In Alignment with NEP 2020

Advance PYTHON PROGRAMMING

BCA | Semester IV | Academic Session 2024 – 2027

Compiled by

Deepak Kumar Tiwari

Asst. Professor, Dept. of BCA RVSCET,
Jamshedpur

Detailed Week-wise Study Notes with Extra Examples & Real-World Scenarios

Course Objectives

1. Strengthen Python programming concepts beyond basics.
2. Introduce advanced data handling and modular programming.
3. Develop skills in file handling, exception handling and OOP using Python.
4. Familiarize students with libraries for data processing and visualization.
5. Prepare students for real-world Python-based application development.

Course Outcomes

- CO1: Apply advanced Python syntax and built-in features effectively.
- CO2: Develop programs using functions, modules and packages.
- CO3: Implement object-oriented programming concepts in Python.
- CO4: Handle files, exceptions and external data efficiently.
- CO5: Use Python libraries for data processing and basic applications.

Table of Contents

Week 1: Review of Python Basics

Course Outcome	Program Outcomes	Topics Covered
CO1	PO1, PO2, PO5, PO12	Python programming structure, Variables and data types, Control statements, Writing efficient Python code

1.1 Python Programming Structure

Definition

Python is a high-level, interpreted, general-purpose programming language. It uses indentation (whitespace) instead of braces to define blocks of code. Every Python program is a sequence of statements executed by the interpreter line-by-line from top to bottom.

Real-World Analogy

Think of a Python program like a recipe. Each line is one cooking step, and indentation is like sub-steps written underneath the main step. The interpreter is the cook who reads each step in order and performs it.

Basic Structure

```
# 1. Comments         -> notes for humans
# 2. Imports           -> bring in extra tools
import math

# 3. Variables / functions
PI = 3.14159

def area_of_circle(radius):
    return PI * radius * radius

# 4. Main logic
r = 5
print("Area =", area_of_circle(r))   # Area = 78.53975
```

Example 1: Hello World with User Input

```
name = input("Enter your name: ")
print(f"Hello, {name}! Welcome to Python.")
```

Example 2: Simple Calculator

```
a = int(input("Enter first number: "))
b = int(input("Enter second number: "))
print("Sum =", a + b)
print("Product =", a * b)
```

Example 3: Indentation Importance

❌ Bad Code (Avoid)	✅ Good Code (Preferred)
`if 5 > 2:` `print("Yes")  # ❌ IndentationError`	`if 5 > 2:` `    print("Yes")  # ✅ 4-space indent`

1.2 Variables and Data Types

Definition

A variable is a named memory location that stores a value. Python is dynamically typed — you don't declare the type; Python figures it out automatically.

Real-World Scenario: Variables in Banking

A banking app uses variables for each customer:

account_number = 12345678 (int)

customer_name = "Asha" (str)

balance = 25750.50 (float)

is_active = True (bool)

These four variables together describe one account record.

Common Built-in Data Types

Type	Example	Use For
int	`10, -5, 1000`	Whole numbers
float	`3.14, -0.5`	Decimal numbers
str	`"hello", 'Python'`	Text
bool	`True, False`	Yes/No values
list	`[1, 2, 3]`	Ordered, changeable collection
tuple	`(1, 2, 3)`	Ordered, fixed collection
dict	`{"name": "Deepak"}`	Key-value pairs
set	`{1, 2, 3}`	Unique items only

Example 1: Type Checking

```
marks = 87.5
name = "Asha"
print(type(marks))   # <class 'float'>
print(type(name))    # <class 'str'>
```

Example 2: Type Conversion (Casting)

```
x = "100"            # str
y = int(x)           # convert to int
z = float(x)         # convert to float
print(y + 50)        # 150
print(z + 0.5)       # 100.5
```

Example 3: Multiple Assignment

```
a, b, c = 5, 10, 15
print(a, b, c)       # 5 10 15

x = y = z = 0        # all three get 0
print(x, y, z)       # 0 0 0
```

Tip

Use type(variable) to check the data type; use isinstance(x, int) to check if x is of a given type.

1.3 Control Statements

Definition

Control statements decide the flow of a program — which lines run, in what order, and how many times. They include conditional (if/elif/else), looping (for, while), and jump statements (break, continue, pass).

(a) if / elif / else

Example 1: Grading System

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

Example 2: Even or Odd

```
n = int(input("Enter a number: "))
if n % 2 == 0:
    print(n, "is Even")
else:
    print(n, "is Odd")
```

(b) Loops: for and while

Example 1: Multiplication Table

```
n = 5
for i in range(1, 11):
    print(f"{n} x {i} = {n*i}")
```

Example 2: Sum of First N Numbers (while)

```
n = 10
total = 0
i = 1
while i <= n:
    total += i
    i += 1
print("Sum =", total)   # Sum = 55
```

(c) Jump Statements: break, continue, pass

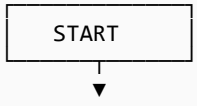
Example 1: Stop at First Negative

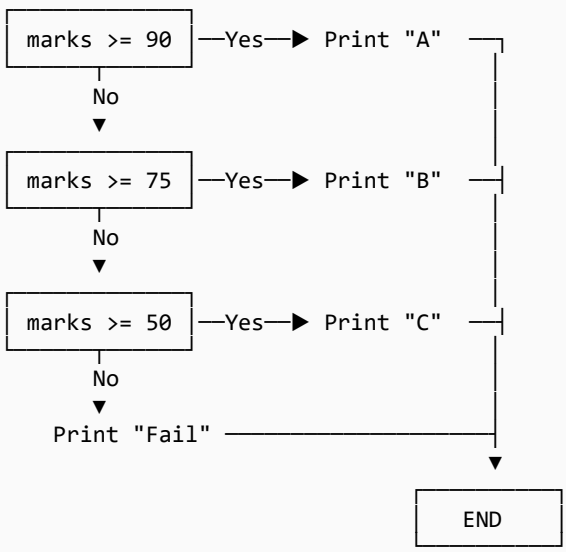
```
for n in [3, 7, 2, -1, 5]:
    if n < 0:
        break
    print(n, end=" ")
# Output: 3 7 2
```

Example 2: Skip Multiples of 3

```
for i in range(1, 11):
    if i % 3 == 0:
        continue
    print(i, end=" ")
# Output: 1 2 4 5 7 8 10
```

Flow Diagram: if-elif-else





1.4 Writing Efficient Python Code

Efficient Python code is readable, short, fast, and "Pythonic" (using Python's natural style).

Key Habits

1. Use meaningful variable names: total_marks instead of t.
2. Avoid unnecessary loops — use built-ins like sum(), max(), len().
3. Use list comprehensions for short transformations.
4. Follow PEP 8 style: 4 spaces indent, snake_case for variables.
5. Comment why, not what.

❌ Bad Code (Avoid)	✅ Good Code (Preferred)

`# Sum 1..10 the long way`
`total = 0`
`for i in range(1, 11):`
`  total = total + i`
`print(total)`
	`# Pythonic one-liner`
	`total = sum(range(1, 11))`
	`print(total)`

Example 1: List Comprehension — Squares

```
squares = [x*x for x in range(1, 6)]
print(squares)  # [1, 4, 9, 16, 25]
```

Example 2: Filter Even Numbers

```
evens = [x for x in range(1, 21) if x % 2 == 0]
print(evens)
```

Week 1 Cheat Sheet

- Python = interpreted + indented + dynamically typed.
- Variables: no type declaration; use type() / isinstance().
- Data types: int, float, str, bool, list, tuple, dict, set.
- Decisions: if/elif/else; Loops: for, while; Jumps: break, continue, pass.
- Write clean, Pythonic code — follow PEP 8.

Week 1 Review Questions

6. Define Python and list any three of its key features.
7. What is dynamic typing? Give an example.
8. Differentiate between list, tuple, and set with examples.
9. Write a Python program using if-elif-else to print grade based on marks.
10. Convert this loop into a list comprehension: a list of cubes from 1 to 10.
11. State any three rules of PEP 8 style.

Week 2: Advanced Data Types

Course Outcome	Program Outcomes	Topics Covered
CO1	PO1, PO2, PO5, PO12	Lists and advanced list operations, Tuples and sets, Dictionary operations, Comprehensions

2.1 Lists and Advanced List Operations

Definition

A list is an ordered, changeable (mutable) collection of items written inside square brackets [ ]. Items can be of any data type and can be repeated.

Real-World Scenario: Shopping Cart in E-commerce

An online shopping cart is naturally a list:

cart = ["Shoes", "T-Shirt", "Watch", "Shoes"]

You can ADD items (append), REMOVE items, count duplicates,

see total items (len), and check order — exactly what a list does.

Syntax

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

```
numbers = [10, 20, 30, 40]
mixed   = [1, "hello", 3.14, True]
```

Example 1: List Methods

```
fruits = ["apple", "banana", "mango"]
fruits.append("orange")        # add at end
fruits.insert(1, "guava")      # insert at index 1
fruits.remove("banana")        # remove by value
last = fruits.pop()            # remove & return last
fruits.sort()                  # sort alphabetically
print(fruits)
```

Example 2: Slicing

```
nums = [10, 20, 30, 40, 50]
print(nums[1:4])     # [20, 30, 40]
print(nums[:3])      # [10, 20, 30]
print(nums[::-1])    # [50, 40, 30, 20, 10]  reverse
```

Example 3: Nested Lists (2-D Matrix)

```
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9],
]
print(matrix[1][2])    # 6
for row in matrix:
    print(row)
```

Example 4: List Comprehension

```
evens = [x for x in range(1, 21) if x % 2 == 0]
print(evens)
# [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

Memory Diagram of a List

```
Index :   0        1        2        3

fruits: | apple  | banana | mango  | orange |

Negative
index :  -4       -3       -2       -1
```

2.2 Tuples and Sets

Tuple

Definition

A tuple is an ordered but immutable collection in parentheses ( ). Once created, you cannot change it. Tuples are faster than lists and ideal for fixed data.

> **Real-World Scenario: GPS Coordinates**
>
> A GPS location never changes its structure: (latitude, longitude).
>
> jamshedpur = (22.8046, 86.2029)
>
> Storing it as a tuple guarantees no one can accidentally modify
>
> the latitude or longitude — perfect for fixed reference data.

Example 1: Basic Tuple

```
point = (3, 4)
days  = ("Mon", "Tue", "Wed", "Thu", "Fri")
print(point[0])      # 3
print(len(days))     # 5
# point[0] = 10      # ❌ TypeError: tuples are immutable
```

Example 2: Tuple Packing and Unpacking

```
student = ("Deepak", 25, "BCA")     # packing
name, age, course = student         # unpacking
print(name, age, course)            # Deepak 25 BCA
```

Example 3: Returning Multiple Values

```
def min_max(nums):
    return min(nums), max(nums)     # returns a tuple

low, high = min_max([5, 1, 9, 3])
print(low, high)                    # 1 9
```

Set

Definition

A set is an unordered collection of unique items in curly braces { }. Duplicates are removed automatically.

> **Real-World Scenario: Unique Email IDs in a Newsletter**
>
> When 10,000 people sign up, some may register twice.
>
> emails = set(submitted_emails)

instantly removes duplicates so each person gets only ONE email.

Example 1: Removing Duplicates

```
marks = [80, 90, 75, 80, 90, 75]
unique_marks = set(marks)
print(unique_marks)   # {80, 90, 75}
```

Example 2: Set Operations

```
a = {1, 2, 3, 4}
b = {3, 4, 5, 6}
print(a | b)   # union          {1,2,3,4,5,6}
print(a & b)   # intersection   {3, 4}
print(a - b)   # difference     {1, 2}
print(a ^ b)   # symm diff      {1,2,5,6}
```

Example 3: Set Membership Test

```
enrolled = {"Deepak", "Asha", "Ravi"}
print("Asha" in enrolled)    # True
print("Sita" in enrolled)    # False
```

List vs Tuple vs Set

Feature	List	Tuple	Set
Brackets	[ ]	( )	{ }
Order	Ordered	Ordered	Unordered
Changeable	Yes	No	Yes (add/remove)
Duplicates	Allowed	Allowed	Not allowed
Indexing	Yes	Yes	No

2.3 Dictionary Operations

Definition

A dictionary stores data as key–value pairs in { }. Each key is unique and is used to look up its value. Python uses a hash table behind the scenes, making dictionary lookups extremely fast.

Real-World Scenario: Phone Contact List

Your phone contacts work exactly like a dictionary:

contacts = {"Mom": "9876543210", "Asha": "9123456789"}

To call Mom, you don't scroll through every contact — you

look up her name and instantly get the number. That's O(1) lookup.

Example 1: Basic Dictionary

```
student = {"name": "Deepak", "age": 25, "course": "BCA"}
print(student["name"])              # Deepak
print(student.get("age"))           # 25
print(student.get("grade", "N/A"))  # N/A  (default if missing)
```

Example 2: Add, Update, Delete

```
student = {"name": "Deepak", "age": 25}
student["college"] = "RVSCET"   # add
student["age"] = 26             # update
del student["name"]             # delete
print(student)                  # {'age': 26, 'college': 'RVSCET'}
```

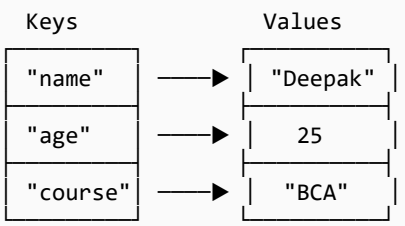
Example 3: Iterating Through a Dictionary

```
marks = {"Maths": 92, "Sci": 85, "Eng": 78}

for subject in marks:
    print(subject)                  # keys only

for subject, score in marks.items():
    print(subject, "->", score)
```

Example 4: Nested Dictionary (Students Database)

```
students = {
    "S101": {"name": "Deepak", "marks": 88},
    "S102": {"name": "Asha",   "marks": 92},
}
print(students["S102"]["name"])    # Asha
```

Dictionary Diagram



2.4 Comprehensions

Definition

A comprehension is a compact, one-line way to build a list, set, or dictionary from another sequence. It replaces a normal for-loop with cleaner, more Pythonic code.

Example 1: List Comprehension

```
squares = [x*x for x in range(1, 6)]
print(squares)              # [1, 4, 9, 16, 25]
```

Example 2: With Condition

```
positives = [x for x in [-3, -1, 0, 2, 5] if x > 0]
print(positives)            # [2, 5]
```

Example 3: Set Comprehension

```
words = ["hi", "hello", "ok", "yes"]
unique_lengths = {len(w) for w in words}
print(unique_lengths)       # {2, 5}
```

Example 4: Dictionary Comprehension

```
squares = {x: x*x for x in range(1, 6)}
print(squares)
# {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

❌ Bad Code (Avoid)	✅ Good Code (Preferred)

`# Long way`
`squares = []`
`for x in range(1, 6):`
`  squares.append(x*x)`
	`# Pythonic`
	`squares = [x*x for x in range(1, 6)]`

Week 2 Cheat Sheet

- List = ordered, changeable, allows duplicates — [ ].
- Tuple = ordered, fixed — ( ).
- Set = unordered, unique items only — { }.
- Dictionary = key:value pairs, fast lookup — { key: value }.
- Comprehension = one-line loop to build list / set / dict.

