In Alignment with **NEP 2020**

PYTHON PROGRAMMING

BCA | Semester I | Academic Session 2025 – 2028

Compiled by

Deepak Kumar Tiwari

Asst. Professor, Dept. of BCA RVSCET,
Jamshedpur

Python Programming — Index

UNIT 1: INTRODUCING PYTHON

UNIT 2: PYTHON DATA TYPES

UNIT 3: PYTHON DATA TYPES

- 1. Introduction to Sets
- 2. Visual Structure of a Set
- 3. Creating a Set
- 4. Set Elements
- 5. Built-in Methods of Set
- 6. Basic Set Operations
- 7. Mathematical Set Operations
- 8. Variety of Sets
- 9. Set Example Program
- 10. Introduction to Dictionaries
- 11. Visual Structure of a Dictionary
- 12. Defining a Dictionary
- 13. Accessing Elements of a Dictionary
- 14. Basic Dictionary Operations
- 15. Dictionary Methods
- 16. Nested Dictionary
- 17. Dictionary Example Program
- 18. Difference Between Set and Dictionary

UNIT 4: COMPREHENSIONS AND FUNCTIONS

- 1. Introduction to Comprehensions
- 2. Comprehension vs Traditional Loop
- 3. List Comprehension
- 4. Set Comprehension
- 5. Dictionary Comprehension
- 6. Quick Comparison Table
- 7. Introduction to Functions
- 8. Defining a Function
- 9. Parameters vs Arguments
- 10. Types of Arguments
- 11. Unpacking Arguments
- 12. Recursive Functions
- 13. Recursion vs Iteration
- 14. Combined Example Program

UNIT 5: FUNCTIONAL PROGRAMMING

UNIT 6: MODULES, PACKAGES AND NAMESPACES

UNIT 1: INTRODUCING PYTHON

Python Programming - I | BCA 1st Semester

Prepared by: Prof. Deepak Kumar Tiwari | RVS College of Engineering and Technology, Jamshedpur

1. Introduction to Python

Python is a **high-level, general-purpose, interpreted programming language** created by **Guido van Rossum** in 1991 at the Centrum Wiskunde & Informatica (CWI), Netherlands. The name "Python" was inspired by the British comedy show *Monty Python's Flying Circus*, not the snake.

Python emphasizes **code readability** and allows programmers to express concepts in fewer lines of code compared to languages like C or Java. It is widely used in web development, data science, AI/ML, automation, scripting, and scientific computing.

Simple example:

```
print("Hello, Students of BCA!")
```

Output:

```
Hello, Students of BCA!
```

Notice: no semicolons, no main() function, no curly braces. One line and it runs.

2. Python Fundamentals

Python programs are built from a few basic building blocks:

- **Statements** — instructions the computer executes (e.g., x = 10)
- **Expressions** — combinations of values & operators that produce a result (e.g., 5 + 3)
- **Variables** — named storage locations
- **Data types** — categories of values (numbers, text, lists, etc.)
- **Functions** — reusable blocks of code
- **Indentation** — Python uses spaces/tabs (not braces) to group code

Key rule — Indentation matters:

```
if 10 > 5:
    print("Ten is greater")   # indented - part of if block
print("Always runs")          # not indented - outside the if
```

Incorrect indentation causes an **IndentationError**.

3. Features of Python

Feature	Meaning
Simple & Easy to Learn	English-like syntax, minimal rules

Interpreted	Executed line-by-line; no separate compilation step
Free & Open Source	Can be downloaded and used without cost
High-Level Language	Hides machine-level details (memory, registers)
Portable	Same code runs on Windows, Linux, macOS
Object-Oriented	Supports classes and objects
Dynamically Typed	Variable types are decided at runtime
Extensible & Embeddable	Can be mixed with C, C++, Java
Large Standard Library	Thousands of built-in modules (math, os, random, etc.)
GUI Support	Tkinter, PyQt for building graphical applications

4. Components of a Python Program

A typical Python program contains:

1. **Comments** — explanation for humans, ignored by the interpreter

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

2. **Identifiers / Variables** — names for data

```
name = "Deepak"
age = 25
```

3. **Keywords** — reserved words (e.g., if, for, while, def, import)
4. **Literals / Values** — actual data (10, "hi", 3.14, True)
5. **Operators** — +, -, *, /, ==, and, etc.
6. **Functions** — reusable blocks

```
def greet():
   print("Good Morning")
greet()
```

7. **Input / Output statements** — input() and print()

Complete small program combining all components:

```
# Program to add two numbers
a = int(input("Enter first number: "))   # input
b = int(input("Enter second number: "))
total = a + b                           # operator
print("Sum is:", total)                 # output
```

5. Understanding the Python Interpreter

An **interpreter** is a program that reads source code **line-by-line** and executes it immediately, unlike a **compiler** which translates the entire program to machine code first.

How Python runs internally:

Source Code (.py) --> Byte Code (.pyc) --> Python Virtual Machine (PVM) --> Output

- You write code in a .py file.
- Python converts it to **byte code** (intermediate form).
- The **PVM (Python Virtual Machine)** executes the byte code.

Two ways to use the interpreter:

(a) Interactive Mode — type commands one at a time at the >>> prompt:

```
>>> 2 + 3
5
>>> print("Hi")
Hi
```

(b) Script Mode — save code in a .py file and run it:

```
python hello.py
```

6. Identifiers

An identifier is a name given to variables, functions, classes, etc.

Rules for identifiers:

1. Must begin with a letter (A–Z, a–z) or underscore _
2. Followed by letters, digits (0–9), or underscores
3. **Cannot** start with a digit
4. **Cannot** be a Python keyword (if, while, class...)
5. Case-sensitive: Age, age, AGE are three different names
6. No special characters like @, $, %, space

Examples:

```
name = "Ravi"       # valid
_age = 20           # valid
roll_no = 101       # valid
2name = "X"         # INVALID (starts with digit)
my-name = "Y"       # INVALID (contains hyphen)
class = 5           # INVALID (keyword)
```

7. Basic Data Types in Python

Type	Description	Example
int	Whole numbers	10, -45, 0
float	Decimal numbers	3.14, -0.5
complex	Real + imaginary	2+3j
str	Sequence of characters	"Python", 'BCA'
bool	True or False	True, False
NoneType	Absence of value	None

Example — checking types:

```
a = 10
b = 3.14
c = "BCA"
d = True

print(type(a))    # <class 'int'>
print(type(b))    # <class 'float'>
print(type(c))    # <class 'str'>
print(type(d))    # <class 'bool'>
```

Python is **dynamically typed** — you don't declare the type:

```
x = 5        # x is int
x = "hello"   # now x is str - perfectly valid
```

8. Operators in Python

(a) Arithmetic Operators

Operator	Meaning	Example (a=10, b=3)	Result
+	Addition	a + b	13
-	Subtraction	a - b	7
*	Multiplication	a * b	30
/	Division	a / b	3.333...
//	Floor division	a // b	3
%	Modulus (remainder)	a % b	1
**	Exponent (power)	a ** b	1000

(b) Relational (Comparison) Operators — return True/False

Operator	Meaning	Example
==	Equal to	5 == 5 -> True

!=	Not equal	5 != 3 -> True
>	Greater than	7 > 2 -> True
<	Less than	2 < 7 -> True
>=	Greater or equal	5 >= 5 -> True
<=	Less or equal	4 <= 6 -> True

(c) Logical Operators

Operator	Meaning	Example
and	True if both true	True and False -> False
or	True if any true	True or False -> True
not	Reverses	not True -> False

(d) Assignment Operators

```
x = 10
x += 5    # x = x + 5  -> 15
x -= 2    # x = x - 2  -> 13
x *= 2    # -> 26
x /= 2    # -> 13.0
x %= 5    # -> 3.0
x **= 2   # -> 9.0
```

(e) Bitwise Operators (work on binary)

& AND, | OR, ^ XOR, ~ NOT, << left shift, >> right shift.

```
print(5 & 3)   # 1
print(5 | 3)   # 7
```

(f) Membership Operators

```
print('a' in 'apple')      # True
print(5 not in [1,2,3])    # True
```

(g) Identity Operators

```
a = [1,2]
b = a
print(a is b)      # True
print(a is not b)  # False
```

9. Precedence and Associativity

When an expression has multiple operators, **precedence** decides which is evaluated first. **Associativity** decides the direction (left-to-right or right-to-left) when operators have the same precedence.

