now

d1 = Dog()             # creating an object (instantiation)
d2 = Dog()
print(type(d1))        # <class '__main__.Dog'>
```

Giving objects data and behaviour:

```
class Dog:
    def __init__(self, name, breed):    # constructor
        self.name = name               # attribute
        self.breed = breed             # attribute

    def bark(self):                    # method
        print(f"{self.name} says: Woof! Woof!")

# build two different dogs from one blueprint
dog1 = Dog("Bruno", "Labrador")
dog2 = Dog("Sheru", "Indie")

print(dog1.name)    # Bruno
print(dog2.breed)   # Indie
dog1.bark()         # Bruno says: Woof! Woof!
dog2.bark()         # Sheru says: Woof! Woof!
```

Understanding init and self

- `__init__` is the **constructor** — it runs automatically the moment an object is created, and its job is to set up the object's starting attributes.

- `self` means “this particular object”. When `dog1.bark()` runs, `self` is `dog1`; when `dog2.bark()` runs, `self` is `dog2`. Every method’s first parameter is `self`, and Python passes it automatically — you never type it in the call.

```
dog1.bark()           # what you write
Dog.bark(dog1)        # what Python effectively does
```

Objects Are Independent

```
class Counter:
    def __init__(self):
        self.count = 0
    def click(self):
        self.count += 1

c1 = Counter()
c2 = Counter()
c1.click(); c1.click(); c2.click()
print(c1.count)      # 2
print(c2.count)      # 1 — separate data per object!
```

Practice Exercises

1. Create a class `Book` with attributes `title` and `author`, and a method `info()` printing them; make 3 books.
2. Create a class `Circle` with attribute `radius` and methods `area()` and `perimeter()` that return values.
3. Create `Mobile` with `brand`, `price`; method `apply_discount(percent)` reduces the price. Show it works per-object.
4. Explain in your notebook: class vs object; attribute vs method; what `self` refers to.
5. Predict output before running:

```
class A:
    def __init__(self):
        self.x = 10
a = A(); b = A(); b.x = 99
print(a.x, b.x)
```

Key Points

- Class = blueprint; object = instance built from it.
- `__init__` runs at creation and sets attributes; `self` = this object.
- Attributes store an object's data; methods are its functions.
- Many independent objects can be created from one class.
- OOP models the real world naturally — the foundation of large software.

Understanding

Yesterday you built basic classes. Today we make them practical: default values, class-level data shared by all objects, methods that call other methods, objects stored in lists, and a taste of **inheritance** — the feature that lets a new class reuse an existing one, like a child inheriting family traits.

Definition

An **instance attribute** belongs to one object (`self.name`). A **class attribute** belongs to the class itself and is shared by all its objects. **Inheritance** lets a **child (derived) class** automatically receive the attributes and methods of a **parent (base) class**, adding or overriding what it needs.

Class Attributes vs Instance Attributes

```
class Student:
    school = "Sunrise Public School"    # class attribute: shared

    def __init__(self, name, marks):
        self.name = name                # instance attributes: per object
        self.marks = marks

s1 = Student("Asha", 92)
s2 = Student("Vikram", 78)

print(s1.school)    # Sunrise Public School
print(s2.school)    # Sunrise Public School (same, shared)
print(s1.name, s2.name)    # Asha Vikram (different, own)

Student.school = "Sunrise International"    # change once...
print(s2.school)    # Sunrise International (...changes for all)
```

Methods Working Together

```
class Student:
    def __init__(self, name, m1, m2, m3):
        self.name = name
        self.marks = [m1, m2, m3]

    def total(self):
        return sum(self.marks)

    def percentage(self):
        return self.total() / 3          # method using another method

    def result(self):
        return "Pass" if min(self.marks) >= 33 else "Fail"

    def report(self):
        print(f"{self.name}: total={self.total()}, "
              f"percent={self.percentage():.1f}%, {self.result()}")

s = Student("Neha", 88, 91, 79)
s.report()          # Neha: total=258, percent=86.0%, Pass
```

A List of Objects (real-world pattern)