Week 2 Review Questions

12. Why is a tuple faster than a list? Give one practical use of tuples.
13. Write a Python program that removes duplicate elements from a list using a set.

14. Create a dictionary of 5 students with their marks and print only those who scored above 60.
15. Convert the following into a list comprehension: cubes of even numbers from 1 to 10.
16. State three differences between list and set.

Week 3: Functions in Python

Course Outcome	Program Outcomes	Topics Covered
CO2	PO1, PO2, PO3, PO5, CO10, PO12	Defining and calling functions, Arguments and return values, Default and keyword arguments, Lambda functions

3.1 Defining and Calling Functions

Definition

A function is a named block of reusable code that performs a specific task. Functions help avoid repetition, make code modular, and easier to test. In Python, functions are defined using the def keyword.

> **Real-World Scenario: ATM Machine**
>
> An ATM has fixed operations: withdraw(amount), deposit(amount), check_balance(). You don't care HOW each works internally — you just call it with the right inputs and get a result. That's a function.

Syntax

```
def function_name(parameters):
    """Optional docstring."""
    # body
    return value     # optional
```

Example 1: Greet Function

```
def greet(name):
    print("Hello,", name, "!")

greet("Deepak")     # Hello, Deepak !
greet("Asha")       # Hello, Asha !
```

Example 2: Square of a Number

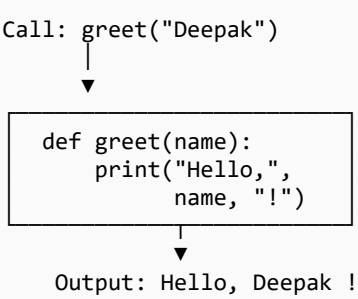
```
def square(n):
```

```
    return n * n

print(square(7))    # 49
print(square(10))   # 100
```

Example 3: Check Prime Number

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

print(is_prime(11))    # True
print(is_prime(15))    # False
```

Function Flow Diagram



3.2 Arguments and Return Values

Definition

Arguments (parameters) are the inputs you pass to a function. The return statement sends a value back from the function. A function without return returns None by default.

Example 1: Add Two Numbers

```
def add(a, b):
    return a + b

print(add(5, 3))         # 8
print(add(10.5, 4.5))    # 15.0
```

Example 2: Multiple Return Values (Statistics)

```
def stats(numbers):
```

```
    return min(numbers), max(numbers), sum(numbers)

low, high, total = stats([5, 9, 2, 7])
print(low, high, total)    # 2 9 23
```

Example 3: Variable-length Arguments (*args, **kwargs)

```
def total(*nums):
    return sum(nums)

print(total(1, 2, 3))         # 6
print(total(10, 20, 30, 40))  # 100

def info(**details):
    for k, v in details.items():
        print(k, ":", v)

info(name="Deepak", age=25, course="BCA")
```

❌ Bad Code (Avoid)	✅ Good Code (Preferred)

`# Repeating logic`
`print(5 + 3)`
`print(10 + 20)`
`print(7 + 8)`
	`# Reusable function`
	`def add(a, b):`
	`  return a + b`
	`print(add(5, 3))`
	`print(add(10, 20))`

3.3 Default and Keyword Arguments

Default Arguments

A default argument has a value in the function definition. If the caller doesn't pass that argument, Python uses the default.

Example 1: Default Greeting

```
def greet(name, msg="Welcome"):
    print(msg + ",", name)

greet("Deepak")               # Welcome, Deepak
greet("Asha", "Good Morning") # Good Morning, Asha
```

Example 2: Default Tax Calculator

```
def price_with_tax(price, tax_rate=0.18):
    return price + (price * tax_rate)

print(price_with_tax(1000))         # 1180.0   (18% default)
```

```
print(price_with_tax(1000, 0.05))   # 1050.0   (5% override)
```

Keyword Arguments

Keyword arguments let you pass values using parameter names — order doesn't matter.

Example 1: Student Info

```
def student(name, course, year):
    print(name, "-", course, "-", year)

student("Deepak", "BCA", 4)                        # positional
student(year=4, name="Deepak", course="BCA")       # keyword
```

> **Tip**
> Default arguments must come AFTER non-default ones. def f(a, b=2, c) is invalid; def f(a, b, c=2) is valid.

3.4 Lambda Functions

Definition

A lambda function is a small, anonymous function written in one line. Used for short tasks where defining a full function is unnecessary — often passed to map(), filter(), or sorted().

Syntax

```
lambda arguments : expression
```

Example 1: Basic Lambdas

```
square = lambda x: x * x
print(square(5))            # 25

add = lambda a, b: a + b
print(add(3, 7))            # 10
```

Example 2: Used with map()

```
nums = [1, 2, 3, 4]
squares = list(map(lambda x: x*x, nums))
print(squares)              # [1, 4, 9, 16]
```

Example 3: Used with filter()

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

```
mul5 = list(filter(lambda x: x % 5 == 0, nums))
print(mul5)                 # [10, 15, 20, 25, 30]
```

Example 4: Sorting with key=

```
students = [("Deepak", 88), ("Asha", 92), ("Ravi", 75)]
students.sort(key=lambda s: s[1], reverse=True)
print(students)
# [('Asha', 92), ('Deepak', 88), ('Ravi', 75)]
```

❌ Bad Code (Avoid)	✅ Good Code (Preferred)

`# Full function for tiny task`
`def double(x):`
`  return x * 2`
`result = list(map(double, [1,2,3]))`
	`# Lambda – concise`
	`result = list(map(lambda x: x*2, [1,2,3]))`

Week 3 Cheat Sheet

- def name(params): defines a function; call with name(args).
- return sends a value back; no return → returns None.
- Default args: def f(x, y=10) — y is optional.
- Keyword args: f(y=5, x=2) — order-independent.
- lambda x: expr is a one-line anonymous function.
- *args = variable positional; **kwargs = variable keyword.

Week 3 Review Questions

17. Define a function. List any three advantages of using functions.
18. Write a function factorial(n) that returns the factorial of n.
19. Differentiate between positional, default and keyword arguments with examples.
20. Write a lambda function that returns the cube of a number, then use it with map() on [1,2,3,4].
21. Can a Python function return more than one value? Justify with an example.

Week 4: Modules and Packages

Course Outcome	Program Outcomes	Topics Covered
CO2	PO1, PO2, PO3, PO5, PO10, PO12	Importing modules, Creating user-defined modules, Packages and package structure, Using standard Python modules

4.1 Importing Modules

Definition

A module is simply a Python file (with .py extension) that contains functions, variables, and classes you can reuse in other programs. Importing a module brings its contents into your program.

Real-World Scenario: Toolbox at Home

A module is like a toolbox. Instead of making a hammer every time,

you open the toolbox (import math) and use the hammer (math.sqrt).

Different toolboxes serve different jobs:

math — calculations

random — random numbers

os — file/folder operations

Example 1: Four Ways to Import

```
# 1. Whole module
import math
print(math.sqrt(25))      # 5.0

# 2. Specific items
from math import pi, sqrt
print(pi, sqrt(16))       # 3.14159..., 4.0

# 3. With alias
import math as m
print(m.factorial(5))     # 120

# 4. Everything (use sparingly)
from math import *
print(cos(0))             # 1.0
```

Example 2: Random Number Generation

```
import random
print(random.randint(1, 100))           # any 1..100
print(random.choice(["red","blue"]))    # random pick
print(random.sample(range(1,50), 5))    # 5 unique nums
```

Example 3: Date and Time

```
import datetime
today = datetime.date.today()
now   = datetime.datetime.now()
print("Today:", today)
print("Now:", now.strftime("%d-%m-%Y %H:%M"))
```

4.2 Creating User-Defined Modules

You can create your own module by saving Python code in a .py file and importing it into another program.

Step 1 — Create the module

File: my_math.py

```
# my_math.py
def add(a, b):
    return a + b

def multiply(a, b):
    return a * b

PI = 3.14159
```

Step 2 — Use the module

File: main.py (same folder)

```
# main.py
import my_math

print(my_math.add(5, 3))          # 8
print(my_math.multiply(4, 6))     # 24
print(my_math.PI)                 # 3.14159
```

Example 1: User Module — calculator.py

```
# calculator.py
def add(a, b):       return a + b
def sub(a, b):       return a - b
def mul(a, b):       return a * b
def div(a, b):       return a / b if b != 0 else None
```

```
# main.py
from calculator import add, mul

print(add(10, 5))     # 15
print(mul(4, 7))      # 28
```

How Python Finds Modules

```
   import my_math
        │
        ▼
   ┌──────────────────────────────────────┐
   │ Python searches in this order:       │
   │   1. Current working folder          │
   │   2. PYTHONPATH environment variable │
   │   3. Standard library locations      │
   │   4. site-packages (installed pkgs)  │
   └──────────────────────────────────────┘
```

4.3 Packages and Package Structure

Definition

A package is a folder that contains multiple Python modules, marked by a special file named __init__.py. Packages help organize large projects by grouping related modules into folders.

Real-World Scenario: Library with Sections

A package is like a library building, modules are bookshelves, and functions are individual books:

college/ ← package (library)

students.py ← module (shelf)

teachers.py ← module (shelf)

results/ ← sub-package (sub-section)

Typical Package Layout

```
my_project/
│
├── main.py
└── college/                ←  package (folder)
    ├── __init__.py         ←  marks it as a package
    ├── students.py         ←  module
    ├── teachers.py         ←  module
    └── results/            ←  sub-package
        ├── __init__.py
        └── exams.py
```

Importing From a Package

```
# main.py
from college import students
from college.results import exams

students.list_all()
exams.show_topper()
```

Note

From Python 3.3+, __init__.py is technically optional, but include it for clarity. You can also place package-level initialization code inside it.

4.4 Using Standard Python Modules

Python's standard library has hundreds of ready-made modules. Here are some you'll use often:

Module	Purpose	Example
math	Math functions / constants	`math.sqrt(16) → 4.0`
random	Random numbers	`random.randint(1, 6)`
datetime	Dates and times	`datetime.date.today()`
os	Files / folders / OS info	`os.getcwd()`
sys	Interpreter & system info	`sys.version`
statistics	Mean, median, stdev	`statistics.mean([1,2,3])`
json	Read/write JSON data	`json.dumps(data)`
time	Pause and measure time	`time.sleep(1)`

Example 1: Working with os Module

```
import os
print(os.getcwd())              # current folder
print(os.listdir("."))          # files in folder
os.mkdir("new_folder")          # create folder
```

Example 2: Statistics Module

```
import statistics as st
marks = [78, 88, 92, 67, 85]
print(st.mean(marks))           # 82.0
print(st.median(marks))         # 85
print(st.stdev(marks))          # standard deviation
```

Week 4 Cheat Sheet

- Module = a .py file you can import.
- Package = a folder of modules with __init__.py.
- import x, from x import y, import x as alias — 3 main styles.
- Python searches: current folder → PYTHONPATH → stdlib → site-packages.
- Useful stdlib modules: math, random, datetime, os, sys, statistics, json.

Week 4 Review Questions

22. Differentiate between a module and a package with an example each.
23. Write a Python module calculator.py with add, sub, mul, div, then import and use it.
24. What is the purpose of __init__.py inside a package folder?
25. Name any four standard Python modules and one function from each.
26. Explain three ways of importing a module with syntax.

Week 5: Object Oriented Programming Basics

Course Outcome	Program Outcomes	Topics Covered
CO3	PO1, PO2, PO3, PO4, PO5, CO10, PO12	Classes and objects, Instance and class variables, Methods and constructors

5.1 Classes and Objects

Definition

Object-Oriented Programming (OOP) is a way of writing programs by combining data and the operations on that data into a single unit called an object. A class is a blueprint that describes what attributes (data) and methods (behaviour) an object will have. An object is an actual instance created from that blueprint.

Why OOP?

Real-world things (a car, a student, a bank account) have BOTH properties (colour, name, balance) AND actions (drive, study, withdraw). OOP lets you model them naturally, instead of keeping data and logic separate.

Real-World Scenario: Class = Architect's Blueprint, Object = Actual House

An architect draws ONE blueprint of a 2BHK house — the class.

From that single blueprint, many real houses are built — the objects.

Each house can have a different colour, different family living inside,

but the structure (rooms, doors, windows) is the same.

➜ Blueprint = class Student

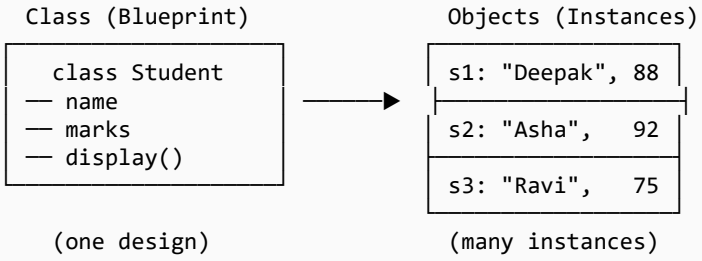
➜ House #1 = s1 = Student("Deepak", 88)

➜ House #2 = s2 = Student("Asha", 92)

Syntax

```
class ClassName:
    # Constructor: runs automatically on object creation
    def __init__(self, params):
        self.attribute = value

    # Method (function inside a class)
    def method_name(self):
        # do something
        pass

# Create an object
obj = ClassName(arguments)
```

Class vs Object Diagram



Example 1: Student Class — The Classic First Example