Order	Operators
1	( ) parentheses
2	** exponent (right-to-left)
3	+x, -x, ~x unary
4	*, /, //, %
5	+, -
6	<, <=, >, >=
7	==, !=
8	not
9	and
10	or
11	= assignment

Example:

```
result = 10 + 5 * 2
print(result)   # 20 (not 30), because * is done before +
```

With parentheses:

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

Associativity example (right-to-left for **):

```
print(2 ** 3 ** 2)   # 2 ** (3**2) = 2 ** 9 = 512
```

10. Decision Control Structures

Used to make decisions in a program. Python supports if, if-else, if-elif-else, and nested if.

(a) Simple if

```
age = 20
if age >= 18:
    print("You can vote")
```

(b) if-else

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

```
if num % 2 == 0:
    print("Even number")
else:
    print("Odd number")
```

(c) if-elif-else (ladder)

```
marks = int(input("Enter marks: "))
if marks >= 90:
    print("Grade A")
elif marks >= 75:
    print("Grade B")
elif marks >= 60:
    print("Grade C")
elif marks >= 40:
    print("Grade D")
else:
    print("Fail")
```

(d) Nested if

```
n = int(input("Enter a number: "))
if n >= 0:
    if n == 0:
        print("Zero")
    else:
        print("Positive")
else:
    print("Negative")
```

11. Looping Structures

Loops execute a block of code repeatedly.

(a) while loop — runs as long as a condition is true

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

Output: Count: 1, 2, 3, 4, 5

(b) for loop — iterates over a sequence

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

range(1, 6) generates 1, 2, 3, 4, 5.

Looping through a string:

```
for ch in "BCA":
    print(ch)
```

Output: B, C, A

Looping through a list:

```
subjects = ["Python", "DSA", "C"]
for s in subjects:
    print(s)
```

(c) Loop Control Statements

break — exits the loop immediately

```
for i in range(1, 10):
    if i == 5:
        break
    print(i)     # prints 1,2,3,4
```

continue — skips current iteration

```
for i in range(1, 6):
    if i == 3:
        continue
    print(i)     # prints 1,2,4,5
```

pass — does nothing, acts as a placeholder

```
for i in range(3):
    pass   # will be implemented later
```

(d) Nested Loops — loop inside a loop (used for patterns, tables)

```
# Print a right-angle star triangle
for i in range(1, 5):
    for j in range(i):
        print("*", end="")
    print()
```

Output:

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

12. Console Input and Output

(a) Output using print()

```
print("Hello")
print("Sum =", 5+3)
print("Python", "is", "fun", sep="-")     # Python-is-fun
print("Line1", end=" ")
print("Line2")                           # Line1 Line2 (same line)
```

(b) Input using input()

input() always returns a **string**. Convert using int(), float() when needed.

```
name = input("Enter your name: ")
age = int(input("Enter your age: "))
height = float(input("Enter height in meters: "))
print("Name:", name, "Age:", age, "Height:", height)
```

(c) Formatted Output

Three common ways:

1. Using f-string (recommended, Python 3.6+):

```
name = "Deepak"
marks = 95
print(f"{name} scored {marks} marks")
```

2. Using .format():

```
print("{} scored {} marks".format(name, marks))
```

3. Using % operator (old style):

```
print("%s scored %d marks" % (name, marks))
```

Sample Complete Program Combining Unit 1 Concepts

```
# Program: Simple Student Grade Calculator
print("===== Student Grade Calculator =====")

name = input("Enter student name: ")
marks = float(input("Enter total marks (out of 100): "))

if marks >= 90:
    grade = "A+"
elif marks >= 75:
    grade = "A"
elif marks >= 60:
```

```
    grade = "B"
elif marks >= 40:
    grade = "C"
else:
    grade = "Fail"

print(f"\nStudent Name : {name}")
print(f"Marks        : {marks}")
print(f"Grade        : {grade}")

# Printing marks table using loop
print("\nMark breakdown (sample):")
for i in range(1, 4):
    print(f"Subject {i}: {marks/3:.2f}")
```

Quick Revision — Important Points

1. Python was created by Guido van Rossum in 1991.

2. Python is interpreted, high-level, dynamically typed, and object-oriented.

3. Indentation replaces { } in Python.

4. Common data types: int, float, str, bool, complex.

5. Identifiers cannot start with a digit or use keywords.

6. Operator precedence: ( ) > ** > *, /, % > +, - > comparison > logical.

7. Decision making: if, if-else, if-elif-else, nested if.

8. Loops: while, for, with break, continue, pass.

9. Input via input() (always string); output via print().

10. Use f-strings for clean formatted output.

Practice Questions (Short Answer)

Q1. Who developed Python and in which year?

Q2. List any five features of Python.

Q3. Differentiate between / and // operators with example.

Q4. What is the difference between break and continue?

Q5. Why is indentation important in Python?

Q6. Explain the difference between interactive mode and script mode.

Q7. Write a Python program to check if a number is positive, negative, or zero.

Q8. Write a program to print the multiplication table of a given number using for loop.

UNIT 2: PYTHON DATA TYPES

LISTS AND TUPLES | Python Programming - I | BCA 1st Semester

PART A : LISTS

1. Introduction to Lists

A **List** in Python is a **collection of items** stored in a single variable. It is one of the most commonly used data types in Python.

Key characteristics of a list:

- **Ordered** — items have a defined position (index)
- **Mutable** — items can be changed after creation
- **Allows duplicates** — same value can appear multiple times
- **Heterogeneous** — can store items of different data types
- **Written in square brackets** [ ]

Example:

```
marks = [85, 90, 78, 92, 88]
fruits = ["apple", "banana", "mango"]
mixed = [1, "Python", 3.14, True]
print(marks)
print(fruits)
print(mixed)
```

Output:

```
[85, 90, 78, 92, 88]
['apple', 'banana', 'mango']
[1, 'Python', 3.14, True]
```

2. Visual Structure of a List

Think of a list like a row of numbered boxes. Each box has two addresses — a **positive index** (from the left) and a **negative index** (from the right).

```
List:    [ 10 ,  20 ,  30 ,  40 ,  50 ]
Index+:    0    1    2    3    4
Index-:   -5   -4   -3   -2   -1
```

- list[0] -> first element (10)
- list[-1] -> last element (50)
- list[2] -> 30

3. Creating a List

```
# Empty list
empty = []

# List of integers
numbers = [10, 20, 30, 40]

# List of strings
```

```
students = ["Ravi", "Amit", "Priya"]

# Mixed list
info = ["Deepak", 25, 5.9, True]

# Nested list (list inside a list)
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

# Using list() constructor
chars = list("BCA")       # ['B', 'C', 'A']
nums = list(range(1, 6))  # [1, 2, 3, 4, 5]
```

4. Accessing Elements of a List

(a) Indexing

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

print(fruits[0])    # apple  (first)
print(fruits[2])    # mango
print(fruits[-1])   # grape  (last)
print(fruits[-2])   # orange
```

(b) Slicing — extracting a portion of a list

Syntax: list[start : stop : step]

- **start** -> index to begin (inclusive)
- **stop** -> index to end (exclusive)
- **step** -> how many items to jump