```
students = [
    Student("Asha", 92, 85, 88),
    Student("Vikram", 45, 30, 60),
    Student("Neha", 88, 91, 79),
]

for s in students:
    s.report()

topper = max(students, key=lambda s: s.total())
print("Topper:", topper.name)
```

Output:

```
Asha: total=265, percent=88.3%, Pass
Vikram: total=135, percent=45.0%, Fail
Neha: total=258, percent=86.0%, Pass
Topper: Asha
```

Inheritance — First Taste

```
class Person:                                # parent class
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def introduce(self):
        print(f"I am {self.name}, {self.age} years old.")

class Teacher(Person):                       # child class – inherits Person
    def __init__(self, name, age, subject):
        super().__init__(name, age)         # reuse parent's constructor
        self.subject = subject
    def introduce(self):                     # override + extend
        super().introduce()
        print(f"I teach {self.subject}.")

class Student(Person):                       # another child
    pass                                    # inherits everything as-is

t = Teacher("Mrs. Rao", 38, "Physics")
s = Student("Karan", 15)
t.introduce()
s.introduce()
```

Output:

```
I am Mrs. Rao, 38 years old.
I teach Physics.
I am Karan, 15 years old.
```

`Teacher` and `Student` got `introduce()` free from `Person`; `Teacher` customized it. That is code reuse at the class level.

The str Method (nice printing)

```
class Book:
    def __init__(self, title, price):
        self.title = title
        self.price = price
    def __str__(self):
        return f'{self.title}' @ Rs.{self.price}"

b = Book("Python Basics", 299)
print(b)          # 'Python Basics' @ Rs.299 (instead of <__main__.Book object...>)
```

Practice Exercises

1. Add a class attribute `count` to `Student` that increases by 1 in `__init__`; print how many students exist.
2. Build `BankAccount` with `holder`, `balance`, methods `deposit(amt)`, `withdraw(amt)` (block overdraft), `show()`.
3. Create a list of 4 `BankAccount` objects; loop to print all balances and the richest account.
4. Parent `Vehicle(brand, wheels)` with `describe()`; children `Bike` and `Car` overriding `describe()` with extra info.
5. Give `BankAccount` a `__str__` so `print(account)` shows `"Asha: Rs.5000"`.

Key Points

- Class attributes are shared; instance attributes are per-object.
- Methods can call other methods via `self.`
- Lists of objects + loops = managing many records elegantly.
- Inheritance: `class Child(Parent)`, `super().__init__()` reuses the parent.
- `__str__` defines how `print(object)` looks.

Understanding

Second lab day — this time weaving together modules, files, exceptions, and classes. These problems mirror real tasks: safe input, saved data, and object-based design. Teachers: run this as a 2-hour lab if possible.

Level 1: Warm-Up

Problem 1 — Password generator (modules).

```
import random
import string

length = int(input("Password length: "))
characters = string.ascii_letters + string.digits + "!@#$%"
password = ""
for _ in range(length):
    password += random.choice(characters)
print("Your password:", password)
```

Problem 2 — Safe average (exceptions). Keep asking for numbers; ignore invalid entries with a message; stop at input `done`; print the average.

```
values = []
while True:
    entry = input("Number (or 'done'): ")
    if entry == "done":
        break
    try:
        values.append(float(entry))
    except ValueError:
        print("Skipped — not a number.")

if values:
    print(f"Average of {len(values)} numbers: {sum(values)/len(values):.2f}")
else:
    print("No valid numbers entered.")
```

Level 2: Core Practice

Problem 3 — Word frequency from a file. Create `story.txt` with a paragraph; read it; print each word's frequency using a dictionary (Day 18's pattern + Day 23's reading).

Problem 4 — Expense logger (files + datetime).