```
class Student:
    def __init__(self, name, marks):
        self.name = name              # attribute
        self.marks = marks            # attribute

    def display(self):                # method
        print(self.name, "scored", self.marks)

# Create three objects from the same class
```

```
s1 = Student("Deepak", 88)
s2 = Student("Asha",   92)
s3 = Student("Ravi",   75)

s1.display()    # Deepak scored 88
s2.display()    # Asha scored 92
s3.display()    # Ravi scored 75
```

Example 2: Car Class — Real-World Object

```
class Car:
    def __init__(self, brand, model, fuel):
        self.brand = brand
        self.model = model
        self.fuel = fuel          # tank in litres

    def drive(self, km):
        consumed = km / 15        # 15 km per litre
        self.fuel -= consumed
        print(f"{self.brand} drove {km} km. Fuel left: {self.fuel:.2f} L")

    def refuel(self, litres):
        self.fuel += litres
        print(f"Refuelled. Total fuel: {self.fuel} L")

c1 = Car("Maruti", "Swift", 30)
c1.drive(60)        # 26.00 L left
c1.drive(45)        # 23.00 L left
c1.refuel(10)       # 33.00 L
```

Example 3: BankAccount — Behaviour Tied to Data

```
class BankAccount:
    def __init__(self, holder, balance=0):
        self.holder = holder
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount
        print(f"Deposited ₹{amount}. New balance: ₹{self.balance}")

    def withdraw(self, amount):
        if amount > self.balance:
            print("Insufficient balance")
        else:
            self.balance -= amount
            print(f"Withdrew ₹{amount}. New balance: ₹{self.balance}")
```

```
acc = BankAccount("Deepak", 5000)
acc.deposit(2000)       # ₹7000
acc.withdraw(1500)      # ₹5500
acc.withdraw(10000)     # Insufficient balance
```

Example 4: Rectangle — Geometry Class

```
class Rectangle:
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width

    def perimeter(self):
        return 2 * (self.length + self.width)

r = Rectangle(5, 3)
print("Area:",      r.area())          # 15
print("Perimeter:", r.perimeter())     # 16
```

5.2 Instance Variables and Class Variables

Instance Variables

Instance variables belong to ONE specific object. Each object has its own copy. They are defined inside __init__ using self.name = value.

Class Variables

Class variables are SHARED by ALL objects of the class. They are defined directly inside the class body (outside any method).

Diagram: Shared vs Independent Variables

```
class Student:
    college = "RVSCET"        ← CLASS variable (shared)
    def __init__(self, name):
        self.name = name      ← INSTANCE variable (unique)


┌──────────────────────────────────────────┐
│            college = "RVSCET"            │  ← SHARED
│   (one copy belongs to the class itself) │
└──────────────────────────────────────────┘
          │                    │
          ▼                    ▼
┌──────────────────┐  ┌──────────────────┐
│ s1.name="Deepak" │  │ s2.name="Asha"   │  ← UNIQUE
└──────────────────┘  └──────────────────┘
   (own copy each)      (own copy each)
```

Example 1: College Name — Shared by All Students

```
class Student:
    college = "RVSCET"        # class variable (shared)

    def __init__(self, name):
        self.name = name      # instance variable (unique)

s1 = Student("Deepak")
s2 = Student("Asha")

print(s1.name, s1.college)    # Deepak RVSCET
print(s2.name, s2.college)    # Asha   RVSCET

# Change for ALL at once
Student.college = "RVS Tech"
print(s1.college, s2.college) # RVS Tech RVS Tech
```

Example 2: Counting Objects with a Class Variable

```
class Employee:
    count = 0                     # class variable

    def __init__(self, name):
        self.name = name
        Employee.count += 1       # update shared counter

e1 = Employee("Asha")
e2 = Employee("Ravi")
e3 = Employee("Deepak")

print("Total employees:", Employee.count)   # 3
```

Example 3: Default Configuration

```
class Car:
    wheels = 4                    # all cars have 4 wheels

    def __init__(self, brand):
        self.brand = brand

c1 = Car("Maruti")
c2 = Car("Honda")
print(c1.wheels, c2.wheels)       # 4 4
```

Tip

Use class variables for data that is the same for ALL objects (like college name, gravity constant, default tax rate). Use instance variables for data UNIQUE to each object.

5.3 Methods and Constructors

Constructor (__init__)

A constructor is a special method named __init__ that is called automatically when an object is created. It is used to initialize (set up) the object's attributes. The double underscores ("dunder") make it special — Python knows to call it on object creation.

Methods

A method is a function defined inside a class. The first parameter is always self, which refers to the object that calls the method. Through self, the method can read and modify that object's attributes.

Diagram: How self Works

```
    s1 = Student("Deepak", 88)
    s1.display()
          |
          |   Python automatically converts this to:
          ▼
    Student.display(s1)        ← s1 is passed as 'self'
                                 inside display(self), self
                                 now points to s1's data
                                 so self.name = "Deepak"
```

Example 1: Method Updating Object's Data

```
class Counter:
    def __init__(self):
        self.value = 0

    def increment(self):
        self.value += 1

    def reset(self):
        self.value = 0

c = Counter()
c.increment()
c.increment()
c.increment()
print(c.value)     # 3
c.reset()
print(c.value)     # 0
```

Example 2: Constructor with Default Values

```
class Book:
    def __init__(self, title, author, pages=100):
        self.title = title
        self.author = author
        self.pages = pages

    def summary(self):
        print(f"'{self.title}' by {self.author} - {self.pages} pages")

b1 = Book("Python Basics", "Lutz")          # default 100 pages
b2 = Book("Data Science", "McKinney", 500)

b1.summary()        # 'Python Basics' by Lutz - 100 pages
b2.summary()        # 'Data Science' by McKinney - 500 pages
```

Example 3: Multiple Methods Interacting

```
class Mobile:
    def __init__(self, brand, battery=100):
        self.brand = brand
        self.battery = battery

    def use(self, hours):
        drain = hours * 10
        self.battery -= drain
        if self.battery < 0: self.battery = 0
        print(f"After {hours}h use: battery = {self.battery}%")

    def charge(self, hours):
        gain = hours * 25
        self.battery += gain
        if self.battery > 100: self.battery = 100
        print(f"After {hours}h charge: battery = {self.battery}%")

m = Mobile("Samsung")
m.use(3)            # 70%
m.use(5)            # 20%
m.charge(2)         # 70%
m.charge(3)         # 100%
```

❌ Bad Code (Avoid)	✅ Good Code (Preferred)

`# Procedural – data and logic separate`
`length = 5`
`width = 3`
`def area(l, w):`
`  return l * w`
`print(area(length, width))`
	`# OOP – data + logic together`
	`class Rectangle:`
	`  def __init__(self, l, w):`
	`  self.l, self.w = l, w`
	`  def area(self):`
	`  return self.l * self.w`
	`print(Rectangle(5, 3).area())`

> **Important**
> Always use self.attribute inside a method to access instance data — plain 'attribute' will give NameError. self IS the object.

Week 5 Cheat Sheet

- Class = blueprint; Object = instance built from the blueprint.
- __init__ runs automatically on object creation (constructor).
- self refers to the current object inside any method.
- Instance variable: unique to each object (self.x).
- Class variable: shared by all objects (ClassName.x).
- Methods are functions inside a class; first parameter is always self.

Week 5 Review Questions

27. Define class and object. Write a Python class Book with title and author, then create two book objects.
28. Differentiate between instance variable and class variable with example.
29. What is a constructor in Python? Why is it named __init__?
30. Why is self required as the first parameter of every instance method?
31. Write a class Circle with radius as attribute and area() and circumference() as methods.
32. Write a class Mobile that tracks battery level with use() and charge() methods.

Week 6: Advanced OOP Concepts

Course Outcome	Program Outcomes	Topics Covered
CO3	PO1, PO2, PO3, PO4, PO5, CO10, PO12	Inheritance, Method overriding, Polymorphism, Encapsulation

6.1 Inheritance

Definition

Inheritance is the OOP mechanism by which one class (child / derived) automatically acquires the attributes and methods of another class (parent / base). It promotes code reuse — you write common features once in the parent and extend them in children.

Real-World Scenario: Family Inheritance

A child inherits features from parents (eye colour, surname, height)

but also has their own unique traits.

➜ Parent class Animal has eat(), sleep()

➜ Child class Dog gets eat() + sleep() FOR FREE

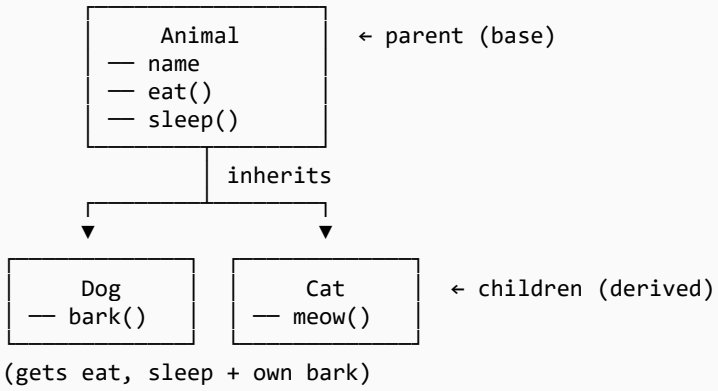
➜ Dog also adds its OWN method: bark()

➜ Child class Cat gets eat() + sleep() too, adds meow()

Syntax

```
class Parent:
    # parent members
    pass

class Child(Parent):
    # child inherits all of Parent
    # may add its own members
    pass
```

Inheritance Diagram



Example 1: Animal → Dog (Basic Inheritance)

```
class Animal:
    def __init__(self, name):
        self.name = name
    def eat(self):
        print(self.name, "is eating")

class Dog(Animal):              # Dog inherits Animal
    def bark(self):
        print(self.name, "says Woof!")
```

```
d = Dog("Tommy")
d.eat()      # Tommy is eating    (inherited)
d.bark()     # Tommy says Woof!   (own)
```

Example 2: Vehicle → Car (Adding Features)

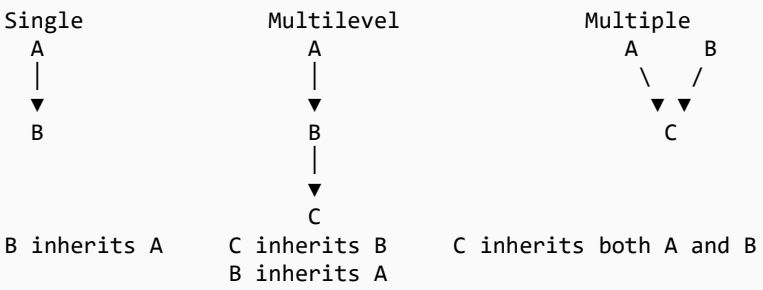
```
class Vehicle:
    def __init__(self, brand, speed):
        self.brand = brand
        self.speed = speed
    def start(self):
        print(self.brand, "started")

class Car(Vehicle):                 # Car inherits Vehicle
    def __init__(self, brand, speed, ac):
        super().__init__(brand, speed)   # call parent's __init__
        self.ac = ac
    def cool(self):
        print(f"{self.brand}: AC at {self.ac}°C")

c = Car("Maruti", 120, 22)
c.start()               # Maruti started   (inherited)
c.cool()                # Maruti: AC at 22°C  (own)
```

Example 3: Person → Student → PostGradStudent (Multilevel)

```
class Person:
    def __init__(self, name):
        self.name = name
    def greet(self):
        print("Hi, I am", self.name)

class Student(Person):
    def __init__(self, name, course):
        super().__init__(name)
        self.course = course
    def study(self):
        print(self.name, "studies", self.course)

class PostGradStudent(Student):
    def __init__(self, name, course, thesis):
        super().__init__(name, course)
        self.thesis = thesis
    def research(self):
        print(self.name, "is researching:", self.thesis)

p = PostGradStudent("Deepak", "MCA", "AI in Education")
```

```
p.greet()       # from Person
p.study()       # from Student
p.research()    # from PostGradStudent
```

Types of Inheritance



Example 4: Multiple Inheritance

```
class Father:
    def gardening(self):
        print("Father loves gardening")

class Mother:
    def cooking(self):
        print("Mother loves cooking")

class Child(Father, Mother):        # inherits BOTH
    def playing(self):
        print("Child loves playing")

c = Child()
c.gardening()    # from Father
c.cooking()      # from Mother
c.playing()      # own
```

6.2 Method Overriding

Definition

Method overriding happens when a child class defines a method with the same name as one in its parent class. The child's version REPLACES the parent's version when called on a child object. This is the basis for runtime polymorphism.

Real-World Scenario: Different Animals, Same Action

All animals make a sound, but each species has its OWN sound.

➜ Parent Animal has sound() that prints "Some sound"

➜ Dog overrides sound() to print "Woof!"

➜ Cat overrides sound() to print "Meow"

Same method name, different behaviour per subclass.

Example 1: Animal Sound Override

```
class Animal:
    def sound(self):
        print("Some sound")

class Dog(Animal):
    def sound(self):                # overrides parent's sound
        print("Woof!")

class Cat(Animal):
    def sound(self):
        print("Meow")

Animal().sound()    # Some sound
Dog().sound()       # Woof!
Cat().sound()       # Meow
```

Example 2: Using super() to Extend Parent Method

```
class Animal:
    def sound(self):
        print("Animals make sounds")

class Dog(Animal):
    def sound(self):
        super().sound()             # call parent FIRST
        print("Specifically: Woof!")