```
nums = [10, 20, 30, 40, 50, 60, 70]

print(nums[1:4])    # [20, 30, 40]      - index 1 to 3
print(nums[:3])     # [10, 20, 30]      - from start to index 2
print(nums[3:])     # [40, 50, 60, 70]  - from index 3 to end
print(nums[::2])    # [10, 30, 50, 70]  - every 2nd element
print(nums[::-1])   # [70, 60, ..., 10] - reverses the list
```

(c) Accessing Nested List Elements

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
print(matrix[0])      # [1, 2, 3]
print(matrix[1][2])   # 6  - row 1, column 2
```

5. Basic List Operations

(a) Concatenation (+) — joining two lists

```
a = [1, 2, 3]
b = [4, 5, 6]
print(a + b)       # [1, 2, 3, 4, 5, 6]
```

(b) Repetition (*)

```
print([0] * 5)      # [0, 0, 0, 0, 0]
print(["hi"] * 3)   # ['hi', 'hi', 'hi']
```

(c) Membership (in, not in)

```
fruits = ["apple", "banana", "mango"]
print("apple" in fruits)      # True
print("grape" not in fruits)  # True
```

(d) Length (len())

```
print(len([10, 20, 30, 40]))  # 4
```

(e) Iteration (looping)

```
subjects = ["Python", "DSA", "C"]
for s in subjects:
    print(s)
```

(f) Modifying elements (Mutability)

```
marks = [80, 75, 90]
marks[1] = 85         # change item at index 1
print(marks)          # [80, 85, 90]
```

(g) Deleting elements

```
nums = [10, 20, 30, 40]
del nums[1]           # deletes 20
print(nums)           # [10, 30, 40]

del nums              # deletes entire list
```

6. Built-in Methods of List

Method	Description	Example
append(x)	Adds x at the end	lst.append(50)
insert(i, x)	Inserts x at index i	lst.insert(1, 99)
extend(iter)	Appends all items of another iterable	lst.extend([4,5])
remove(x)	Removes first occurrence of x	lst.remove(20)
pop(i)	Removes & returns item at index i (default last)	lst.pop()
clear()	Removes all items	lst.clear()
index(x)	Returns index of first occurrence of x	lst.index(30)
count(x)	Counts occurrences of x	lst.count(10)
sort()	Sorts list in ascending order	lst.sort()
sort(reverse=True)	Sorts in descending order	lst.sort(reverse=True)
reverse()	Reverses the list	lst.reverse()
copy()	Returns a shallow copy	lst.copy()

Examples of each method:

```
lst = [10, 20, 30]

# append
lst.append(40)        # [10, 20, 30, 40]

# insert
lst.insert(1, 15)     # [10, 15, 20, 30, 40]

# extend
lst.extend([50, 60])  # [10, 15, 20, 30, 40, 50, 60]

# remove
lst.remove(15)        # [10, 20, 30, 40, 50, 60]

# pop
x = lst.pop()         # x = 60, lst = [10, 20, 30, 40, 50]
y = lst.pop(0)        # y = 10, lst = [20, 30, 40, 50]

# index
print(lst.index(30))  # 1

# count
nums = [1, 2, 2, 3, 2]
print(nums.count(2))  # 3

# sort
nums.sort()           # [1, 2, 2, 2, 3]
nums.sort(reverse=True)# [3, 2, 2, 2, 1]

# reverse
```

```
nums.reverse()        # reverses order

# copy
new = lst.copy()

# clear
lst.clear()           # []
```

7. Built-in Functions Useful with Lists

```
nums = [10, 25, 5, 40, 15]

print(len(nums))     # 5       - number of items
print(max(nums))     # 40      - largest
print(min(nums))     # 5       - smallest
print(sum(nums))     # 95      - total
print(sorted(nums))  # [5,10,15,25,40]  (does not modify original)
```

8. List Example Program

```
# Program: Manage a list of student marks
marks = []
n = int(input("How many students? "))

for i in range(n):
    m = int(input(f"Enter marks of student {i+1}: "))
    marks.append(m)

print("\nAll Marks :", marks)
print("Highest   :", max(marks))
print("Lowest    :", min(marks))
print("Average   :", sum(marks)/len(marks))
```

PART B : TUPLES

9. Introduction to Tuples

A **Tuple** is very similar to a list, but with one key difference — a tuple is **immutable** (cannot be changed after creation).

Key characteristics:

- **Ordered** — items have positions (index)
- **Immutable** — cannot add, remove, or change items
- **Allows duplicates**
- **Heterogeneous** — items can be of different types
- **Written in parentheses ( )**

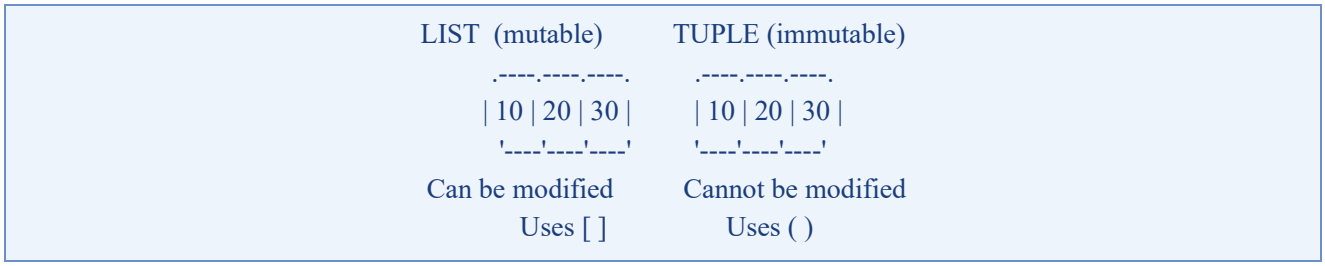
Why use tuples?

- Faster than lists (due to immutability)
- Safer — protects data from accidental modification
- Can be used as dictionary keys (lists cannot)
- Used to store fixed collections like coordinates, RGB values, days of week

Example:

```
point = (10, 20)
colors = ("Red", "Green", "Blue")
mixed = ("Deepak", 25, 5.9, True)
```

10. Visualising Lists vs Tuples



11. Creating a Tuple

```
# Empty tuple
t1 = ()

# Tuple with one element - MUST include a comma
t2 = (5,)          # correct
t3 = (5)           # WRONG - this is just an integer

# Multiple elements
t4 = (1, 2, 3)

# Without parentheses (tuple packing)
t5 = 10, 20, 30

# Using tuple() constructor
t6 = tuple("BCA")    # ('B', 'C', 'A')
t7 = tuple([1, 2])   # (1, 2)

# Nested tuple
t8 = (1, 2, (3, 4))
```

Important: A single element tuple **must** have a trailing comma — (5) is an int, but (5,) is a tuple.

12. Working with Tuple Elements

(a) Indexing and Slicing (same as lists)

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

print(colors[0])     # red
print(colors[-1])    # black
print(colors[1:4])   # ('green', 'blue', 'yellow')
print(colors[::-1])  # reversed
```

(b) Iteration

```
for c in colors:
    print(c)
```

(c) Immutability demonstration

```
t = (1, 2, 3)
t[0] = 99    # ERROR - TypeError: tuple object does not support item assignment
```

(d) Tuple Packing and Unpacking

```
# Packing
student = ("Deepak", 25, "BCA")

# Unpacking
name, age, course = student
print(name)     # Deepak
print(age)      # 25
print(course)   # BCA
```

This is very useful for swapping variables:

```
a, b = 10, 20
a, b = b, a     # swap
print(a, b)     # 20 10
```

13. Basic Tuple Operations

(a) Concatenation (+)

```
a = (1, 2, 3)
b = (4, 5, 6)
print(a + b)    # (1, 2, 3, 4, 5, 6)
```

(b) Repetition (*)

```
print(("Hi",) * 3)    # ('Hi', 'Hi', 'Hi')
```

(c) Membership (in, not in)

```
colors = ("red", "green", "blue")
print("red" in colors)       # True
print("yellow" not in colors) # True
```

(d) Length