```
import datetime

expense = input("What did you buy? ")
amount = input("Amount: ")
stamp = datetime.date.today()

with open("expenses.txt", "a") as f:
    f.write(f"{stamp},{expense},{amount}\n")

print("--- All expenses ---")
total = 0.0
with open("expenses.txt") as f:
    for line in f:
        date, item, amt = line.strip().split(",")
        print(f"{date} {item:15} Rs.{amt}")
        total += float(amt)
print(f"Total spent: Rs.{total:.2f}")
```

Run it several times — the file keeps growing. Persistence!

Problem 5 — Temperature class.

```
class Temperature:
    def __init__(self, celsius):
        self.celsius = celsius
    def to_fahrenheit(self):
        return self.celsius * 9 / 5 + 32
    def status(self):
        if self.celsius >= 40: return "Very hot"
        if self.celsius >= 30: return "Hot"
        if self.celsius >= 20: return "Pleasant"
        return "Cold"

t = Temperature(float(input("Temp in C: ")))
print(f"{t.celsius}C = {t.to_fahrenheit():.1f}F -> {t.status()}")
```

Level 3: Challenge

Problem 6 — Library system (classes + list of objects). Class `Book(title, author, available=True)` with methods `issue()` and `return_book()` that flip availability with messages. Maintain a list of 5 books; menu: show all / issue / return, finding books by title.

Problem 7 — Robust marks file. Combine everything: a `Student` class; read `marks.txt` safely (handle `FileNotFoundError`); skip corrupt lines with try-except; build objects into a list; print a ranked report card and write it to `report.txt`.

Problem 8 — Rock-Paper-Scissors vs computer (`random.choice`), best of 5, with input validation and a scores dictionary.

Homework

1. Number-to-words: 543 → “five four three” (dictionary of digits).
2. Append a line to `diary.txt` with today’s date and a sentence; show the full diary after.
3. `Rectangle` class with `area()` and `is_square()`; make 3 objects from user input using safe-input loops.
4. Explain the difference between `IndexError`, `KeyError`, and `ValueError` with one example each.

Key Points

- Real programs = containers + functions + files + exceptions + classes working together.
- Append mode + loops = simple databases in plain text files.
- try-except around file reading and user input is professional hygiene.
- Classes with lists of objects can model libraries, banks, schools — anything.

Review of Week 4

You can now import power (`math` , `random` , `datetime` , your own modules), give programs permanent memory (files), keep them unbreakable (exceptions), and structure them like real software (classes and objects). This is the complete beginner-to-intermediate bridge.

Rapid Revision Quiz

1. Difference between `import math` and `from math import sqrt` in how you call `sqrt` .
2. Which file mode would you use for a daily diary program, and why?
3. What does `finally` guarantee?
4. In `class Dog:` , what exactly is `self` inside its methods?
5. What is inheritance in one sentence, and what does `super()` do?

Mini Project 4: Quiz Master — Persistent Quiz Application

Goal: A quiz game that reads questions, quizzes the user with bullet-proof input, uses a class for logic, and saves high scores to a file.

Complete Solution:

```

# quiz_master.py - Week 4 Mini Project
import random
import datetime

class Quiz:
    def __init__(self, questions):
        self.questions = questions
        self.score = 0

    def ask(self, q):
        """Ask one question dict; return True if correct."""
        print("\n" + q["question"])
        for i, opt in enumerate(q["options"], start=1):
            print(f" {i}. {opt}")
        while True:
            # bullet-proof input
            try:
                choice = int(input("Your answer (1-4): "))
                if 1 <= choice <= 4:
                    break
                print("Choose between 1 and 4.")
            except ValueError:
                print("Numbers only, please.")
        return q["options"][choice - 1] == q["answer"]

    def play(self):
        random.shuffle(self.questions)
        for q in self.questions:
            if self.ask(q):
                self.score += 1
                print("Correct!")
            else:
                print(f"Wrong. Answer: {q['answer']}")
        return self.score

    def save_score(name, score, total):
        stamp = datetime.date.today()
        with open("scores.txt", "a") as f:
            f.write(f"{stamp},{name},{score}/{total}\n")

    def show_history():
        try:
            with open("scores.txt") as f:
                print("\n--- Score History ---")
                for line in f:
                    date, name, result = line.strip().split(",")
                    print(f"{date} {name:12} {result}")
        except FileNotFoundError:
            print("No previous scores yet.")