Dog().sound()
# Animals make sounds
# Specifically: Woof!
```

Example 3: Shape → Specific Shapes (Real Use)

```
class Shape:
    def area(self):
        return 0

class Circle(Shape):
    def __init__(self, r):
```

```
        self.r = r
    def area(self):
        return 3.14 * self.r * self.r

class Square(Shape):
    def __init__(self, s):
        self.s = s
    def area(self):
        return self.s * self.s

shapes = [Circle(5), Square(4)]
for s in shapes:
    print("Area:", s.area())
# Area: 78.5
# Area: 16
```

6.3 Polymorphism

Definition

Polymorphism means "many forms" — the same function or operator behaves DIFFERENTLY depending on the object it acts on. In Python, polymorphism is achieved naturally through method overriding and duck typing ("if it walks like a duck and quacks like a duck, treat it as a duck").

Real-World Scenario: One Button, Many Devices

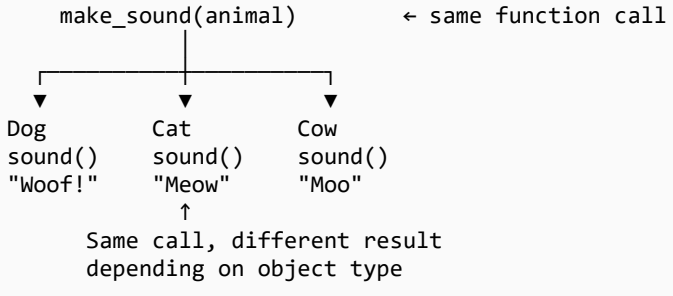
Press the POWER button on a TV → TV turns on.

Press the POWER button on an AC → AC turns on.

Press the POWER button on a fan → fan turns on.

Same action (press power), different behaviour per device — polymorphism.

Polymorphism Diagram



Example 1: Different Animals, Same Function

```
class Dog:
```

```
    def sound(self):
        print("Woof!")

class Cow:
    def sound(self):
        print("Moo")

class Duck:
    def sound(self):
        print("Quack")

def make_sound(animal):
    animal.sound()           # same call, different behaviour

make_sound(Dog())            # Woof!
make_sound(Cow())            # Moo
make_sound(Duck())           # Quack
```

Example 2: Polymorphism in a Loop

```
class Bird:
    def fly(self):
        print("Flying high in the sky")

class Penguin:
    def fly(self):
        print("Sorry, I can't fly – I swim!")

class Sparrow:
    def fly(self):
        print("Fluttering between branches")

for b in [Bird(), Penguin(), Sparrow()]:
    b.fly()
```

Example 3: Operator Polymorphism (Built-in)

```
# The '+' operator behaves differently based on operand types
print(3 + 5)              # 8         (numeric addition)
print("py" + "thon")      # python    (string concatenation)
print([1,2] + [3,4])      # [1,2,3,4] (list concatenation)
```

Example 4: len() Function — Polymorphic

```
print(len("Python"))            # 6
print(len([1, 2, 3, 4]))        # 4
print(len({"a":1, "b":2}))      # 2
# Same function, works on different types
```

6.4 Encapsulation

Definition

Encapsulation is the practice of binding data and the methods that work on that data into one unit (a class), AND restricting direct access to some of that data from outside. This prevents accidental corruption of internal state.

Real-World Scenario: ATM Machine

You can deposit and withdraw money — but you can NEVER

directly open the ATM and change the cash balance.

The cash is HIDDEN; you interact only through buttons (methods).

➜ balance is private (cannot touch directly)

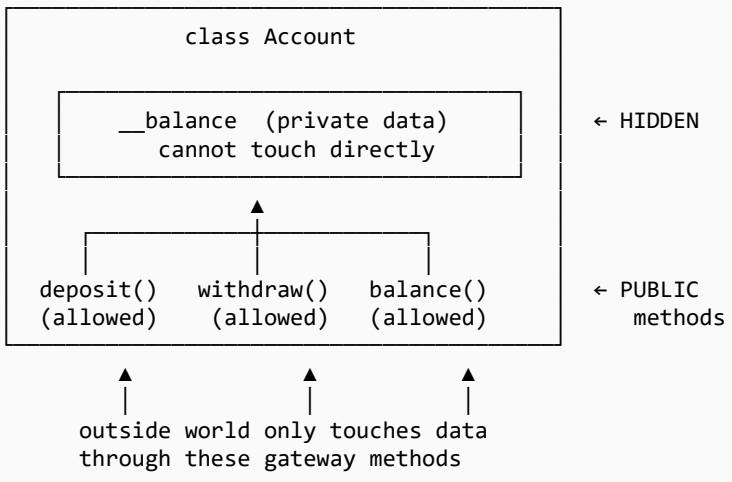
➜ withdraw() and deposit() are public methods

This protects the data from misuse.

Access Levels in Python

Naming	Access Level	Meaning
name	Public	Accessible anywhere
_name	Protected (convention)	Intended for class & subclasses only
__name	Private (name-mangled)	Hard to access from outside

Encapsulation Diagram



Example 1: Bank Account with Private Balance

```
class Account:
    def __init__(self, owner, balance):
        self.owner = owner             # public
        self.__balance = balance       # private (note __)

    def deposit(self, amt):
        if amt > 0:
            self.__balance += amt

    def withdraw(self, amt):
        if 0 < amt <= self.__balance:
            self.__balance -= amt
        else:
            print("Invalid amount")

    def get_balance(self):             # gateway method
        return self.__balance

a = Account("Deepak", 1000)
a.deposit(500)
print(a.get_balance())      # 1500
# print(a.__balance)        # ❌ AttributeError
a.withdraw(200)
print(a.get_balance())      # 1300
```

Example 2: Student Grade — Validated Access

```
class Student:
    def __init__(self, name):
        self.name = name
        self.__marks = 0

    def set_marks(self, m):
        if 0 <= m <= 100:
            self.__marks = m
        else:
            print("Invalid marks (0-100 only)")

    def get_marks(self):
        return self.__marks

s = Student("Asha")
s.set_marks(95)
print(s.get_marks())    # 95
s.set_marks(150)        # Invalid marks
s.set_marks(-10)        # Invalid marks
print(s.get_marks())    # 95 (unchanged)
```

Example 3: Using @property Decorator (Pythonic)

```
class Circle:
    def __init__(self, radius):
        self.__radius = radius

    @property
    def radius(self):
        return self.__radius

    @radius.setter
    def radius(self, value):
        if value > 0:
            self.__radius = value
        else:
            print("Radius must be positive")

c = Circle(5)
print(c.radius)         # 5  (looks like attribute, calls method)
c.radius = 10
print(c.radius)         # 10
c.radius = -3           # Radius must be positive
```

❌ **Bad Code (Avoid)**

```
# Data exposed — anyone can corrupt
class Account:
    def __init__(self, bal):
        self.balance = bal

a = Account(1000)
a.balance = -9999     # ❌ invalid but allowed!
```

✅ **Good Code (Preferred)**

```
# Data protected — must go through method
class Account:
    def __init__(self, bal):
        self.__balance = bal
    def deposit(self, amt):
        if amt > 0:
            self.__balance += amt
# negative deposits silently rejected
```

Four Pillars of OOP — Summary

Pillar	Meaning	Example
Inheritance	Reuse parent's code in child	Dog(Animal) gets eat()
Polymorphism	Same call, many forms	make_sound(Dog) vs make_sound(Cat)
Encapsulation	Hide internal data; expose via methods	__balance + deposit()
Abstraction	Show only essentials; hide complexity	ATM buttons hide circuits

Week 6 Cheat Sheet

- Inheritance — child class reuses parent's attributes & methods.
- Use super() to call parent's version from the child.
- Override — child redefines a parent method with same name.
- Polymorphism — same call, different behaviour by object type.
- Encapsulation — hide internal data using __private; expose methods.
- Types: single, multilevel, multiple, hierarchical, hybrid.

Week 6 Review Questions

33. Define inheritance. Explain single and multilevel inheritance with example.
34. What is method overriding? How is it different from method overloading?
35. Explain polymorphism with a real-world example using two classes.
36. How does Python implement encapsulation? Explain public, protected, private.
37. Write a program where Vehicle is a parent class and Car & Bike are its children, each overriding a description() method.
38. Create a class BankAccount with private __balance and methods deposit, withdraw and get_balance.

Week 7: Special Methods and Iterators

Course Outcome	Program Outcomes	Topics Covered
CO3	PO1, PO2, PO3, PO4, PO5, CO10, PO12	Magic methods, Operator overloading, Iterators and generators

7.1 Magic (Dunder) Methods

Definition

Magic methods (also called "dunder" methods because they start and end with double underscores) are special methods that Python calls AUTOMATICALLY in response to certain operations — creating an object, printing it, comparing it, adding it, finding its length, etc. You don't usually call them directly; you simply define them, and Python invokes them when needed.

Real-World Scenario: Doorbell vs Walking In

Imagine your home has a doorbell. When someone presses it, you go

answer automatically — you don't have to be told.

Magic methods work the same way:

➜ print(obj) rings the __str__ doorbell

➜ len(obj) rings the __len__ doorbell

➜ obj1 + obj2 rings the __add__ doorbell

➜ obj1 == obj2 rings the __eq__ doorbell

Python rings; your method answers.

Common Magic Methods

Method	Triggered By	Purpose
`__init__`	Object creation	Initialize attributes
`__str__`	str(obj), print(obj)	User-friendly string
`__repr__`	repr(obj), debugger output	Developer string
`__len__`	len(obj)	Length / size
`__add__`	obj1 + obj2	Define + behaviour
`__sub__`	obj1 - obj2	Define − behaviour
`__mul__`	obj1 * obj2	Define × behaviour
`__eq__`	obj1 == obj2	Equality check
`__lt__`	obj1 < obj2	Less-than check
`__getitem__`	obj[i]	Indexing support
`__iter__`	for x in obj	Make iterable
`__call__`	obj()	Make object callable like function

Example 1: __str__ — Beautiful Print Output

```
class Book:
    def __init__(self, title, pages):
        self.title = title
        self.pages = pages

    def __str__(self):
        return f"Book: {self.title} ({self.pages} pages)"

b = Book("Python", 350)
print(b)            # Book: Python (350 pages)  ← uses __str__
```

```
# Without __str__, print would show:
#   <__main__.Book object at 0x000001E2F4B1A8>
```

Example 2: __len__ — Make len() Work

```
class Playlist:
    def __init__(self, songs):
        self.songs = songs

    def __len__(self):
        return len(self.songs)

p = Playlist(["Song A", "Song B", "Song C"])
print(len(p))        # 3   ← uses __len__
```

Example 3: __eq__ — Custom Equality

```
class Student:
    def __init__(self, roll, name):
        self.roll = roll
        self.name = name

    def __eq__(self, other):
        return self.roll == other.roll   # equal if same roll number

s1 = Student(101, "Deepak")
s2 = Student(101, "DEEPAK KUMAR")        # same roll
s3 = Student(102, "Asha")
print(s1 == s2)      # True
print(s1 == s3)      # False
```

Example 4: __str__ vs __repr__

```
class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __str__(self):           # for end-users
        return f"({self.x}, {self.y})"

    def __repr__(self):          # for developers / debugging
        return f"Point(x={self.x}, y={self.y})"

p = Point(3, 4)
print(p)        # (3, 4)               ← __str__
print(repr(p))  # Point(x=3, y=4)      ← __repr__
```

7.2 Operator Overloading

Definition

Operator overloading means giving extra meaning to a built-in operator (like +, −, *, ==) so it works with objects of YOUR own class. This is achieved by defining the matching magic method. The result is intuitive, readable code: p1 + p2 instead of p1.add(p2).

Real-World Scenario: Adding Points on a Map

If you have two coordinates (3,4) and (1,2), "adding" them should logically give (4,6). Built-in '+' doesn't know how to add Point objects — so we teach it by defining __add__.

Operator → Magic Method Map

Operator	Magic Method	Use
+	__add__	obj1 + obj2
-	__sub__	obj1 − obj2
*	__mul__	obj1 * obj2
/	__truediv__	obj1 / obj2
==	__eq__	obj1 == obj2
<	__lt__	obj1 < obj2
>	__gt__	obj1 > obj2
>=	__ge__	obj1 >= obj2

Example 1: Adding Two Points

```
class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __add__(self, other):
        return Point(self.x + other.x, self.y + other.y)

    def __str__(self):
        return f"({self.x}, {self.y})"

p1 = Point(2, 3)
p2 = Point(4, 5)
p3 = p1 + p2            # calls __add__
print(p3)               # (6, 8)
```

Example 2: Vector Class — Multiple Operators

```
class Vector:
    def __init__(self, a, b):
        self.a, self.b = a, b

    def __add__(self, other):
        return Vector(self.a + other.a, self.b + other.b)

    def __sub__(self, other):
        return Vector(self.a - other.a, self.b - other.b)

    def __mul__(self, k):                # scalar multiply
        return Vector(self.a * k, self.b * k)

    def __str__(self):
        return f"<{self.a}, {self.b}>"

v1 = Vector(1, 2)
v2 = Vector(3, 4)
print(v1 + v2)         # <4, 6>
print(v2 - v1)         # <2, 2>
print(v1 * 3)          # <3, 6>
```

Example 3: Money Class — Real-World Use

```
class Money:
    def __init__(self, amount):
        self.amount = amount

    def __add__(self, other):
        return Money(self.amount + other.amount)

    def __sub__(self, other):
        return Money(self.amount - other.amount)

    def __lt__(self, other):              # less than
        return self.amount < other.amount

    def __str__(self):
        return f"₹{self.amount}"

a = Money(500)
b = Money(300)
print(a + b)            # ₹800
print(a - b)            # ₹200
print(a < b)            # False
```

Example 4: Comparing Students by Marks

```
class Student:
    def __init__(self, name, marks):
        self.name, self.marks = name, marks

    def __lt__(self, other):
        return self.marks < other.marks

    def __str__(self):
        return f"{self.name}({self.marks})"

a = Student("Deepak", 88)
b = Student("Asha", 92)
print(a < b)             # True
# Now sorted() works automatically:
print(sorted([Student("R", 75), a, b]))
# [R(75), Deepak(88), Asha(92)]
```

7.3 Iterators and Generators

Iterator

Definition

An iterator is an object that produces values ONE AT A TIME using the next() function. Internally, it uses two magic methods: __iter__ (returns the iterator object) and __next__ (returns the next value; raises StopIteration when finished). Every Python for-loop quietly uses iterators behind the scenes.

Real-World Scenario: Ticket Counter Queue

At a railway counter, only ONE person is served at a time.

The clerk calls "Next!" and the next person steps up.

When the queue is empty, the counter closes (StopIteration).

➜ next(iterator) is like calling "Next!"

Example 1: Built-in Iterator from a List

```
nums = [10, 20, 30]
it = iter(nums)             # get an iterator
print(next(it))             # 10
print(next(it))             # 20
print(next(it))             # 30
# print(next(it))           # ❌ StopIteration
```

Example 2: Custom Iterator Class