```
print(len((10, 20, 30, 40)))  # 4
```

(e) Iteration

```
for item in (1, 2, 3):
    print(item)
```

14. Tuple Methods

Since tuples are immutable, they only have **two methods**:

Method	Description	Example
count(x)	Returns number of times x appears	t.count(2)
index(x)	Returns index of first occurrence of x	t.index(5)

```
t = (1, 2, 3, 2, 4, 2, 5)

print(t.count(2))   # 3
print(t.index(4))   # 4
```

Built-in functions that work with tuples:

```
nums = (10, 25, 5, 40, 15)

print(len(nums))     # 5
print(max(nums))     # 40
print(min(nums))     # 5
print(sum(nums))     # 95
print(sorted(nums))  # [5, 10, 15, 25, 40] - returns a list
```

15. Types of Tuples

Tuples can be classified based on their content:

(a) Empty Tuple

```
t = ()
```

(b) Single-element (Singleton) Tuple

```
t = (10,)      # must have trailing comma
```

(c) Multiple-element Tuple

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

(d) Mixed / Heterogeneous Tuple

```
t = ("Ravi", 21, 4.5, True)
```

(e) Nested Tuple

```
t = (1, 2, (3, 4), (5, 6, 7))
print(t[2])      # (3, 4)
print(t[2][1])   # 4
```

(f) Tuple from Packing

```
t = 10, 20, 30    # parentheses optional
```

16. Tuple Example Program

```
# Program: Store and display student information using tuple
student = ("Deepak", "BCA", 2, 85.5)

name, course, sem, marks = student   # unpacking

print("Student Information")
print("--------------------")
print(f"Name   : {name}")
print(f"Course : {course}")
print(f"Sem    : {sem}")
print(f"Marks  : {marks}")
```

17. Difference Between List and Tuple

Feature	List	Tuple
Syntax	[1, 2, 3]	(1, 2, 3)

Mutability	Mutable	Immutable
Methods available	Many (append, remove, sort, ...)	Only count() and index()
Speed	Slower	Faster
Memory usage	More	Less
Use as dictionary key	Not allowed	Allowed
Use case	When data may change	When data is fixed
Iteration speed	Slower	Faster

Simple rule to remember: Use a **list** when your collection of items needs to be **changed** later. Use a **tuple** when the data should stay **fixed / safe** from changes.

18. Conversion between List and Tuple

```
# List -> Tuple
lst = [1, 2, 3]
tpl = tuple(lst)
print(tpl)      # (1, 2, 3)

# Tuple -> List
tpl = (4, 5, 6)
lst = list(tpl)
print(lst)      # [4, 5, 6]
```

This is useful when you want to temporarily modify a tuple:

```
t = (10, 20, 30)
temp = list(t)
temp.append(40)
t = tuple(temp)
print(t)        # (10, 20, 30, 40)
```

19. Combined Example Program (List + Tuple)

```
# Program: Store a list of students; each student info is a tuple

students = [
    ("Ravi", 85),
    ("Amit", 72),
    ("Priya", 91),
    ("Sneha", 68)
]

print(f"{'Name':<10}{'Marks':<10}")
print("-" * 20)
for name, marks in students:
    print(f"{name:<10}{marks:<10}")
```

```
# Find topper
topper = max(students, key=lambda x: x[1])
print(f"\nTopper: {topper[0]} with {topper[1]} marks")
```

Quick Revision — Important Points

1. List uses [ ]; Tuple uses ( ).

2. Lists are mutable; Tuples are immutable.

3. Both support indexing, slicing, concatenation, repetition, membership, iteration.

4. Negative indexing starts from -1 (last element).

5. Lists have many methods (append, insert, sort, etc.); tuples only have count() and index().

6. A single-element tuple needs a trailing comma: (5,).

7. Tuples are faster and consume less memory than lists.

8. Use list() and tuple() to convert between the two.

Practice Questions

Short Answer:

Q1. Define list and tuple with one example each.

Q2. Differentiate between list and tuple with at least four points.

Q3. What is the output of len(("Python",)) ?

Q4. Explain tuple packing and unpacking with example.

Q5. Why does a tuple have only two methods?

Programming:

Q1. Write a program to find the sum and average of elements in a list.

Q2. Write a program to find the largest and smallest number in a list without using max() / min().

Q3. Write a program to count occurrences of an element in a tuple.

Q4. Write a program to merge two tuples and sort the result.

Q5. Write a program to reverse a list using slicing.

Q6. Create a list of 5 student tuples (name, marks) and display the name of the student with the highest marks.

UNIT 3: PYTHON DATA TYPES

SETS AND DICTIONARIES | Python Programming - I | BCA 1st Semester

PART A: SETS

1. Introduction to Sets

A **Set** in Python is a **collection of unique (non-duplicate) items**. It is inspired by the mathematical concept of sets.

Key characteristics of a set:

- **Unordered** — items do not have a fixed position; no indexing or slicing
- **Unique** — duplicate values are automatically removed
- **Mutable** — you can add or remove elements from the set itself
- **Elements must be immutable** — only numbers, strings, tuples; lists/dicts are not allowed as elements
- **Written using curly braces** { } or the set() constructor

Example:

```
s1 = {10, 20, 30, 40}
s2 = {"red", "green", "blue"}
s3 = {1, 2, 2, 3, 3, 3, 4}   # duplicates removed
print(s1)
print(s2)
print(s3)
```

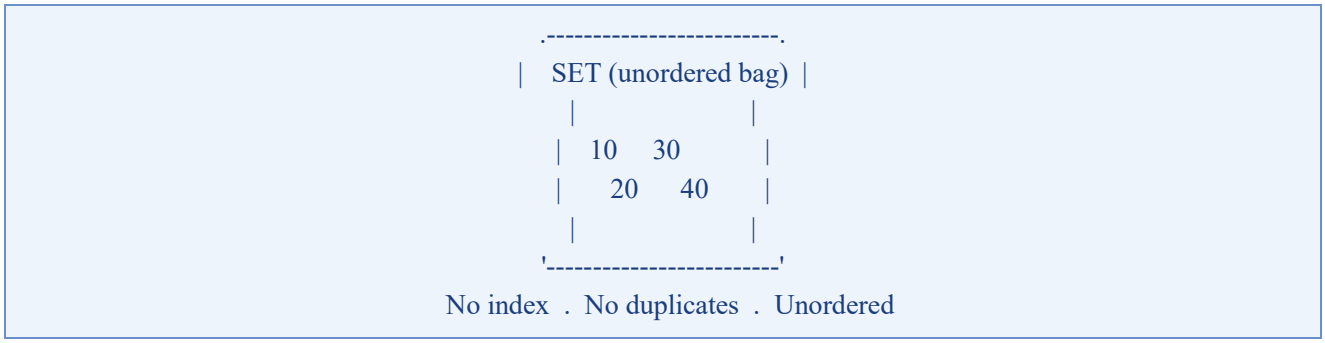
Output:

```
{40, 10, 20, 30}
{'red', 'green', 'blue'}
{1, 2, 3, 4}
```

Since sets are **unordered**, the print order may differ from the insertion order — that is normal behaviour.

2. Visual Structure of a Set

A set is like a bag of unique items — items are inside, but they have no fixed position. You cannot say "the third element" of a set.



3. Creating a Set

```
# Using curly braces
s1 = {1, 2, 3}

# Using set() constructor
s2 = set([10, 20, 30])      # from a list
s3 = set((4, 5, 6))         # from a tuple
s4 = set("BCA")             # from a string -> {'B', 'C', 'A'}
```

```
# Empty set - MUST use set(), NOT { }
empty = set()          # correct
wrong = {}             # this is an empty DICTIONARY, not a set!
```

Important: Writing { } creates an empty **dictionary**, not an empty set. To create an empty set, always use set().

4. Set Elements

Set elements must be immutable (unchangeable). Allowed: numbers, strings, tuples. Not allowed: lists, dictionaries, sets themselves.