# ----- question bank -----
QUESTIONS = [
    {"question": "Who created Python?",
     "options": ["Guido van Rossum", "James Gosling", "Dennis Ritchie", "Elon Musk"],
     "answer": "Guido van Rossum"},
    {"question": "Which keyword defines a function?",

```

```

    "options": ["func", "def", "define", "lambda"],
    "answer": "def"},
{"question": "What does len('Python') return?",
 "options": ["5", "6", "7", "Error"],
 "answer": "6"},
{"question": "Which collection does NOT allow duplicates?",
 "options": ["list", "tuple", "set", "string"],
 "answer": "set"},
{"question": "What is the output of 10 // 3?",
 "options": ["3.33", "3", "4", "1"],
 "answer": "3"},
]

# ----- main -----
print("==== QUIZ MASTER =====")
name = input("Your name: ")
quiz = Quiz(QUESTIONS)
score = quiz.play()

total = len(QUESTIONS)
print(f"\n{name}, you scored {score}/{total} ({score/total*100:.0f}%)")
save_score(name, score, total)
show_history()

```

Sample run (shortened):

```

==== QUIZ MASTER =====
Your name: Asha

Which keyword defines a function?
  1. func
  2. def
  3. define
  4. lambda
Your answer (1-4): 2
Correct!
...
Asha, you scored 4/5 (80%)

--- Score History ---
2026-07-19  Asha      4/5

```

Run it again — the history grows. A program with memory, structure, and resilience.

Extension Challenges

1. Load questions from a `questions.txt` file (one line: `question|opt1|opt2|opt3|opt4|answer`).
2. Add difficulty levels stored in the question dicts; let the user choose.
3. Show a leaderboard: parse `scores.txt` and print the top 3 scores.
4. Add a timer per question with `time` (advanced).

Teacher's Assessment Rubric

Criterion	Marks
Quiz class works (shuffle, ask, score)	3
Input cannot crash the program	3
Scores saved to and read from file	3
Code organized into functions/methods	1
Total	10

Key Points

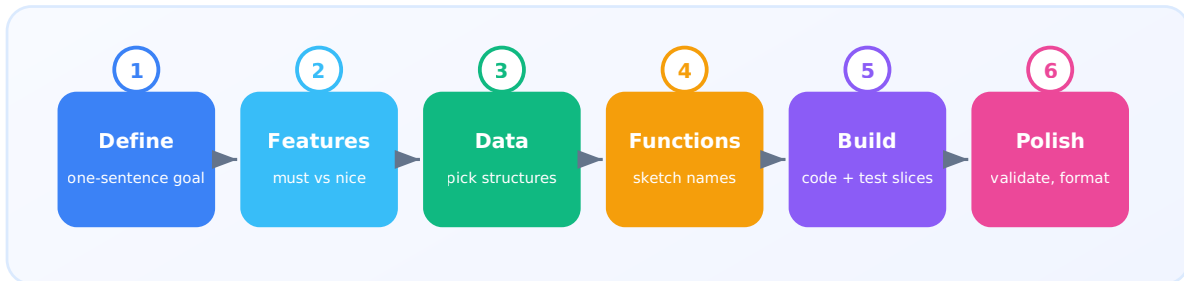
- A dictionary-per-question + a Quiz class is a clean, extensible design.
- Persistence (files) turns a toy into an application.
- try-except at every input and file boundary = crash-proof software.
- You are now ready for the Final Push: a complete project of your own!

Days 29-30

FINAL PUSH: Days 29-30

Consolidate the entire course by planning and building a complete application from scratch — the graduation project.

Build a Small Project



The 6-step professional project workflow — plan in writing before you type

Understanding

Today students stop following recipes and start cooking on their own. The most important skill in programming is not memorizing syntax — it is **decomposition**: breaking a big problem into small, solvable steps. Today's chapter teaches the professional project workflow, then walks through a full project as a model.

The 6-Step Project Workflow

1. **Define** — one sentence: *What does the program do?*
2. **List features** — number them; mark “must have” vs “nice to have”.
3. **Choose data structures** — what stores the data? (list of dicts? class? file?)
4. **Sketch functions** — name each function and what it takes/returns *before* coding.
5. **Build & test in slices** — code one feature, test it, then the next. Never write 100 lines untested.
6. **Polish** — validation, messages, formatting.

Model Project: To-Do List Manager (built step by step)

Step 1 — Definition: “A menu program to add, view, complete, and delete daily tasks, saved to a file.”

Step 2 — Features: 1) Add task 2) View tasks with status 3) Mark done 4) Delete 5) Auto-save/load 6) Exit.

Step 3 — Data structure: a list of dictionaries: `{"task": str, "done": bool}` — file line format: `task|done` .

Step 4 — Function sketch: `load_tasks()` , `save_tasks(tasks)` , `add_task(tasks, text)` , `show_tasks(tasks)` , `mark_done(tasks, i)` , `delete_task(tasks, i)` .

Steps 5-6 — Complete Program:

```

# todo.py - Day 29 model project
FILENAME = "tasks.txt"

def load_tasks():
    """Read tasks from file into a list of dicts."""
    tasks = []
    try:
        with open(FILENAME) as f:
            for line in f:
                text, done = line.strip().split("|")
                tasks.append({"task": text, "done": done == "True"})
    except FileNotFoundError:
        pass # first run: no file yet
    return tasks

def save_tasks(tasks):
    with open(FILENAME, "w") as f:
        for t in tasks:
            f.write(f"{t['task']}|{t['done']}\n")

def show_tasks(tasks):
    if not tasks:
        print("No tasks yet. Enjoy your day!")
        return
    print("\n# Status Task")
    for i, t in enumerate(tasks, start=1):
        status = "[x]" if t["done"] else "[ ]"
        print(f"{i}. {status} {t['task']}")

def valid_index(tasks, prompt):
    """Bullet-proof: return a valid task number or None."""
    if not tasks:
        print("List is empty.")
        return None
    try:
        i = int(input(prompt))
        if 1 <= i <= len(tasks):
            return i - 1
        print("No such task number.")
    except ValueError:
        print("Numbers only.")
    return None

# ----- main -----
tasks = load_tasks()
print("===== MY TO-DO LIST =====")

while True:
    print("\n1.Add 2.View 3.Mark Done 4.Delete 5.Exit")
    choice = input("Choose: ")

    if choice == "1":
        text = input("New task: ").strip()
        if text:
            tasks.append({"task": text, "done": False})

```

```

        save_tasks(tasks)
        print("Added.")
    else:
        print("Task cannot be empty.")
    elif choice == "2":
        show_tasks(tasks)
    elif choice == "3":
        i = valid_index(tasks, "Task number to mark done: ")
        if i is not None:
            tasks[i]["done"] = True
            save_tasks(tasks)
            print("Marked done.")
    elif choice == "4":
        i = valid_index(tasks, "Task number to delete: ")
        if i is not None:
            removed = tasks.pop(i)
            save_tasks(tasks)
            print(f"Deleted: {removed['task']}")
    elif choice == "5":
        pending = sum(1 for t in tasks if not t["done"])
        print(f"Goodbye! {pending} task(s) still pending.")
        break
    else:
        print("Choose 1-5.")

```

Sample run (shortened):

```

===== MY TO-DO LIST =====

1.Add 2.View 3.Mark Done 4.Delete 5.Exit
Choose: 1
New task: Revise Python Week 3
Added.