```
class Counter:
    def __init__(self, limit):
        self.n = 0
        self.limit = limit

    def __iter__(self):
        return self

    def __next__(self):
        if self.n >= self.limit:
            raise StopIteration
        self.n += 1
        return self.n

for v in Counter(5):
    print(v, end=" ")        # 1 2 3 4 5
```

How a for-loop Works Internally

```
for x in [10, 20, 30]:
    print(x)
```

```
Python silently does:

it = iter([10,20,30])
while True:
    try:
        x = next(it)
        print(x)
    except StopIteration:
        break
```

Generator

Definition

A generator is a much simpler way to create an iterator — it is just a function that uses yield instead of return. Each yield pauses the function and returns a value; the next call to next() resumes from there. Generators save memory because values are produced one at a time, ON DEMAND.

Real-World Scenario: Tea-Stall Owner Making Cups On Demand

A tea-stall owner doesn't pre-make 1,000 cups in the morning.

He makes ONE cup when a customer asks, then waits.

That's exactly how a generator works — produce when needed,

saving water, milk, gas, and counter space (memory).

Example 1: Simple Counter Generator

```
def counter(limit):
    n = 1
    while n <= limit:
        yield n           # pause here, return n
        n += 1

for v in counter(5):
    print(v, end=" ")     # 1 2 3 4 5
```

Example 2: Fibonacci Generator

```
def fib(count):
    a, b = 0, 1
    for _ in range(count):
        yield a
        a, b = b, a + b

print(list(fib(10)))
# [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Example 3: Reading Huge File Line-by-Line

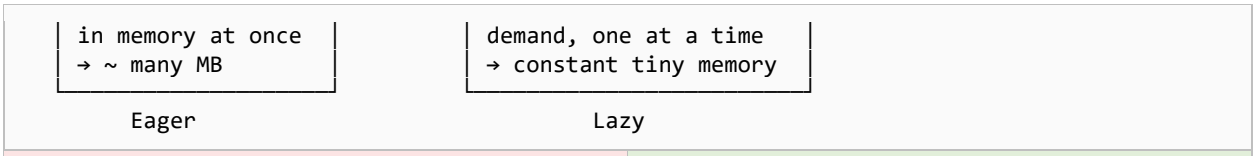
```
def read_lines(filename):
    with open(filename) as f:
        for line in f:
            yield line.strip()

# Even if file has 10 million lines, only ONE is in memory
for ln in read_lines("big_log.txt"):
    process(ln)
```

Example 4: Generator Expression (One-liner)

```
# Like list comprehension but with ( ) instead of [ ]
squares_gen = (x*x for x in range(1, 6))
print(next(squares_gen))     # 1
print(next(squares_gen))     # 4

# Use in sum() — no list ever built
total = sum(x*x for x in range(1, 1000001))
print(total)
```

Generator vs List (Memory Comparison)

```
  List of 1..1,000,000          Generator for 1..1,000,000
 ┌──────────────────┐          ┌──────────────────────────┐
 │ All numbers stored │          │ Numbers produced on      │
```



❌ **Bad Code (Avoid)**

```
# Eager – builds whole list first
def squares(n):
    result = []
    for i in range(n):
        result.append(i*i)
    return result
big = squares(10_000_000)    # huge memory!
```

✅ **Good Code (Preferred)**

```
# Lazy generator – one at a time
def squares(n):
    for i in range(n):
        yield i*i
for x in squares(10_000_000):  # tiny memory
    process(x)
```

Week 7 Cheat Sheet

- Magic methods = __name__; Python calls them automatically.
- __str__ → user-friendly print; __repr__ → developer/debug form.
- Operator overloading via __add__, __sub__, __eq__, __lt__, etc.
- Iterator: object with __iter__ & __next__; raises StopIteration when done.
- Generator: function with yield — produces values lazily, memory-efficient.
- Generator expression: (x for x in range(n)) — like list-comp with ( ).

Week 7 Review Questions

39. List any five magic methods and state when each is called.
40. Write a class Vector(x, y) and overload +, − and == operators.
41. Differentiate between __str__ and __repr__.
42. Write a generator function that yields the first n Fibonacci numbers.
43. How does a generator save memory compared to a list? Explain with example.
44. Create a custom iterator that yields squares of numbers from 1 to n.

Week 8: Exception Handling

Course Outcome	Program Outcomes	Topics Covered
CO4	PO1, PO2, PO3, PO4, PO5, CO10, PO12	Types of errors, try/except/finally blocks, User-defined exceptions

8.1 Types of Errors

Definition

An error is anything that prevents your Python program from running correctly. Errors fall into two broad categories: SYNTAX ERRORS (caught BEFORE the program runs, due to wrong grammar) and EXCEPTIONS (occur WHILE the program is running, due to logical or runtime conditions like dividing by zero or opening a missing file).

Real-World Scenario: Cooking Without Salt vs Burning the Curry

Imagine you're cooking from a recipe:

➜ Recipe is in a language you don't understand → can't start = SYNTAX ERROR

➜ You started cooking but ran out of salt mid-recipe = EXCEPTION

Syntax errors stop you BEFORE; exceptions interrupt you DURING.

Only exceptions can be "handled" — syntax errors must be fixed.

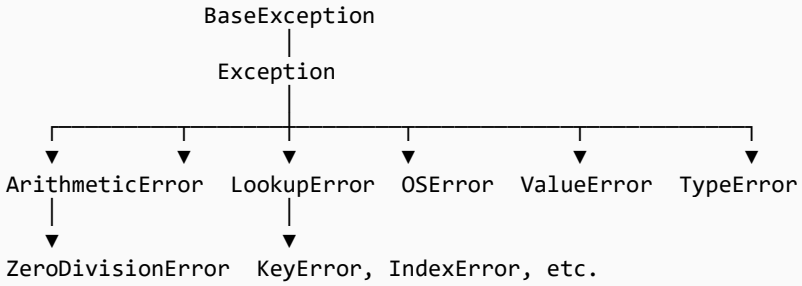
Syntax Error vs Exception

Feature	Syntax Error	Exception
When detected	Before execution	During execution
Cause	Wrong grammar	Logical/runtime issue
Can be handled?	No — must be fixed	Yes — using try/except
Example	if x = 5:	10 / 0 (ZeroDivisionError)

Common Built-in Exceptions

Exception	Raised When
`ZeroDivisionError`	Dividing by zero (10/0)
`ValueError`	Wrong type of value (int("abc"))
`TypeError`	Operation on incompatible types ('a' + 5)
`IndexError`	List index out of range (lst[100])
`KeyError`	Dictionary key not found
`FileNotFoundError`	Trying to open a non-existent file
`NameError`	Variable not defined
`AttributeError`	Attribute does not exist on object
`ImportError`	Failed to import a module
`RuntimeError`	Generic runtime problem

Exception Hierarchy (Simplified)



8.2 try / except / else / finally Blocks

Definition

Exception handling lets your program detect problems at runtime and respond gracefully instead of crashing. Python uses the keywords try, except, else and finally for this. The try block contains code that MIGHT fail; the except block catches and handles the specific error; else runs only if NO error happened; finally ALWAYS runs (used for cleanup like closing files or database connections).

Real-World Scenario: Driving a Car

You start your car (try) — but the engine may not start.

➜ If engine cranks fine → drive (else block)

➜ If battery is dead → call mechanic (except block)

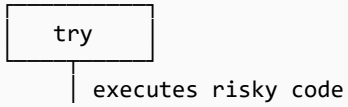
➜ Either way, lock the car when done (finally block)

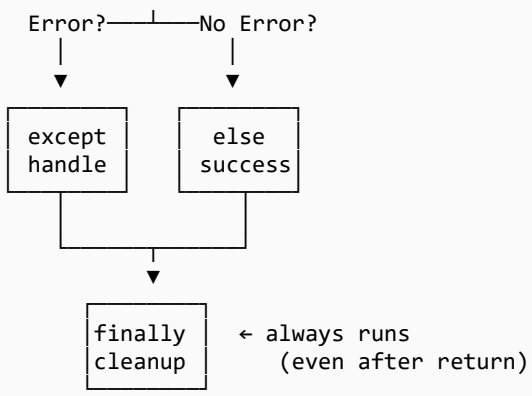
The 'finally' part is GUARANTEED — even if something went wrong.

Syntax

```
try:
    # risky code
except SomeError:
    # runs only if SomeError happened
else:
    # runs only if NO error happened
finally:
    # ALWAYS runs (cleanup)
```

Flow Diagram





Example 1: Division — Handling ZeroDivisionError

```
try:
    a = int(input("Enter numerator: "))
    b = int(input("Enter denominator: "))
    result = a / b
except ZeroDivisionError:
    print("Cannot divide by zero")
except ValueError:
    print("Please enter valid integers")
else:
    print("Result =", result)
finally:
    print("Program finished")
```

Example 2: List Index Out of Range

```
marks = [85, 90, 78]
try:
    print(marks[10])             # ❌ IndexError
except IndexError as e:
    print("Error:", e)
# Error: list index out of range
```

Example 3: File Not Found

```
try:
    f = open("missing.txt", "r")
    data = f.read()
except FileNotFoundError:
    print("Sorry, that file doesn't exist!")
else:
    print(data)
    f.close()
```

Example 4: Multiple Exceptions in One Except

```
try:
    age = int(input("Enter your age: "))
    print(100 / age)
except (ValueError, ZeroDivisionError) as e:
    print("Invalid input:", e)
```

Example 5: Catching ANY Exception (Last Resort)

```
try:
    risky_function()
except Exception as e:
    print("Something went wrong:", type(e).__name__, "-", e)
# Catches almost anything — use sparingly, debug-friendly
```

Example 6: Why finally Matters — Even With return

```
def divide(a, b):
    try:
        return a / b
    except ZeroDivisionError:
        return None
    finally:
        print("Division attempt finished")

print(divide(10, 2))
# Division attempt finished
# 5.0

print(divide(10, 0))
# Division attempt finished
# None
```

Tip

Always catch SPECIFIC exceptions first (ValueError, KeyError) before generic Exception. Catching too broadly hides bugs you'd rather see.

8.3 User-Defined Exceptions

Definition

Sometimes Python's built-in exceptions aren't specific enough for YOUR application's rules. For example: "Age must be between 5 and 120" or "Discount cannot exceed 50%". You can create a custom error type by writing a class that inherits from the built-in Exception class. This makes your code more readable and your errors more meaningful.

Real-World Scenario: College Rules vs General Rules

A college may have its OWN rules in addition to legal rules:

➜ Legal: minimum age 18 to enrol (general)

➜ College: minimum 60% in 12th to enrol in BCA (custom)

The 2nd rule isn't a built-in 'error' in society — the college

DEFINED it. Similarly, you define custom exceptions for YOUR app.

Syntax

```
class MyError(Exception):
    pass

raise MyError("Something specific went wrong")
```

Example 1: Custom Exception — Age Validator

```
class InvalidAgeError(Exception):
    def __init__(self, age, msg="Age must be between 5 and 120"):
        self.age = age
        super().__init__(f"{msg}. Got: {age}")

def set_age(age):
    if age < 5 or age > 120:
        raise InvalidAgeError(age)
    print("Age accepted:", age)

try:
    set_age(150)
except InvalidAgeError as e:
    print("Error:", e)
# Error: Age must be between 5 and 120. Got: 150
```

Example 2: Custom Exception — Insufficient Balance

```
class InsufficientBalanceError(Exception):
    def __init__(self, requested, available):
        super().__init__(
            f"Cannot withdraw ₹{requested}; only ₹{available} available"
        )

class Account:
    def __init__(self, balance):
        self.balance = balance

    def withdraw(self, amt):
```

```
        if amt > self.balance:
            raise InsufficientBalanceError(amt, self.balance)
        self.balance -= amt
        return self.balance

a = Account(1000)
try:
    a.withdraw(5000)
except InsufficientBalanceError as e:
    print("Transaction blocked:", e)
# Transaction blocked: Cannot withdraw ₹5000; only ₹1000 available
```

Example 3: Custom Exception — Password Strength

```
class WeakPasswordError(Exception):
    pass

def set_password(pwd):
    if len(pwd) < 8:
        raise WeakPasswordError("Password too short (min 8 chars)")
    if pwd.isalpha():
        raise WeakPasswordError("Password must contain digits too")
    print("Password set successfully")

try:
    set_password("abc")
except WeakPasswordError as e:
    print("Error:", e)
# Error: Password too short (min 8 chars)
```

Example 4: Raising Built-in Exception Manually

```
def square_root(n):
    if n < 0:
        raise ValueError("Cannot take sqrt of negative number")
    return n ** 0.5

try:
    print(square_root(-9))
except ValueError as e:
    print("Error:", e)
```

❌ Bad Code (Avoid)	✅ Good Code (Preferred)

`# No protection — crashes on bad input`
`n = int(input("Number: "))`
`print(100 / n)`
	`# Robust — handles both errors`
	`try:`
	`  n = int(input("Number: "))`
	`  print(100 / n)`
	`except ValueError:`
	`  print("Not an integer")`

	except ZeroDivisionError: print("Cannot divide by zero")

Week 8 Cheat Sheet

- Syntax error = grammar mistake (must fix); Exception = runtime issue (can handle).
- try / except handles exceptions; else runs on success; finally ALWAYS runs.
- Catch specific exceptions BEFORE generic Exception.
- Use raise SomeError("msg") to throw exceptions manually.
- Create custom errors by inheriting from Exception.
- except (E1, E2) catches multiple types; as e captures the error object.

Week 8 Review Questions

45. Distinguish between syntax error and exception with one example each.
46. Write a Python program that asks the user for two numbers and divides them, handling both ValueError and ZeroDivisionError.
47. Explain the role of else and finally blocks in exception handling.
48. Define a custom exception NegativeNumberError and raise it when input < 0.
49. List any five built-in Python exceptions and state when each occurs.
50. Write a class BankAccount and a custom InsufficientBalanceError when withdrawal exceeds balance.