```
# Valid elements
s = {1, "hello", (2, 3), 3.14, True}
print(s)

# Invalid - list is mutable
s = {[1, 2], 3}    # TypeError: unhashable type: 'list'
```

Sets do not support indexing:

```
s = {10, 20, 30}
print(s[0])    # TypeError: 'set' object is not subscriptable
```

But you can iterate through a set using a loop:

```
s = {10, 20, 30}
for item in s:
    print(item)
```

5. Built-in Methods of Set

Method	Description	Example
add(x)	Adds element x to the set	s.add(50)
update(iter)	Adds all items from an iterable	s.update([4, 5])
remove(x)	Removes x; error if x is not present	s.remove(20)
discard(x)	Removes x; no error if x is missing	s.discard(20)
pop()	Removes & returns a random element	s.pop()
clear()	Removes all elements	s.clear()
copy()	Returns a shallow copy of the set	s.copy()
union(t)	Returns a new set with items of both	s.union(t)
intersection(t)	Returns items common to both	s.intersection(t)
difference(t)	Items in s but not in t	s.difference(t)
symmetric_difference(t)	Items in either but not both	s.symmetric_difference(t)

issubset(t)	True if every item of s is in t	s.issubset(t)
issuperset(t)	True if s contains all items of t	s.issuperset(t)
isdisjoint(t)	True if s and t share no items	s.isdisjoint(t)

Examples of each method:

```
s = {10, 20, 30}

# add
s.add(40)              # {10, 20, 30, 40}

# update
s.update([50, 60])     # {10, 20, 30, 40, 50, 60}

# remove (error if missing)
s.remove(10)           # {20, 30, 40, 50, 60}

# discard (safe - no error)
s.discard(100)         # no change, no error

# pop (removes a random item)
x = s.pop()            # x gets some element

# copy
new = s.copy()

# clear
s.clear()              # set()
```

6. Basic Set Operations

(a) Length (len())

```
print(len({10, 20, 30}))  # 3
```

(b) Membership (in, not in)

```
s = {"apple", "banana", "mango"}
print("apple" in s)      # True
print("grape" not in s)  # True
```

(c) Iteration (looping)

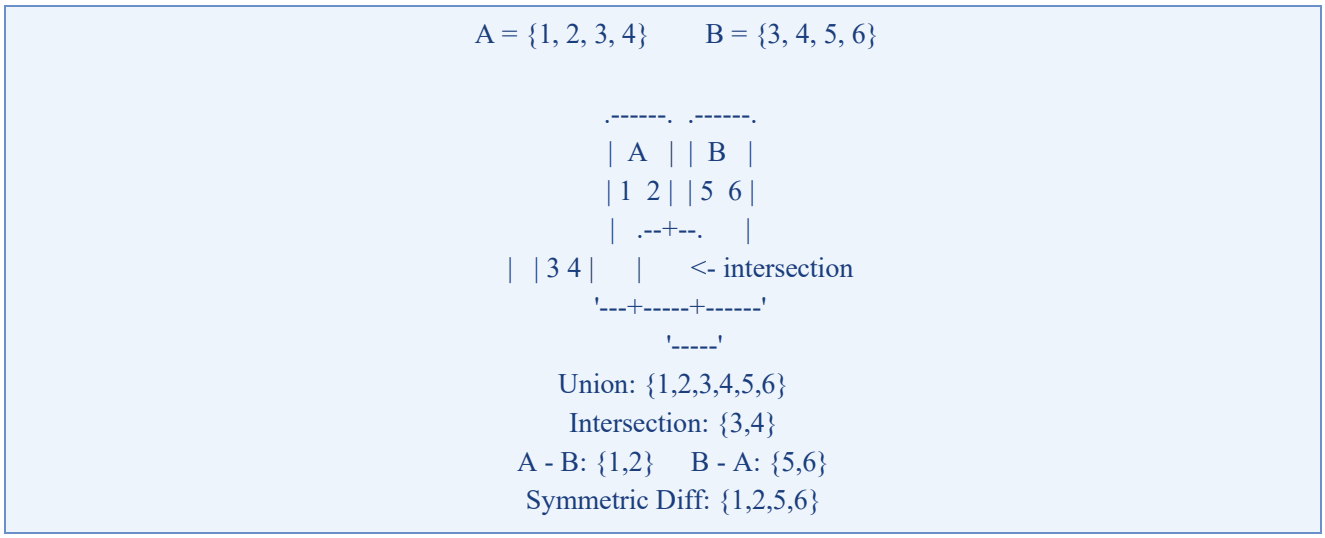
```
colors = {"red", "green", "blue"}
for c in colors:
    print(c)
```

(d) Adding and Removing elements

```
s = {1, 2, 3}
s.add(4)       # {1, 2, 3, 4}
s.remove(2)    # {1, 3, 4}
s.discard(99)  # safe - no error
```

7. Mathematical Set Operations

Python sets support the same operations as mathematical sets: Union, Intersection, Difference, and Symmetric Difference. These can be written using either the operator symbol or the method name.



(a) Union (| or union()) — all elements from both sets

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

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

(b) Intersection (& or intersection()) — common elements only

```
print(A & B)              # {3, 4}
print(A.intersection(B))  # {3, 4}
```

(c) Difference (- or difference()) — in A but not in B

```
print(A - B)              # {1, 2}
print(A.difference(B))    # {1, 2}
print(B - A)              # {5, 6}
```

(d) Symmetric Difference (^ or symmetric_difference())

Returns elements that are in either of the sets but not in both.

```
print(A ^ B)                    # {1, 2, 5, 6}
print(A.symmetric_difference(B))    # {1, 2, 5, 6}
```

(e) Subset / Superset / Disjoint tests

```
X = {1, 2}
Y = {1, 2, 3, 4}

print(X.issubset(Y))     # True  - every item of X is in Y
print(Y.issuperset(X))   # True  - Y contains all items of X
print(X.isdisjoint({8, 9}))   # True  - no common items
```

8. Variety of Sets

(a) Empty Set

```
s = set()       # { } is a dictionary, not a set
```

(b) Singleton Set (one element)

```
s = {100}
```

(c) Mixed / Heterogeneous Set

```
s = {1, "hello", (2, 3), 4.5, True}
```

(d) Set from other iterables

```
s1 = set([1, 2, 3, 2, 1])     # {1, 2, 3}
s2 = set("Mississippi")       # {'M', 'i', 's', 'p'}
s3 = set(range(1, 6))         # {1, 2, 3, 4, 5}
```

(e) Frozen Set — an immutable version of a set

A **frozenset** is like a set but cannot be modified after creation. It can be used as a dictionary key or as an element inside another set.

```
fs = frozenset([1, 2, 3])
print(fs)             # frozenset({1, 2, 3})

fs.add(4)             # ERROR - frozensets are immutable
```

9. Set Example Program

```
# Program: Find common subjects taken by two students
ravi = {"Python", "C", "DSA", "Math"}
amit = {"Python", "Java", "DSA", "DBMS"}

print("Subjects taken by Ravi :", ravi)
print("Subjects taken by Amit :", amit)

print("Common subjects       :", ravi & amit)
print("All unique subjects   :", ravi | amit)
print("Only Ravi takes       :", ravi - amit)
print("Only Amit takes       :", amit - ravi)
print("Unique to one of them :", ravi ^ amit)
```

PART B : DICTIONARIES

10. Introduction to Dictionaries

A **Dictionary** in Python is a collection of **key-value pairs**. Think of it like a real-world dictionary where each word (key) has a meaning (value).

Key characteristics:

- Stores data as **key : value** pairs
- Keys must be **unique and immutable** (strings, numbers, tuples)
- Values can be **any data type** and can be duplicated
- **Mutable** — you can add, remove, or update items
- **Ordered** (from Python 3.7+) — remembers the order items were added
- **Written using curly braces** { } with colons : between key and value

Example:

```
student = {
    "name": "Deepak",
    "age": 25,
    "course": "BCA",
    "marks": 85.5
}
print(student)
```

Output:

```
{'name': 'Deepak', 'age': 25, 'course': 'BCA', 'marks': 85.5}
```

11. Visual Structure of a Dictionary

```
                    DICTIONARY
                .----------------------------.
```

```
| KEY       :     VALUE     |
 |-------------------------------|
 | 'name'    :  'Deepak'     |
   | 'age'     :   25          |
 | 'course'  :  'BCA'        |
  | 'marks'   :   85.5        |
 '-------------------------------'
```

Keys are unique - Values can repeat

12. Defining a Dictionary

```
# Empty dictionary
d1 = {}
d2 = dict()

# Dictionary with items
marks = {"Python": 85, "C": 78, "DSA": 90}

# Using dict() constructor with keyword arguments
student = dict(name="Ravi", age=20, course="BCA")

# From a list of key-value tuples
d = dict([("a", 1), ("b", 2), ("c", 3)])

# Nested dictionary (dictionary inside dictionary)
students = {
    "s1": {"name": "Ravi", "marks": 85},
    "s2": {"name": "Amit", "marks": 72}
}
```

13. Accessing Elements of a Dictionary

(a) Using square brackets [ ]

```
student = {"name": "Deepak", "age": 25, "course": "BCA"}

print(student["name"])     # Deepak
print(student["course"])   # BCA

# If key not found -> KeyError
print(student["city"])     # KeyError: 'city'
```

(b) Using get() method (safer)

The **get()** method returns None (or a default value you provide) if the key is missing — no error is raised.

```
print(student.get("name"))           # Deepak
print(student.get("city"))           # None
print(student.get("city", "Unknown")) # Unknown  (default value)
```

(c) Accessing Nested Dictionary

```
students = {
   "s1": {"name": "Ravi", "marks": 85},
   "s2": {"name": "Amit", "marks": 72}
}

print(students["s1"])           # {'name': 'Ravi', 'marks': 85}
print(students["s1"]["name"])   # Ravi
print(students["s2"]["marks"])  # 72
```

14. Basic Dictionary Operations

(a) Adding / Updating items

```
student = {"name": "Ravi", "age": 20}

# Add new key-value pair
student["course"] = "BCA"
print(student)   # {'name': 'Ravi', 'age': 20, 'course': 'BCA'}

# Update existing value (same syntax)
student["age"] = 21
print(student)   # age is now 21
```

(b) Deleting items

```
student = {"name": "Ravi", "age": 20, "course": "BCA"}

# Using del
del student["age"]         # {'name': 'Ravi', 'course': 'BCA'}

# Using pop() - also returns the removed value
x = student.pop("course")   # x = 'BCA'

# Delete entire dictionary
del student
```

(c) Length (len())

```
print(len({"a": 1, "b": 2, "c": 3}))   # 3
```

(d) Membership (in, not in) — checks the KEY, not the value

```
d = {"name": "Deepak", "age": 25}

print("name" in d)        # True
```

```
print("Deepak" in d)      # False  (checks keys, not values)
print("city" not in d)    # True
```

(e) Iteration (looping)

```
marks = {"Python": 85, "C": 78, "DSA": 90}

# Loop over keys (default)
for k in marks:
    print(k, "->", marks[k])

# Loop over key-value pairs using items()
for k, v in marks.items():
    print(k, ":", v)
```

15. Dictionary Methods

Method	Description	Example
keys()	Returns all keys	d.keys()
values()	Returns all values	d.values()
items()	Returns all (key, value) pairs	d.items()
get(k)	Returns value of key k (None if missing)	d.get('name')
get(k, default)	Returns default if key is missing	d.get('x', 0)
update(d2)	Merges another dict into this one	d.update(d2)
pop(k)	Removes key k and returns its value	d.pop('age')
popitem()	Removes & returns last inserted (k, v)	d.popitem()
setdefault(k, v)	Returns value of k; sets to v if missing	d.setdefault('x', 0)
fromkeys(keys, v)	Creates dict with same value for all keys	dict.fromkeys(['a','b'], 0)
clear()	Removes all items	d.clear()
copy()	Returns a shallow copy	d.copy()

Examples of each method:

```
student = {"name": "Deepak", "age": 25, "course": "BCA"}

# keys, values, items
print(student.keys())     # dict_keys(['name', 'age', 'course'])
print(student.values())   # dict_values(['Deepak', 25, 'BCA'])
print(student.items())    # dict_items([('name', 'Deepak'), ...])

# get
print(student.get("age"))          # 25
print(student.get("city", "N/A"))  # N/A

# update - merges another dictionary
```

```
student.update({"age": 26, "city": "Jamshedpur"})
print(student)

# pop
age = student.pop("age")    # age = 26, key removed

# popitem - removes last inserted pair
last = student.popitem()

# setdefault - add only if missing
student.setdefault("grade", "A")  # adds only if 'grade' not present

# fromkeys - create a dict with same value for all keys
default_marks = dict.fromkeys(["Python", "C", "DSA"], 0)
print(default_marks)   # {'Python': 0, 'C': 0, 'DSA': 0}

# copy and clear
backup = student.copy()
student.clear()            # {}
```

16. Nested Dictionary

A dictionary can contain another dictionary as a value. This is useful for storing structured data like a list of students, employees, products, etc.

```
students = {
    "s1": {"name": "Ravi",  "marks": 85, "course": "BCA"},
    "s2": {"name": "Amit",  "marks": 72, "course": "BCA"},
    "s3": {"name": "Priya", "marks": 91, "course": "BCA"}
}

# Accessing inner values
print(students["s1"]["name"])     # Ravi
print(students["s3"]["marks"])    # 91

# Looping through nested dictionary
for sid, info in students.items():
    print(sid, "->", info["name"], ":", info["marks"])
```

17. Dictionary Example Program

```
# Program: Store and display student marks using a dictionary
marks = {}
n = int(input("How many subjects? "))

for i in range(n):
    sub = input("Enter subject name: ")
    m   = int(input("Enter marks: "))
    marks[sub] = m

print("\n--- Marks Report ---")
```

```
for subject, mark in marks.items():
    print(f"{subject:<10} : {mark}")

print("\nTotal   :", sum(marks.values()))
print("Average :", sum(marks.values()) / len(marks))
print("Highest :", max(marks.values()))
print("Lowest  :", min(marks.values()))
```

18. Difference Between Set and Dictionary

Feature	Set	Dictionary
Syntax	{1, 2, 3}	{'a': 1, 'b': 2}
Stores	Only values (unique)	Key-value pairs
Order	Unordered	Ordered (Python 3.7+)
Mutability	Mutable	Mutable
Duplicates	Not allowed	Duplicate values allowed; keys must be unique
Indexing	Not supported	Access via keys (d['name'])
Empty form	set()	{ } or dict()
Main use	Remove duplicates, set math	Lookup data by a key

Remember: { } creates an empty **dictionary**, not a set. Use set() to make an empty set.

Quick Revision — Important Points

1. A set stores unique, unordered, immutable elements inside { } or set().
2. { } creates an empty dictionary, not an empty set. Use set() for an empty set.
3. Set operations: union (|), intersection (&), difference (-), symmetric difference (^).
4. remove() raises error if item missing; discard() does not.
5. frozenset is an immutable version of set (used as dictionary keys).
6. Dictionaries store data as key : value pairs inside { }.
7. Dictionary keys must be unique and immutable (str, int, tuple); values can be anything.
8. Access dictionary items using d[key] or the safer d.get(key).
9. Methods: keys(), values(), items(), update(), pop(), setdefault(), fromkeys().
10. in / not in checks KEYS of a dictionary, not values.

Practice Questions

Short Answer:

Q1. Define set and dictionary with one example each.
Q2. Why is { } an empty dictionary and not an empty set?
Q3. Differentiate between remove() and discard() in sets.

Q4. What is the output of len({1, 1, 2, 2, 3}) ?

Q5. Can a list be used as a dictionary key? Why or why not?

Q6. Differentiate between set and frozenset.

Q7. Write the difference between keys(), values(), and items() methods.

Programming:

Q1. Write a program to remove duplicate elements from a list using a set.

Q2. Write a program to find union, intersection, and difference of two sets.

Q3. Write a program to count how many common subjects two students are studying.

Q4. Write a program to create a dictionary of 5 students (name : marks) and display the topper.

Q5. Write a program to count the frequency of each character in a string using a dictionary.

Q6. Write a program to merge two dictionaries into one.

Q7. Write a program to swap keys and values of a dictionary.

Q8. Write a program to store roll number (key) and name (value) of 5 students and search a name by roll number.

UNIT 4: COMPREHENSIONS AND FUNCTIONS

PART A : COMPREHENSIONS

1. Introduction to Comprehensions

A **comprehension** is a short, single-line way to create a list, set, or dictionary in Python. It replaces a multi-line for loop with one compact expression that is easier to read once you get used to it.

Three types of comprehensions:

- **List Comprehension** — builds a **list** using [ ]
- **Set Comprehension** — builds a **set** using { } (no duplicates)
- **Dictionary Comprehension** — builds a **dictionary** using {key: value}

Why use comprehensions?

- Shorter and cleaner than a traditional for loop
- Faster execution in most cases
- Easier to understand once practiced
- Looks closer to mathematical notation

2. Comprehension vs Traditional Loop

Goal: create a list of squares of numbers from 1 to 5.

Traditional way (using a for loop):