Choose: 2

# Status Task
1. [ ] Revise Python Week 3

```

Close and reopen the program — the tasks are still there. **Trace which day taught each part:** input loop (D10-11), dict-in-list (D15/18), functions with returns (D19-20), file persistence (D23), exception safety (D24).

Student Project Menu (choose one to build today)

- **Expense Tracker** — add expenses with category; monthly total; save to file.
- **Student Result System** — class `Student`, list of objects, ranked report to file.
- **Word Game (Hangman-style)** — random word, guess letters, limited lives.
- **Shop Billing System** — product dictionary, cart list, GST, printed bill saved to file.
- **Flashcard Revision App** — question/answer pairs from a file, shuffled quiz, score tracking.

Rule: whatever you choose, follow the 6 steps *in writing* before typing any code.

Key Points

- Decomposition (the 6 steps) matters more than typing speed.
- Build in slices; test each slice before the next.
- Every project = data structure + functions + loop + persistence + validation.

Understanding

Thirty days ago you didn't know what a variable was. Today you will complete a capstone project and review the entire journey. This chapter contains the full-course revision map, the capstone specification, and your roadmap for what comes after Day 30.

The Whole Course on One Page

Week	Days	You mastered
1 — Fundamentals	1-7	Python setup, <code>print</code> , variables, data types, operators, <code>input</code> , f-strings, type casting
2 — Control Flow	8-14	<code>if/elif/else</code> , <code>for</code> , <code>while</code> , <code>range</code> , <code>break/continue/pass</code> , nested loops
3 — Data & Functions	15-21	Lists, tuples, sets, dictionaries, <code>def</code> , parameters, <code>return</code> , scope
4 — Real Software	22-28	Modules, file handling, exceptions, classes, objects, inheritance
Final Push	29-30	Project workflow, capstone build

Grand Revision Quiz (30 questions — one per day!)

1. Who created Python and when?
2. What does `print("A", "B", sep="/")` output?
3. Name the type: `7.0`, `"7"`, `True`, `None`.
4. Evaluate: `17 % 5`, `17 // 5`, `2 ** 3`.
5. Why does `input()` need casting before arithmetic?
6. `int(9.9)` vs `round(9.9)`?
7. Write the f-string for `Total: 250.50` given `t = 250.5`.
8. When is an `if` block executed?
9. In an `elif` chain, how many branches can run?
10. `range(1, 10, 2)` produces...?

11. `break` vs `continue` ?
12. Total inner-body executions for `outer×inner = 3×4` loops?
13. What is FizzBuzz's first rule-ordering trap?
14. What pattern is `while True: + break` ?
15. Index of the last item of list `L` (two ways)?
16. Why does `(5)` not create a tuple?
17. What does a set do to duplicates?
18. `d.get("x", 0)` vs `d["x"]` when "x" is missing?
19. Parameter vs argument?
20. What does a function without `return` give back?
21. Which container for key→value mapping?
22. Three ways to import `sqrt` from `math` .
23. File mode that erases existing content?
24. Why is `with open(...)` preferred?
25. Which exception: `int("abc")` ? `10/0` ? `lst[99]` ?
26. What does `__init__` do, and when does it run?
27. What is `self` ?
28. Class attribute vs instance attribute?
29. Name the 6 project steps in order.
30. What does `super().__init__()` do in a child class?

Capstone Project: Library Management System

Specification. Build a complete Library Management System that demonstrates every week of the course:

1. **Class Book** — attributes: `book_id` , `title` , `author` , `available` ; a `__str__` for neat printing. (Week 4)
2. **Data** — a dictionary `{book_id: Book}` for fast lookup; a set of issued IDs; issue history as a list. (Week 3)
3. **Menu loop** — Add Book / Show All / Search by title / Issue / Return / Statistics / Exit. (Week 2)
4. **Persistence** — books saved to `library.txt` (format `id|title|author|available`), loaded at start. (Week 4)
5. **Safety** — every input validated with try-except; missing file handled gracefully. (Week 4)
6. **Statistics** — total books, issued count, availability percentage with formatted output. (Week 1)

Skeleton to start from:

```

# library.py - Capstone
FILENAME = "library.txt"

class Book:
    def __init__(self, book_id, title, author, available=True):
        self.book_id = book_id
        self.title = title
        self.author = author
        self.available = available

    def __str__(self):
        status = "Available" if self.available else "Issued"
        return f"[{self.book_id}] {self.title} by {self.author} - {status}"

def load_books():
    books = {}
    try:
        with open(FILENAME) as f:
            for line in f:
                bid, title, author, avail = line.strip().split("|")
                books[bid] = Book(bid, title, author, avail == "True")
    except FileNotFoundError:
        pass
    return books

def save_books(books):
    with open(FILENAME, "w") as f:
        for b in books.values():
            f.write(f"{b.book_id}|{b.title}|{b.author}|{b.available}\n")

def add_book(books):
    bid = input("Book ID : ").strip()
    if bid in books:
        print("ID already exists.")
        return
    title = input("Title : ").strip()
    author = input("Author : ").strip()
    books[bid] = Book(bid, title, author)
    save_books(books)
    print("Book added.")

def issue_book(books):
    bid = input("Book ID to issue: ").strip()
    book = books.get(bid)
    if book is None:
        print("No such book.")
    elif not book.available:
        print("Already issued.")
    else:
        book.available = False
        save_books(books)
        print(f"Issued: {book.title}")

def statistics(books):
    total = len(books)

```

```

    if total == 0:
        print("Library is empty.")
        return
    issued = sum(1 for b in books.values() if not b.available)
    print(f"Total books : {total}")
    print(f"Issued      : {issued}")
    print(f"Available   : {total - issued} ({(total - issued) / total * 100:.0f}%)")

# --- main menu: complete the remaining options yourself! ---
books = load_books()
while True:
    print("\n1.Add 2.Show All 3.Search 4.Issue 5.Return 6.Stats 7.Exit")
    choice = input("Choose: ")
    if choice == "1":
        add_book(books)
    elif choice == "2":
        for b in books.values():
            print(b)
    elif choice == "4":
        issue_book(books)
    elif choice == "6":
        statistics(books)
    elif choice == "7":
        print("Thank you for using PyLibrary!")
        break
    else:
        print("Complete options 3 and 5 as your final task!")

```

Your final task: implement Search (option 3: loop titles with `in` for partial match) and Return (option 5: reverse of issue). Then demonstrate the full system to your class.

Teacher's Capstone Rubric (out of 25)

Criterion	Marks
Book class correctly designed with <code>_str_</code>	4
All 7 menu options functional	7
Data persists across runs	4
No input can crash the program	4
Code style: names, functions, comments	3
Viva: student explains any 3 parts	3

After Day 30: Your Roadmap

- **Practice platforms:** HackerRank, LeetCode (easy), Codewars — one problem daily keeps skills alive.

- **Next topics:** list comprehensions, lambda/map/filter, JSON & CSV modules, virtual environments, pip.
- **Choose a direction:**
 - *Web development* → Flask, then Django
 - *Data science* → NumPy, Pandas, Matplotlib
 - *Automation* → `os` , `shutil` , `openpyxl` , web scraping with Requests/BeautifulSoup
 - *AI/ML* → after data science: scikit-learn
- **Build in public:** put your projects on GitHub; each project teaches more than ten tutorials.

Congratulations!

You have completed the 30-Day Python Learning Plan. You began with `print("Hello, World!")` and finished with a persistent, object-oriented, crash-proof application. Programming is a craft — you now hold the tools.

Keep coding, keep growing!

Appendix A: Quick Reference Card

Data types: `int float str bool None list tuple set dict`

Casting: `int(x) float(x) str(x) bool(x) list(x) tuple(x) set(x)`

String essentials: `len(s) s.upper() s.lower() s.strip() s.split(",") s.replace(a,b) s.isdigit()
"x" in s s[0] s[-1] s[1:4]`

List essentials: `L.append(x) L.insert(i,x) L.remove(x) L.pop() L.sort() L.reverse() L.count(x)
L.index(x) len/sum/max/min(L)`

Dict essentials: `d[k] d.get(k, default) d[k]=v d.pop(k) d.keys() d.values() d.items() k in d`

Set essentials: `s.add(x) s.discard(x) a|b a&b a-b a^b`

Control flow skeletons:

```
if cond: ... elif cond: ... else: ...  
for i in range(start, stop, step): ...  
while cond: ...  
break / continue / pass
```

Function skeleton:

```
def name(param, opt=default):  
    """docstring"""  
    return value
```

File skeleton:

```
with open("f.txt", "r") as f: # r / w / a / x  
    for line in f:  
        line.strip()
```

Exception skeleton:

```
try: ...  
except ValueError: ...  
except Exception as e: ...  
else: ...  
finally: ...
```

Class skeleton:

```
class Name(Parent):  
    def __init__(self, x):  
        self.x = x  
    def method(self):  
        return self.x
```

Appendix B: Common Errors & Fixes

Error	Typical cause	Fix
<code>SyntaxError</code>	Missing <code>:</code> or bracket	Check line end and pairs
<code>IndentationError</code>	Wrong/mixed spaces	Use 4 spaces consistently
<code>NameError</code>	Typo / using before defining	Check spelling & order
<code>TypeError</code>	Mixing types (<code>"2"+2</code>)	Cast first
<code>ValueError</code>	Bad conversion (<code>int("abc")</code>)	Validate with try-except
<code>IndexError</code>	Index beyond list	Check <code>len()</code>
<code>KeyError</code>	Missing dict key	Use <code>.get()</code>
<code>ZeroDivisionError</code>	Dividing by 0	Check denominator first
<code>FileNotFoundError</code>	Missing file in <code>"r"</code> mode	Check <code>os.path.exists</code> / try-except
<code>AttributeError</code>	Wrong method name	Check spelling / type

Appendix C: 30-Day Plan at a Glance

Day	Topic	Day	Topic
1	Introduction to Python	16	Tuples
2	Setup & First Program	17	Sets
3	Variables & Data Types	18	Dictionaries
4	Basic Operators	19	Functions Basics
5	Input & Output	20	Return Statement
6	Type Casting	21	Mini Project & Review
7	Mini Project & Review	22	Modules & Import
8	If Statements	23	File Handling (Basics)
9	If-Else Statements	24	Errors & Exceptions
10	Loops (For, While)	25	OOP Introduction
11	Break, Continue & Pass	26	Classes & Objects
12	Nested Loops	27	Practice Problems
13	Practice Problems	28	Mini Project & Review
14	Mini Project & Review	29	Build a Small Project
15	Lists	30	Final Project & Review

From Hello, World! to Real Projects — in 30 Days

A complete, classroom-ready Python course. One chapter a day: a concept explained in plain language, an exam-ready definition, working examples with output, and practice exercises — finishing every week with a real mini project.

- ✓ 30 structured daily chapters
- ✓ 150+ practice exercises & lab days
- ✓ 5 complete projects: Report Card, ATM, Contact Book, Quiz App, Library System
- ✓ Charts, flowcharts & revision cards in every week
- ✓ Teacher rubrics for classroom assessment

Gaurav Malviya

Educator & Software Consultant



CONSISTENCY > MOTIVATION