Week 9: File Handling

Course Outcome	Program Outcomes	Topics Covered
CO4	PO1, PO2, PO3, PO4, PO5, PO10, PO12	Reading and writing text files, Working with CSV files, File handling using with statement

9.1 Reading and Writing Text Files

Definition

A file is permanent storage on disk. Unlike variables (which vanish when the program ends), files keep your data even after shutdown. Python provides built-in functions to create, read, write, and append text files. The basic three-step process is: (1) OPEN the file, (2) READ from or WRITE to it, (3) CLOSE it.

Real-World Scenario: Notebook in Your Bag

A file is like a physical notebook:

➜ open() = take notebook out of bag

➜ write() = write something with a pen

➜ read() = read what's already written

➜ close() = put notebook back in bag

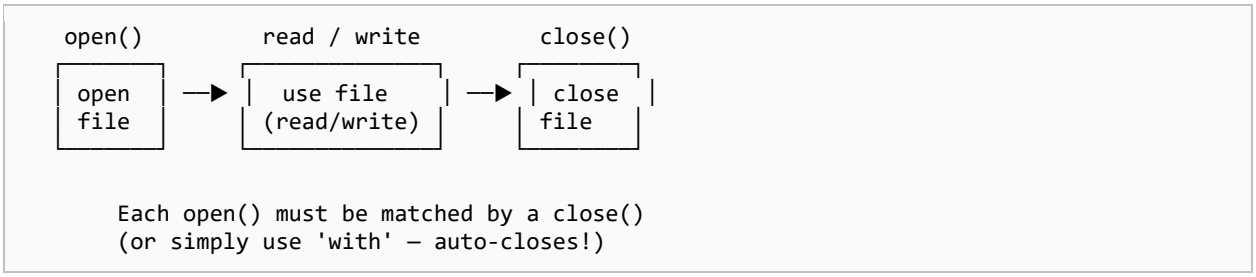
If you forget to close it, pages may flap, get lost, or torn

(in computing terms: file stays locked, data may not be saved).

open() Function and Modes

Mode	Meaning	If File Exists / Missing
`'r'`	Read (default)	Error if missing
`'w'`	Write (overwrites)	Creates new / erases old
`'a'`	Append	Creates new / keeps old
`'x'`	Exclusive create	Error if already exists
`'r+'`	Read & write	Error if missing
`'b'`	Binary (combine: 'rb', 'wb')	For images/PDFs

File Lifecycle Diagram



Example 1: Write to a Text File

```
f = open("notes.txt", "w")
f.write("Hello, Python!\n")
f.write("This is line 2.\n")
f.write("This is line 3.\n")
f.close()
# File 'notes.txt' is created with 3 lines
```

Example 2: Read the Whole File

```
f = open("notes.txt", "r")
content = f.read()           # entire file as one string
print(content)
f.close()
```

Example 3: Read One Line at a Time

```
f = open("notes.txt", "r")
print(f.readline().strip())   # line 1
print(f.readline().strip())   # line 2
print(f.readline().strip())   # line 3
f.close()
```

Example 4: Read All Lines as a List

```
f = open("notes.txt", "r")
lines = f.readlines()         # list of strings
print(lines)
# ['Hello, Python!\n', 'This is line 2.\n', 'This is line 3.\n']
f.close()
```

Example 5: Loop Through Lines

```
f = open("notes.txt", "r")
for line in f:                # most memory-friendly way
    print(line.strip())
f.close()
```

Example 6: Append (Add Without Erasing)

```
f = open("notes.txt", "a")
f.write("This line is added later.\n")
f.close()
# Old content preserved; new line appended at end
```

Example 7: Mini Project — Daily Diary

```
from datetime import date

entry = input("Today's thought: ")
with open("diary.txt", "a") as f:
    f.write(f"[{date.today()}] {entry}\n")

print("Entry saved to diary.txt")

# Read all entries
print("\n--- Your Diary ---")
with open("diary.txt") as f:
```

```
    print(f.read())
```

Read Methods — Quick Reference

Method	Returns	Use When
`read()`	Whole file as one string	Small files
`read(n)`	First n characters	Partial reads
`readline()`	One line as a string	Process line-by-line
`readlines()`	List of all lines	Need all lines at once
`for line in f`	Iterates line by line	Best for huge files

9.2 Working with CSV Files

Definition

CSV (Comma-Separated Values) is a plain-text file format where each row is one line and columns are separated by commas. It is the most common way to exchange tabular data between spreadsheets (Excel, Google Sheets), databases, and programs. Python's built-in csv module handles reading and writing CSV safely (it deals with quoting, embedded commas, newlines, etc.).

Real-World Scenario: Class Attendance Register

A CSV file is just like a paper attendance register:

➜ Each ROW = one student record

➜ Each COLUMN (separated by comma) = name, roll, status

➜ Easy to open in Excel for the staff who don't code

Almost every dataset you'll download from the web (Kaggle, data.gov.in, IMDb) is in CSV format.

Sample CSV Content

```
Name,Course,Marks
Deepak,BCA,88
Asha,BCA,92
Ravi,BCA,75
```

Example 1: Writing CSV with csv.writer

```
import csv
```

```
rows = [
    ["Name", "Course", "Marks"],
    ["Deepak", "BCA", 88],
    ["Asha",   "BCA", 92],
    ["Ravi",   "BCA", 75],
]

with open("students.csv", "w", newline="") as f:
    writer = csv.writer(f)
    writer.writerows(rows)        # writes all rows at once

print("students.csv created")
```

Example 2: Reading CSV with csv.reader

```
import csv

with open("students.csv", "r") as f:
    reader = csv.reader(f)
    for row in reader:
        print(row)

# Output:
# ['Name', 'Course', 'Marks']
# ['Deepak', 'BCA', '88']
# ['Asha',   'BCA', '92']
# ['Ravi',   'BCA', '75']
```

Example 3: CSV as Dictionary (Recommended)

```
import csv

with open("students.csv", "r") as f:
    reader = csv.DictReader(f)        # uses first row as keys
    for row in reader:
        print(row["Name"], "-", row["Marks"])

# Output:
# Deepak - 88
# Asha   - 92
# Ravi   - 75
```

Example 4: Writing CSV with DictWriter

```
import csv

rows = [
    {"Name": "Deepak", "Course": "BCA", "Marks": 88},
```

```
    {"Name": "Asha",   "Course": "BCA", "Marks": 92},
    {"Name": "Ravi",   "Course": "BCA", "Marks": 75},
]

with open("students.csv", "w", newline="") as f:
    writer = csv.DictWriter(f, fieldnames=["Name","Course","Marks"])
    writer.writeheader()
    writer.writerows(rows)
```

Example 5: Filtering — Toppers from CSV

```
import csv

with open("students.csv") as f:
    reader = csv.DictReader(f)
    for row in reader:
        if int(row["Marks"]) >= 85:
            print(row["Name"], "is a topper with", row["Marks"])

# Deepak is a topper with 88
# Asha is a topper with 92
```

Example 6: Add a New Row to Existing CSV

```
import csv

new_student = ["Sita", "BCA", 80]

with open("students.csv", "a", newline="") as f:
    writer = csv.writer(f)
    writer.writerow(new_student)

print("New record added")
```

Tip

Always use newline="" when opening a CSV for write on Windows; otherwise you'll get blank lines between rows. Numeric values read via csv module are strings — convert with int() / float() before calculation.

9.3 File Handling Using the with Statement

Definition

The with statement opens a file and guarantees it will be CLOSED automatically — even if an error occurs inside the block. This is the recommended way to handle files in modern Python. It uses a feature called "context managers" behind the scenes.

Real-World Scenario: Restaurant Self-Service Tray

Imagine carrying your tray at a fast-food restaurant.

Without 'with': you carry it, eat, may FORGET to drop it back.

With 'with': it's like an automatic conveyor belt — once you're

done, the tray is GUARANTEED to be returned, even if you fall.

➜ Code outside with block? Tray returned? Yes, always.

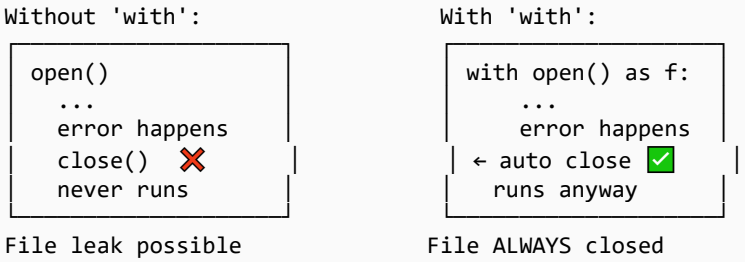
Syntax

```
with open("file.txt", "r") as f:
    data = f.read()
# File is automatically closed here, even on error
```

❌ Bad Code (Avoid)	✅ Good Code (Preferred)

`# Risky – file may stay open on error`
`f = open("data.txt", "r")`
`data = f.read()`
`process(data)  # if this fails,`
`f.close()  # close() never runs`
	`# Safe – file always closed`
	`with open("data.txt", "r") as f:`
	`    data = f.read()`
	`    process(data)`
	`# (auto-closes here)`

Why 'with' is Better



Example 1: Basic Read with 'with'

```
with open("notes.txt", "r") as f:
    for line in f:
        print(line.strip())
# No need to call f.close() – Python does it
```

Example 2: Write with 'with'

```
subjects = ["Maths", "Science", "English", "Hindi"]

with open("subjects.txt", "w") as f:
    for s in subjects:
        f.write(s + "\n")
```

Example 3: Copy a File Using Two with Statements

```
with open("source.txt", "r") as src, open("copy.txt", "w") as dst:
    dst.write(src.read())

print("File copied successfully")
```

Example 4: Count Words in a File

```
with open("notes.txt") as f:
    text = f.read()

words = text.split()
print("Total words:", len(words))
print("Unique words:", len(set(words)))
```

Example 5: Find Lines Containing a Keyword

```
keyword = "Python"

with open("notes.txt") as f:
    for n, line in enumerate(f, start=1):
        if keyword in line:
            print(f"Line {n}: {line.strip()}")
```

Example 6: Mini Project — Student Marks System

```
import csv

# Step 1 – Add students
def add_student():
    name = input("Name: ")
    marks = input("Marks: ")
    with open("marks.csv", "a", newline="") as f:
        csv.writer(f).writerow([name, marks])
    print("Added.")

# Step 2 – Show all
def show_all():
    with open("marks.csv") as f:
        for row in csv.reader(f):
            print(row[0], "-", row[1])

# Step 3 – Find topper
def topper():
    with open("marks.csv") as f:
        rows = list(csv.reader(f))
    best = max(rows, key=lambda r: int(r[1]))
    print("Topper:", best[0], "with", best[1])
```

```
# Driver
add_student()
show_all()
topper()
```

Week 9 Cheat Sheet

- Open with open(filename, mode); modes: r / w / a / x / r+ / b.
- Read methods: read(), readline(), readlines(), or for-loop.
- Write methods: write(text), writelines(list).
- Always use with open(...) as f: — file closes automatically.
- CSV: csv.reader / csv.writer; use DictReader / DictWriter for column names.
- Use newline="" while writing CSV on Windows to avoid blank lines.

Week 9 Review Questions

51. List any four file-opening modes in Python and their use.
52. Write a program that writes 5 names into a file and reads them back.
53. Differentiate between read(), readline() and readlines().
54. Why is the with statement preferred for file handling? Give an example.
55. Write a program that reads students.csv and prints names of students with marks > 80.
56. Create a CSV of 5 books (title, author, price) and find the costliest book.

Week 10: Regular Expressions and String Processing

Course Outcome	Program Outcomes	Topics Covered
CO4	PO1, PO2, PO3, PO4, PO5, PO10, PO12	Regular expression syntax, Pattern matching, String searching and manipulation

10.1 Regular Expression Syntax

Definition

A Regular Expression (regex) is a tiny language for describing TEXT PATTERNS. You write a pattern, and Python's re module searches the text for anything that matches it. Regex is widely used for input validation (email, phone, PIN code), searching log files, and extracting structured data from messy text.

Real-World Scenario: Stencil Over Text

A regex is like a STENCIL you place over text — only the shapes that fit the cutouts show through.

➜ Pattern \d{10} = stencil for any 10-digit number

➜ Apply to "Call me at 9876543210 or 8888888888"

➜ Result: both phone numbers slip through, words don't

Importing

```
import re
```

Common Metacharacters

Symbol	Meaning	Example Match
.	Any single char (except newline)	a.c → abc, axc
^	Start of string	^Hi → 'Hi there'
$	End of string	end$ → 'the end'
*	0 or more of previous	ab* → a, ab, abbb
+	1 or more of previous	ab+ → ab, abbb
?	0 or 1 of previous	colou?r → color, colour
\d	Any digit (0-9)	\d\d → 25, 99
\D	Any NON-digit	—
\w	Word char (a-z, 0-9, _)	\w+ → hello, abc1
\W	Non-word char	—
\s	Whitespace (space, tab)	—
\S	Non-whitespace	—
[abc]	Any one of a, b, c	[aeiou] → vowel
[^abc]	NOT a, b, c	[^0-9] → non-digit
{n,m}	Between n and m repeats	\d{2,4} → 25, 2026
( )	Grouping	(ab)+ → ab, abab
|	OR	yes|no → yes or no

Tip

Symbol	Meaning	Example Match
Always use a RAW STRING for regex patterns: r"\d+". Without the r, \d would be interpreted as an escape sequence by Python before regex even sees it.		