```
squares = []
for x in range(1, 6):
    squares.append(x ** 2)
print(squares)   # [1, 4, 9, 16, 25]
```

Same task using a list comprehension:

```
squares = [x ** 2 for x in range(1, 6)]
print(squares)   # [1, 4, 9, 16, 25]
```

Both produce the same result, but the comprehension does it in **one line**.

3. List Comprehension

General syntax:

```
[ expression  for  item  in  iterable  if  condition ]
```

Structure breakdown:

```
[  x*x      for  x   in  range(1,6)  if  x % 2 == 0  ]
          |      |  |       |            |
expression  loop  var     sequence      condition (optional)
```

(a) Simple list comprehension

```
# Numbers from 1 to 5
nums = [x for x in range(1, 6)]
print(nums)       # [1, 2, 3, 4, 5]

# Squares
squares = [x ** 2 for x in range(1, 6)]
print(squares)    # [1, 4, 9, 16, 25]

# Cubes
cubes = [x ** 3 for x in range(1, 5)]
print(cubes)      # [1, 8, 27, 64]
```

(b) With a condition (filter)

```
# Even numbers from 1 to 10
evens = [x for x in range(1, 11) if x % 2 == 0]
print(evens)      # [2, 4, 6, 8, 10]

# Squares of odd numbers
odd_sq = [x*x for x in range(1, 11) if x % 2 != 0]
print(odd_sq)     # [1, 9, 25, 49, 81]
```

(c) With if-else inside the expression

When you want different output based on a condition, place the if-else **before** the for.

```
# Mark each number as Even or Odd
labels = ["Even" if x % 2 == 0 else "Odd" for x in range(1, 6)]
```

```
print(labels)      # ['Odd', 'Even', 'Odd', 'Even', 'Odd']
```

(d) Working with strings

```
# Convert each letter of a string to uppercase
word = "python"
caps = [ch.upper() for ch in word]
print(caps)        # ['P', 'Y', 'T', 'H', 'O', 'N']

# Pick only vowels from a string
text = "BCA Student"
vowels = [ch for ch in text if ch.lower() in "aeiou"]
print(vowels)      # ['A', 'u', 'e']
```

(e) Nested list comprehension (matrix)

```
# 3x3 multiplication table
table = [[i*j for j in range(1, 4)] for i in range(1, 4)]
print(table)
# [[1, 2, 3], [2, 4, 6], [3, 6, 9]]

# Flatten a 2D matrix into a single list
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
flat = [num for row in matrix for num in row]
print(flat)        # [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

4. Set Comprehension

Same idea as a list comprehension, but uses { } instead of [ ] — the result is a **set** (unique items, unordered).

Syntax:

```
{ expression  for  item  in  iterable  if  condition }
```

Examples:

```
# Squares of numbers 1 to 5 as a set
squares = {x*x for x in range(1, 6)}
print(squares)        # {1, 4, 9, 16, 25}

# Duplicates are automatically removed
nums = [1, 2, 2, 3, 3, 3, 4]
unique_squares = {n*n for n in nums}
print(unique_squares) # {1, 4, 9, 16}

# Unique vowels from a sentence
sentence = "Welcome to BCA Department"
vowels = {ch.lower() for ch in sentence if ch.lower() in "aeiou"}
print(vowels)         # {'e', 'o', 'a'}
```

5. Dictionary Comprehension

Used to build a **dictionary** in one line. The expression must produce **key : value** pairs.

Syntax:

```
{ key_expr : value_expr  for  item  in  iterable  if  condition }
```

Examples:

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

# Subject and default marks (0)
subjects = ["Python", "C", "DSA"]
marks = {sub: 0 for sub in subjects}
print(marks)       # {'Python': 0, 'C': 0, 'DSA': 0}

# Combining two lists into a dictionary
names = ["Ravi", "Amit", "Priya"]
scores = [85, 72, 91]
student_marks = {n: s for n, s in zip(names, scores)}
print(student_marks) # {'Ravi': 85, 'Amit': 72, 'Priya': 91}

# With a condition - keep only passing students
result = {n: s for n, s in zip(names, scores) if s >= 75}
print(result)      # {'Ravi': 85, 'Priya': 91}

# Swap keys and values of a dictionary
d = {"a": 1, "b": 2, "c": 3}
inv = {v: k for k, v in d.items()}
print(inv)         # {1: 'a', 2: 'b', 3: 'c'}
```

6. Quick Comparison Table

Comprehension	Brackets	Result	Example
List	[ ]	List (ordered, duplicates allowed)	[x*x for x in range(5)]
Set	{ }	Set (unordered, unique)	{x*x for x in range(5)}
Dictionary	{ : }	Dict of key:value pairs	{x: x*x for x in range(5)}

PART B : FUNCTIONS

7. Introduction to Functions

A **function** is a named block of code that performs a specific task. We write it once and reuse it as many times as needed. Functions make programs **modular, reusable, and easier to debug**.

Advantages of functions:

- Avoid writing the same code again and again (code reuse)
- Break a big problem into small, manageable pieces
- Make programs easier to read and maintain
- Allow team members to work on different functions independently
- Make debugging and testing simpler

Two categories of functions in Python:

- **Built-in functions** — already provided by Python (e.g., print(), len(), input(), max(), sum())
- **User-defined functions** — written by the programmer using the def keyword

8. Defining a Function

Syntax:

```
def function_name(parameters):
    """ optional docstring """
    statement(s)
    return value     # optional
```

Structure of a function:

```
def    greet ( name )  :
        |      |    |
keyword  name  parameter

print('Hello', name)     <- function body (indented)
 return 'Done'           <- returns a value (optional)
```

Example 1: A function with no parameters

```
def greet():
    print("Good morning, students!")

# Calling the function
greet()
greet()      # call again - reusable!
```

Example 2: A function with parameters

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

greet("Deepak")    # Hello, Deepak
greet("Ravi")      # Hello, Ravi
```

Example 3: A function that returns a value

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

```
    return n * n

result = square(5)
print(result)       # 25
print(square(8))    # 64
```

Example 4: A function with multiple parameters and return

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

print(add(10, 20))   # 30
print(add(3.5, 2.5)) # 6.0
```

9. Parameters vs Arguments

Parameter	Argument
A variable listed in the function definition	The actual value passed when calling the function
Used inside the function body	Provided by the caller
Example: def add(a, b) -> a, b are parameters	Example: add(10, 20) -> 10, 20 are arguments

10. Types of Arguments

Python supports four main types of arguments:

(a) Positional Arguments

Values are matched to parameters by their position (order matters).