10.2 Pattern Matching

Main re Functions

Function	Purpose
`re.match()`	Check pattern only at the START of string
`re.search()`	Find FIRST match anywhere in string
`re.findall()`	Return a LIST of all matches
`re.finditer()`	Return iterator of match objects
`re.sub(pat, rep, s)`	Replace all matches with rep
`re.split(pat, s)`	Split string by pattern
`re.compile(pat)`	Pre-compile pattern (faster reuse)

Match vs Search vs Findall Diagram

```
Text:  "py is fun, py is fast"
Pattern: "py"

re.match("py", text)    ──▶  match (starts with "py")
                             returns Match object: 'py' at pos 0

re.search("py", text)   ──▶  finds FIRST "py" anywhere
                             returns Match object: 'py' at pos 0

re.findall("py", text)  ──▶  finds ALL matches
                             returns ['py', 'py']
```

Example 1: Basic Search

```
import re

text = "Today is 13 May 2026"

m = re.search(r"\d+", text)        # first digit run
print(m.group())                   # 13
print(m.start(), m.end())          # 9 11
```

Example 2: Find All Digits

```
import re

text = "Order 25 placed on 2026-05-13 worth ₹4500"
print(re.findall(r"\d+", text))
# ['25', '2026', '05', '13', '4500']
```

Example 3: Replace Pattern (sub)

```
import re

text = "My phone is 9876543210, alt 8888888888"
hidden = re.sub(r"\d{10}", "XXXXXXXXXX", text)
print(hidden)
# My phone is XXXXXXXXXX, alt XXXXXXXXXX
```

Example 4: Split by Pattern

```
import re

text = "apple, banana ; mango|grape"
parts = re.split(r"[,;|]\s*", text)
print(parts)
# ['apple', 'banana', 'mango', 'grape']
```

Example 5: Email Validation

```
import re

pattern = r"^[\w.+-]+@[\w-]+\.[\w.-]+$"
emails = ["deepak@rvscet.in", "bad@@x", "asha_2025@gmail.com"]

for e in emails:
    if re.match(pattern, e):
        print(e, "→ valid")
    else:
        print(e, "→ invalid")

# deepak@rvscet.in → valid
# bad@@x → invalid
# asha_2025@gmail.com → valid
```

Example 6: Indian Mobile Number Validation

```
import re

# Must start with 6, 7, 8 or 9 and have 10 digits total
```

```
pattern = r"^[6-9]\d{9}$"

for num in ["9876543210", "5876543210", "98765-43210"]:
    if re.match(pattern, num):
        print(num, "✓ valid")
    else:
        print(num, "✗ invalid")
```

Example 7: PIN Code (Indian) Validation

```
import re

pattern = r"^[1-9]\d{5}$"        # 6 digits, can't start with 0

for pin in ["110001", "012345", "831001"]:
    if re.match(pattern, pin):
        print(pin, "✓ valid PIN")
    else:
        print(pin, "✗ invalid PIN")
```

Example 8: Extract All URLs

```
import re

text = "Visit https://rvscet.in or http://example.com for info"
urls = re.findall(r"https?://\S+", text)
print(urls)
# ['https://rvscet.in', 'http://example.com']
```

Example 9: Capturing Groups

```
import re

text = "Deepak Tiwary, age 25"
m = re.search(r"(\w+)\s(\w+),\sage\s(\d+)", text)

if m:
    print("First name:", m.group(1))   # Deepak
    print("Last name:",  m.group(2))   # Tiwary
    print("Age:",        m.group(3))   # 25
```

Regex Flow Diagram

```
    ┌──────────────────────────────────┐
    │ Tries to match 2 to 4 digits    │
    └──────────────┬───────────────────┘
                   ▼
Text: "Order 25 on 2026-05-13"
           ▲▲      ▲▲▲▲ ▲▲ ▲▲
           │       │    │  │
           '25'   '2026'  '13'
                     '05'
           All matches collected as a list
```

10.3 String Searching and Manipulation

Python also has many built-in string methods that work WITHOUT regex. Use them for simple tasks; reach for regex only when patterns are complex.

Common String Methods

Method	Purpose	Example
.strip()	Remove leading/trailing whitespace	`'  hi  '.strip() → 'hi'`
.lower()	Lowercase all	`'HI'.lower() → 'hi'`
.upper()	Uppercase all	`'hi'.upper() → 'HI'`
.replace(a, b)	Replace substring	`'hi'.replace('h','H')`
.split(sep)	Split into list	`'a,b,c'.split(',')`
.join(list)	Join list with sep	`','.join(['a','b'])`
.find(s)	Index of substring (-1 if no)	`'hello'.find('l') → 2`
.count(s)	Count occurrences	`'hello'.count('l') → 2`
.startswith(s)	Check prefix	`'hello'.startswith('he')`
.endswith(s)	Check suffix	`'hi.py'.endswith('.py')`

Example 1: Clean User Input

```
raw = "   Deepak Kumar Tiwary   "
clean = raw.strip().title()
print(clean)         # 'Deepak Kumar Tiwary'
```

Example 2: Count Vowels — Two Ways

❌ Bad Code (Avoid)	✅ Good Code (Preferred)

`# Manual loop`
`s = "Education"`
`count = 0`
`for ch in s.lower():`
	`# Regex one-liner`
	`import re`
	`print(len(re.findall(r"[aeiou]",`
	`"Education", re.I)))`

❌ Bad Code (Avoid)	✅ Good Code (Preferred)

`if ch in "aeiou":`
`count += 1`
`print(count)`	

Example 3: Word Frequency

```
text = "python is fun and python is easy"
words = text.split()
freq = {}
for w in words:
    freq[w] = freq.get(w, 0) + 1
print(freq)
# {'python': 2, 'is': 2, 'fun': 1, 'and': 1, 'easy': 1}
```

Example 4: Reverse a Sentence Word-by-Word

```
s = "Python is awesome"
rev = " ".join(s.split()[::-1])
print(rev)            # 'awesome is Python'
```

Example 5: Format Phone Number with Regex

```
import re

phone = "9876543210"
formatted = re.sub(r"(\d{5})(\d{5})", r"\1-\2", phone)
print(formatted)      # 98765-43210
```

Week 10 Cheat Sheet

- Always import re. Use raw strings r"..." for patterns.
- \d = digit, \w = word char, \s = whitespace; +, *, ? for repeats.
- re.search → first match; re.findall → all matches; re.sub → replace.
- Use ^ for start and $ for end of string.
- Groups: (pattern) — access via m.group(1), m.group(2), ...
- Prefer simple string methods (.strip, .replace, .split) for plain text.

Week 10 Review Questions

57. Define regular expression. List any five metacharacters with meaning.
58. Differentiate between re.match() and re.search().
59. Write a regex to validate a 10-digit Indian mobile number starting with 6-9.
60. Write a regex to extract all email addresses from a paragraph.
61. Replace all multiple spaces in a string with a single space using regex.

62. Write a program to count vowels in a string using regex.

Week 11: Python Libraries for Data Handling

Course Outcome	Program Outcomes	Topics Covered
CO5	PO1, PO2, PO3, PO5, PO10, PO12	Introduction to NumPy, Array operations, Introduction to Pandas

11.1 Introduction to NumPy

Definition

NumPy (Numerical Python) is the foundational library for numerical computing in Python. It introduces a powerful object called the ndarray (n-dimensional array) that can store large amounts of numeric data and perform fast mathematical operations on ALL elements at once. This style of operating on whole arrays is called VECTORIZATION and is dramatically faster than Python loops.

Real-World Scenario: Truck vs Many Small Bags

A Python list of 1 million numbers is like a thousand small bags

scattered around — you handle them one at a time, slowly.

A NumPy array is like a single neatly-stacked container truck —

you can move and process ALL items in one go.

➜ list operations = on foot

➜ NumPy operations = on a highway (10–100× faster)

Installation & Import

```
# Install once
pip install numpy

# Import (standard alias is np)
import numpy as np
```

Example 1: Creating Arrays

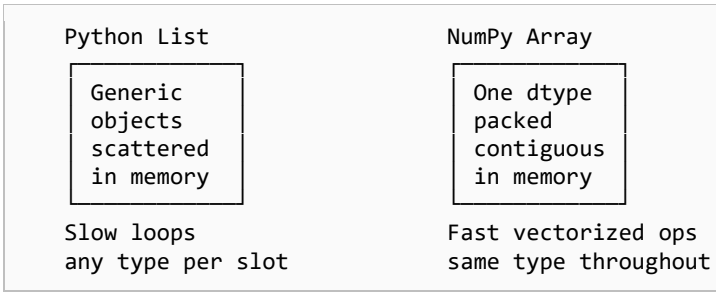
```
import numpy as np

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

```
b = np.array([[1, 2], [3, 4]])          # 2-D matrix
zeros = np.zeros((2, 3))                # 2x3 of zeros
ones  = np.ones((3, 3))                 # 3x3 of ones
rng   = np.arange(1, 10, 2)             # 1,3,5,7,9
lin   = np.linspace(0, 1, 5)            # 0, 0.25, 0.5, 0.75, 1.0

print(a.shape, a.dtype)                 # (4,) int64
print(b.shape)                          # (2, 2)
```

Example 2: Array Properties

```
a = np.array([[1, 2, 3], [4, 5, 6]])
print(a.shape)        # (2, 3)  → rows, cols
print(a.size)         # 6       → total elements
print(a.ndim)         # 2       → dimensions
print(a.dtype)        # int64   → data type
```

Array vs List (Diagram)



11.2 Array Operations

Example 1: Element-wise Math (Vectorization)

```
a = np.array([1, 2, 3, 4])
b = np.array([10, 20, 30, 40])

print(a + b)        # [11 22 33 44]
print(a * 2)        # [ 2  4  6  8]
print(b - a)        # [ 9 18 27 36]
print(a ** 2)       # [ 1  4  9 16]
```

Example 2: Indexing and Slicing 2-D Array

```
m = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]])

print(m[0, 2])          # 3      (row 0, col 2)
```

```
print(m[:, 1])          # [2 5 8]    (whole column 1)
print(m[1, :])          # [4 5 6]    (whole row 1)
print(m[0:2, 1:3])      # [[2 3]
                        #  [5 6]]
```

Example 3: Aggregation Functions

```
x = np.array([5, 9, 1, 7, 3])
print(x.sum())          # 25
print(x.mean())         # 5.0
print(x.min(), x.max()) # 1 9
print(np.median(x))     # 5.0
print(x.std())          # 2.828... (std deviation)
```

Example 4: Reshape

```
v = np.arange(1, 13)
print(v.reshape(3, 4))
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
```

Example 5: Boolean Masking (Filtering)

```
marks = np.array([78, 92, 45, 88, 30, 67])
print(marks[marks >= 60])       # [78 92 88 67]
print(marks[marks < 60])        # [45 30]
```

Example 6: Matrix Multiplication

```
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])
print(A @ B)
# [[19 22]
#  [43 50]]
```

❌ Bad Code (Avoid)	✅ Good Code (Preferred)

`# Pure Python: slow loops`
`a = [1,2,3,4,5]`
`b = [10,20,30,40,50]`
`c = []`
`for i in range(len(a)):`
`  c.append(a[i] + b[i])`
	`# NumPy: vectorized, fast`
	`import numpy as np`
	`a = np.array([1,2,3,4,5])`
	`b = np.array([10,20,30,40,50])`
	`c = a + b`

11.3 Introduction to Pandas

Definition

Pandas is built on top of NumPy and gives Python two powerful data structures: SERIES (1-D labelled data, like one column) and DATAFRAME (2-D table, like a spreadsheet). It is the go-to library for handling tabular / real-world data — student records, sales reports, marks, weather, etc.

Real-World Scenario: Excel Inside Python

Think of Pandas as Excel that you can control with code.

➜ Series = one column of Excel

➜ DataFrame = whole sheet

Read CSV → analyse → filter → chart → save back. The same operations that take 20 clicks in Excel become 2 lines in Pandas.

Installation & Import

```
pip install pandas
import pandas as pd
```

Example 1: Creating a Series

```
s = pd.Series([88, 92, 75, 80],
              index=["Deepak","Asha","Ravi","Sita"])
print(s)
# Deepak    88
# Asha      92
# Ravi      75
# Sita      80

print(s["Asha"])         # 92
print(s.mean())          # 83.75
```

Example 2: Creating a DataFrame

```
data = {
    "Name":   ["Deepak","Asha","Ravi","Sita"],
    "Course": ["BCA","BCA","BCA","BCA"],
    "Marks":  [88, 92, 75, 80],
}

df = pd.DataFrame(data)
print(df)
#     Name Course  Marks
# 0  Deepak    BCA     88
# 1    Asha    BCA     92
# 2    Ravi    BCA     75
# 3    Sita    BCA     80
```

Example 3: Common DataFrame Operations

```
print(df.head())                  # first 5 rows
print(df.shape)                   # (rows, cols)
print(df.columns)                 # column names
print(df["Marks"].mean())         # avg = 83.75
print(df["Marks"].max())          # 92
print(df[df["Marks"] > 80])       # filter
df["Grade"] = ["A","A","B","B"]   # new column
```

Example 4: Read and Write CSV

```
# Read CSV
df = pd.read_csv("students.csv")
print(df)

# Modify
df["Pass"] = df["Marks"] >= 35

# Write back
df.to_csv("students_with_status.csv", index=False)
```

Example 5: Sorting and Grouping

```
df_sorted = df.sort_values("Marks", ascending=False)
print(df_sorted)

# Group by course and get average marks
print(df.groupby("Course")["Marks"].mean())
```

Example 6: Mini Data Analysis

```
import pandas as pd

data = {
    "Student": ["A","B","C","D","E","F"],
    "Maths":   [90, 60, 70, 85, 55, 95],
    "Science": [80, 65, 75, 88, 50, 92],
}
df = pd.DataFrame(data)
df["Total"]   = df["Maths"] + df["Science"]
df["Average"] = df["Total"] / 2

print(df)
print("\nTopper:", df.loc[df["Total"].idxmax(), "Student"])
print("Class avg:", df["Average"].mean())
```

Note

Pandas can read/write CSV, Excel (.xlsx), JSON, and SQL. The line df = pd.read_csv("file.csv") is often the very first line of a real data project.

Week 11 Cheat Sheet

- NumPy ndarray: fast, vectorized, single-dtype arrays.
- Common creators: np.array, zeros, ones, arange, linspace.
- Vectorized ops: a + b, a * 2 work on whole arrays.
- Boolean masking: a[a > 50] filters elements.
- Pandas Series = labelled 1-D; DataFrame = labelled 2-D table.
- df["col"].mean(), df[df["col"] > x], pd.read_csv, df.to_csv.

Week 11 Review Questions

63. Write any three differences between a Python list and a NumPy array.
64. Create a 3×3 NumPy array of numbers 1–9 and print its mean, max and transpose.
65. What is vectorization? Why is it faster than a Python loop?
66. Differentiate between Series and DataFrame in Pandas.
67. Read a CSV file into a DataFrame, add a new column "Result" ("Pass" if marks ≥ 35 else "Fail") and save it back.
68. Write a program to find the topper from a DataFrame of 5 students.

Week 12: Data Visualization and Applications

Course Outcome	Program Outcomes	Topics Covered
CO5	PO1, PO2, PO3, PO4, PO5, PO9, PO10, PO12	Introduction to Matplotlib, Plotting graphs, Simple data analysis examples

12.1 Introduction to Matplotlib

Definition

Matplotlib is the most popular Python library for creating charts and graphs. Its pyplot module gives a simple, MATLAB-like interface to draw line plots, bar charts, histograms, scatter plots, pie charts and more. Visualization helps us understand data quickly — patterns and outliers that are invisible in a table become obvious in a chart.

Real-World Scenario: Numbers vs Picture

You are given 365 daily temperatures of Jamshedpur for 2025.

➜ As a TABLE: takes hours to spot the hot month

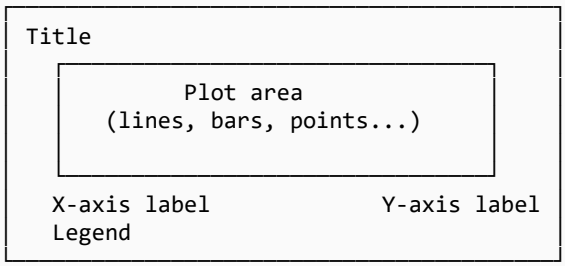
➜ As a LINE PLOT: hot summer hump is visible instantly

A good chart is worth a thousand cells of a spreadsheet.

Installation & Import

```
pip install matplotlib
import matplotlib.pyplot as plt
```

Anatomy of a Matplotlib Figure



12.2 Plotting Graphs

(a) Line Plot — Trend Over Time

Example 1: Monthly Sales Trend

```
import matplotlib.pyplot as plt

months = ["Jan","Feb","Mar","Apr","May"]
sales  = [100, 150, 200, 175, 220]

plt.plot(months, sales, marker="o", color="blue")
plt.title("Monthly Sales")
plt.xlabel("Month")
plt.ylabel("Sales (units)")
plt.grid(True)
plt.show()
```

Example 2: Multiple Lines on One Chart

```
x = list(range(1, 6))
y1 = [10, 25, 30, 45, 50]       # Product A
```

```
y2 = [15, 20, 35, 40, 60]       # Product B

plt.plot(x, y1, label="A", marker="o")
plt.plot(x, y2, label="B", marker="s")
plt.title("Sales: A vs B")
plt.xlabel("Week")
plt.ylabel("Units sold")
plt.legend()
plt.show()
```

(b) Bar Chart — Compare Categories

Example 1: Student Marks

```
students = ["Deepak","Asha","Ravi","Sita"]
marks    = [88, 92, 75, 80]

plt.bar(students, marks, color="orange")
plt.title("Student Marks")
plt.xlabel("Student")
plt.ylabel("Marks")
plt.show()
```

Example 2: Horizontal Bar Chart

```
cities = ["Delhi","Mumbai","Kolkata","Chennai","Bengaluru"]
pop    = [29, 20, 14, 11, 12]   # million

plt.barh(cities, pop, color="green")
plt.title("Indian Cities by Population (in millions)")
plt.xlabel("Population")
plt.show()
```

(c) Histogram — Distribution

Example 1: Class Marks Distribution

```
import numpy as np
data = np.random.randint(0, 100, 200)   # 200 random marks

plt.hist(data, bins=10, color="green", edgecolor="black")
plt.title("Distribution of Marks")
plt.xlabel("Marks")
plt.ylabel("Number of Students")
plt.show()
```

(d) Pie Chart — Parts of a Whole

Example 1: Study Hours per Subject

```
subjects = ["Maths","Science","English","Hindi"]
hours    = [4, 3, 2, 1]

plt.pie(hours, labels=subjects, autopct="%1.1f%%", startangle=90)
plt.title("Study Hours per Subject")
plt.show()
```

(e) Scatter Plot — Relationship Between Variables

Example 1: Study Hours vs Marks

```
hours = [1, 2, 3, 4, 5, 6, 7]
marks = [35, 50, 55, 65, 80, 85, 92]

plt.scatter(hours, marks, color="red")
plt.title("Hours of Study vs Marks Obtained")
plt.xlabel("Hours")
plt.ylabel("Marks")
plt.grid(True)
plt.show()
```

Choosing the Right Chart

Chart Type	Best Used For
Line	Trend / change over time (sales, temperature)
Bar	Comparing categories (students, products)
Histogram	Distribution of a single numeric variable
Pie	Parts of a whole (max 5–6 slices)
Scatter	Relationship between two numeric variables

12.3 Simple Data Analysis Example

Below is a small end-to-end mini project that uses NumPy, Pandas and Matplotlib together — exactly the integrated skills CO5 expects.

Problem: Analyse Marks of a Class

```
import pandas as pd
import matplotlib.pyplot as plt

# Step 1 – sample data
data = {
    "Name":  ["Deepak","Asha","Ravi","Sita","Aman","Neha"],
```

```
    "Maths": [88, 92, 75, 80, 60, 95],
    "Sci":   [82, 90, 70, 78, 65, 88],
    "Eng":   [78, 85, 72, 80, 70, 90],
}
df = pd.DataFrame(data)

# Step 2 – total & average
df["Total"]   = df[["Maths","Sci","Eng"]].sum(axis=1)
df["Average"] = df["Total"] / 3

print(df)

# Step 3 – class statistics
print("Class average:", df["Average"].mean())
print("Topper:", df.loc[df["Average"].idxmax(), "Name"])

# Step 4 – bar chart
plt.bar(df["Name"], df["Average"], color="teal")
plt.title("Average Marks per Student")
plt.xlabel("Student")
plt.ylabel("Average")
plt.show()
```

Example 1: Save Chart to File

```
plt.bar(df["Name"], df["Average"])
plt.title("Student Averages")
plt.savefig("averages.png", dpi=200, bbox_inches="tight")
plt.show()
# Creates averages.png in current folder
```

❌ Bad Code (Avoid)	✅ Good Code (Preferred)

`# Just printing – hard to compare`
`for name, avg in zip(names, averages):`
`  print(name, avg)`
	`# Visualize – instant comparison`
	`plt.bar(names, averages)`
	`plt.title("Averages")`
	`plt.show()`

Week 12 Cheat Sheet

- import matplotlib.pyplot as plt — standard alias.
- plt.plot, plt.bar, plt.hist, plt.pie, plt.scatter — main chart types.
- Always set title, xlabel, ylabel; call plt.show() at the end.
- Use Pandas to clean / aggregate data, then plot from a DataFrame.
- plt.savefig("name.png") saves the chart as image.
- Choose chart by purpose: trend → line, compare → bar, distribution → histogram.

Week 12 Review Questions

69. Name any four chart types in Matplotlib and state when each is most useful.
70. Write a Python program to plot a line graph of monthly temperatures.
71. Differentiate between a bar chart and a histogram.
72. Create a DataFrame of 5 students with three subjects, compute averages, and draw a bar chart.
73. What is the role of plt.show() and plt.savefig() in Matplotlib?
74. Plot a pie chart of daily time spent on different activities.

Week 13: Python Applications and Best Practices

Course Outcome	Program Outcomes	Topics Covered
CO5	PO1, PO2, PO3, PO4, PO5, PO9, PO10, PO12	Python coding standards, Debugging and testing basics, Mini project overview

13.1 Python Coding Standards (PEP 8)

Definition

PEP 8 is the official Python style guide. It tells developers HOW to format their code so that it is consistent and readable across the community. Following PEP 8 makes your code look professional, easier to debug, and easier for others (including your future self) to maintain.

Real-World Scenario: Traffic Rules

Imagine if every driver picked their own side of the road —
chaos. Traffic rules exist so everyone can predict each other.
PEP 8 is the same idea for Python coders: consistent style means
any developer can read any Python project quickly.

Key PEP 8 Rules

Rule	Recommendation
Indentation	4 spaces (never tabs)
Line length	≤ 79 characters
Variable/function names	snake_case (total_marks, get_data)
Class names	PascalCase (StudentRecord)
Constants	ALL_CAPS (MAX_LIMIT = 100)
Imports	One per line, at top of file
Spaces around =	x = 5 not x=5; no space inside ( )
Blank lines	2 between functions/classes, 1 inside functions
Comments	Full sentences; explain WHY, not WHAT

❌ Bad Code (Avoid)	✅ Good Code (Preferred)
`# Hard to read`	`# PEP 8 compliant`

Rule	Recommendation

def calc(a,b,c):
 x=a+b
 y=x*c
 return y
 	
result=calc(2,3,4)
print(result)
	def calculate_total(a, b, multiplier):
	  total = a + b
	  return total * multiplier
	
	result = calculate_total(2, 3, 4)
	print(result)

13.2 Debugging Basics

Definition

Debugging is the process of finding and fixing errors ("bugs") in your program. Even experienced programmers spend a large part of their time debugging — it is a normal and essential skill, not a sign of poor coding.

Common Debugging Techniques

75. Read the error message carefully — Python tells you the file, line, and type of error.
76. Use print() to inspect variable values at different points.
77. Use Python's built-in pdb (Python Debugger) for step-by-step execution.
78. In an IDE (VS Code, PyCharm), set breakpoints and use the visual debugger.
79. Reproduce the bug consistently BEFORE trying to fix it.

Example 1: print()-based Debugging

```
def avg(nums):
    print("DEBUG: nums =", nums)          # what came in?
    total = sum(nums)
    print("DEBUG: total =", total)        # is sum correct?
    return total / len(nums)

print(avg([10, 20, 30]))
```

Example 2: Using pdb (Python Debugger)

```
import pdb

def divide(a, b):
    pdb.set_trace()      # execution pauses here
    return a / b

print(divide(10, 0))

# When paused, type:
```

```
#   n  → next line
#   p a → print value of a
#   c  → continue
#   q  → quit
```

13.3 Testing Basics

Definition

Testing means writing extra code that automatically checks whether your main code behaves correctly. Tests catch bugs early and prevent old bugs from coming back when you change something later. Python has the built-in unittest module for this.

Example 1: Simple unittest Example

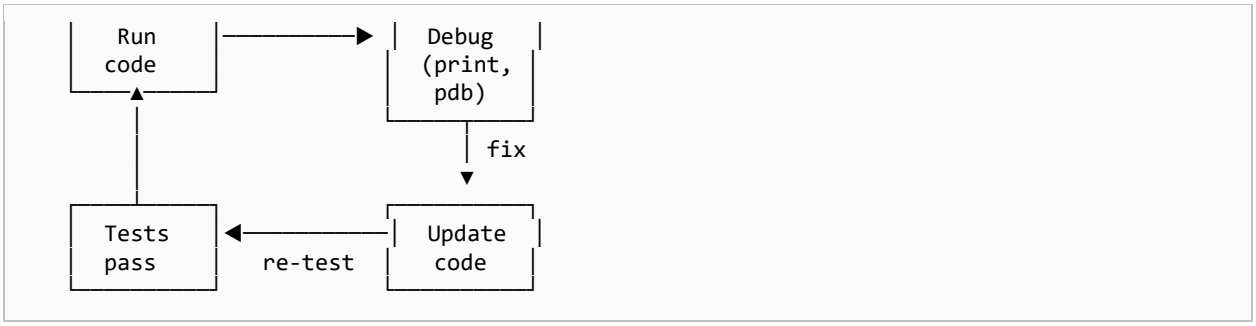
```
# file: my_math.py
def add(a, b):
    return a + b

def divide(a, b):
    if b == 0:
        raise ValueError("Cannot divide by zero")
    return a / b
```

```
# file: test_my_math.py
import unittest
from my_math import add, divide

class TestMyMath(unittest.TestCase):
    def test_add(self):
        self.assertEqual(add(2, 3), 5)
        self.assertEqual(add(-1, 1), 0)

    def test_divide(self):
        self.assertEqual(divide(10, 2), 5)
        with self.assertRaises(ValueError):
            divide(10, 0)

if __name__ == "__main__":
    unittest.main()
```

```
# Run from terminal
python -m unittest test_my_math.py
```

Debug → Fix → Test Cycle

```
   ┌──────────┐     bug?     ┌──────────┐
```



13.4 Mini Project Overview

Goal

A mini project lets you apply EVERYTHING you have learned — variables, functions, OOP, file handling, libraries — to build a small but complete application. Below is a suggested structure and a few project ideas at the BCA level.

Suggested Project Structure

```
student_record_system/
│
├── main.py                  ←  program entry point
├── student.py               ←  Student class (OOP)
├── storage.py               ←  read/write CSV (file handling)
├── analytics.py             ←  averages, topper (NumPy/Pandas)
├── plots.py                 ←  charts (Matplotlib)
├── students.csv             ←  data file
└── tests/
    └── test_student.py      ←  unittest cases
```

Mini Project Ideas (BCA Level)

80. Student Record System — add, search, update, delete students; save to CSV; plot marks chart.
81. Library Management System — book class, issue/return, fine calculation, file storage.
82. Expense Tracker — add daily expenses, monthly summary, pie chart of categories.
83. Weather Data Analyzer — read CSV of city weather, find hottest/coldest day, plot trend.
84. Quiz Application — load questions from a file, score the user, store leaderboard.

Best-Practice Checklist Before Submission

- Code follows PEP 8 (snake_case, 4-space indent, line ≤ 79).
- Functions and classes have docstrings explaining their purpose.
- No bare except: clauses; only catch what you can handle.
- All file operations use with open(...).
- Sensitive logic (calculations, validation) covered by at least one test.
- README.md explaining how to install, run, and use the project.

- Code pushed to a GitHub repository with meaningful commit messages.

> **Tip**
> Start small: build the simplest working version FIRST, then add features. A working tiny project beats a half-finished big one.

Week 13 Cheat Sheet

- PEP 8 = official Python style guide; 4-space indent, snake_case, ≤79 cols.
- Debug with print(), pdb, or IDE breakpoints — read error messages carefully.
- Test with unittest; use assertEqual, assertRaises, assertTrue.
- Run tests: python -m unittest test_file.py.
- Mini project = OOP + file handling + libraries + tests + clean code.

Week 13 Review Questions

85. What is PEP 8? List any five rules from it.
86. Explain three techniques for debugging a Python program.
87. What is unit testing? Write a small unittest case for a function add(a, b).
88. Suggest a BCA-level mini project and describe its modules briefly.
89. Differentiate between debugging and testing.