```
def student(name, course):
    print("Name:", name)
    print("Course:", course)

student("Deepak", "BCA")
# Name: Deepak
# Course: BCA

student("BCA", "Deepak")   # WRONG order - confusing output
```

(b) Keyword Arguments

Values are passed using parameter names, so the order does not matter.

```
def student(name, course):
    print(name, "is studying", course)

student(course="BCA", name="Deepak")
```

```
# Deepak is studying BCA
```

(c) Default Arguments

A parameter can have a default value. If the caller does not provide a value, the default is used.

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

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

Rule: Default arguments must come **after** non-default arguments in the parameter list.

(d) Variable-Length Arguments (*args and **kwargs)

Sometimes we don't know in advance how many arguments will be passed. Python provides two special syntaxes for this:

- ***args** — collects extra positional arguments as a **tuple**
- ****kwargs** — collects extra keyword arguments as a **dictionary**

Example with *args:

```
def total(*nums):
    print("Numbers received:", nums)   # tuple
    return sum(nums)

print(total(10, 20))            # 30
print(total(1, 2, 3, 4, 5))     # 15
```

Example with **kwargs:

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

show_info(name="Deepak", age=25, course="BCA")
# name : Deepak
# age : 25
# course : BCA
```

11. Unpacking Arguments

Unpacking is the **opposite** of *args / **kwargs. We can **unpack** a list/tuple/dictionary into a function's parameters using * or **.

(a) Unpacking a list/tuple with *

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

```
nums = [10, 20, 30]
print(add(*nums))      # 60   - same as add(10, 20, 30)
```

(b) Unpacking a dictionary with **

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

data = {"name": "Deepak", "course": "BCA", "marks": 85}
student(**data)       # same as student(name="Deepak", course="BCA", marks=85)
```

(c) Common pattern — forwarding arguments

```
def display(a, b, c):
    print(a, b, c)

values = (1, 2, 3)
display(*values)      # 1 2 3
```

12. Recursive Functions

A **recursive function** is a function that **calls itself** to solve a smaller version of the same problem.

Two essential parts of any recursive function:

- **Base case** — the simplest case where the function stops calling itself
- **Recursive case** — the function calls itself with a smaller/simpler input

> Without a base case, a recursive function calls itself **forever** — leading to a **RecursionError**.

Visualising Recursion (Factorial of 4)

```
factorial(4)
= 4 * factorial(3)
  = 3 * factorial(2)
    = 2 * factorial(1)
      = 1   <- base case
      = 2 * 1   = 2
    = 3 * 2   = 6
= 4 * 6   = 24
```

Example 1: Factorial using Recursion

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

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

Example 2: Sum of First N Natural Numbers

```
def total(n):
    if n == 0:
        return 0
    return n + total(n - 1)

print(total(5))    # 15  (5+4+3+2+1)
```

Example 3: Fibonacci Series using Recursion

The Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, ...

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

for i in range(7):
    print(fib(i), end=" ")    # 0 1 1 2 3 5 8
```

Example 4: Power of a number (a^b)

```
def power(a, b):
    if b == 0:
        return 1
    return a * power(a, b - 1)

print(power(2, 5))   # 32
```

13. Recursion vs Iteration

Feature	Recursion	Iteration (loop)
Definition	Function calls itself	Uses for/while loop
Termination	Base case stops it	Loop condition stops it
Code length	Shorter, more elegant	Usually longer
Memory usage	More (uses call stack)	Less
Speed	Generally slower	Generally faster
Best for	Problems split into similar sub-problems	Repeating fixed steps

14. Combined Example Program

A program that uses comprehensions and a custom function together.

```
# Program: Generate marks of 5 students and find passed students

import random

def is_pass(mark):
    """Returns True if mark >= 40"""
    return mark >= 40

# Generate 5 random marks (0-100) using a list comprehension
marks = [random.randint(0, 100) for _ in range(5)]
print("All Marks   :", marks)

# Filter passed marks using comprehension + function
passed = [m for m in marks if is_pass(m)]
print("Passed Marks:", passed)

# Build a dictionary {student_no: mark} using dict comprehension
report = {f"S{i+1}": m for i, m in enumerate(marks)}
print("Report      :", report)
```

Quick Revision — Important Points

1. A comprehension is a one-line way to build a list, set, or dictionary.

2. List comprehension uses [ ], set uses { }, dictionary uses { key: value }.

3. General syntax: [expression for item in iterable if condition].

4. If-else inside a comprehension is placed BEFORE the for keyword.

5. A function is a reusable block of code defined with the def keyword.

6. Argument types: positional, keyword, default, *args, **kwargs.

7. Default arguments must always come after non-default ones.

8. *args collects extra positional args as a tuple; **kwargs as a dictionary.

9. Use * to unpack a list/tuple into arguments and ** to unpack a dictionary.

10. A recursive function calls itself; it must have a base case to stop.

11. Recursion is elegant but may be slower and use more memory than loops.

Practice Questions

Short Answer:

Q1. Define list, set, and dictionary comprehension with one example each.

Q2. Differentiate between parameter and argument.

Q3. Explain *args and **kwargs with example.

Q4. What is a recursive function? Give the two essential parts.

Q5. Differentiate between recursion and iteration.

Q6. What is the difference between default and keyword arguments?

Q7. Why is unpacking useful in Python? Give an example.

Programming:

Q1. Write a program using list comprehension to print squares of even numbers from 1 to 20.

Q2. Write a program using set comprehension to find unique vowels from a sentence.

Q3. Write a program using dictionary comprehension to map numbers 1–5 to their cubes.

Q4. Write a function to check whether a number is prime.

Q5. Write a function add(*nums) that returns the sum of any number of arguments.

Q6. Write a recursive function to find the factorial of a number.

Q7. Write a recursive function to print the Fibonacci series up to N terms.

Q8. Write a recursive function to find the sum of digits of a number (e.g., 1234 -> 10).

Q9. Write a function info(**details) that prints all key-value pairs passed to it.

Q10. Write a program that uses both a function and a comprehension together to filter passed students from a list of marks.

UNIT 5: FUNCTIONAL PROGRAMMING

Lambda, Higher-Order Functions, map(), filter(), reduce()

1. Introduction to Functional Programming

Functional Programming is a style of writing programs where **functions are treated as values** — they can be passed as arguments, returned from other functions, and stored in variables, just like numbers or strings.

Key ideas of Functional Programming:

- Functions are 'first-class objects' — they can be assigned to variables
- Pass functions as arguments to other functions
- Return functions from other functions
- Avoid changing data once it is created (work on copies)
- Write small, reusable, single-purpose functions

In this unit, we will study four important tools:

- **Lambda functions** — small, one-line anonymous functions
- **Higher-order functions** — functions that take other functions as input or return them
- **map(), filter(), reduce()** — three powerful built-in higher-order functions

2. Lambda Functions

A **lambda function** is a small, **anonymous** (nameless) function that is written in a single line. It is also called an *anonymous function* because it does not need a name like a normal function defined with def.

Syntax:

```
lambda  arguments  :  expression
```

Structure breakdown:

```
lambda    x , y    :    x + y
   |       |         |
keyword  arguments   expression
      (single line, returns value automatically)
```

Important characteristics:

- Can have any number of arguments, but only ONE expression
- Automatically returns the value of the expression (no return keyword)
- Cannot contain multiple statements or loops
- Mostly used inside higher-order functions like map(), filter(), reduce()

(a) Normal function vs Lambda

Normal function:

```
def square(x):
    return x * x

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

Same task using a lambda function:

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

Both produce the same result, but the lambda version takes only **one line**.

(b) Lambda with multiple arguments

```
add  = lambda a, b: a + b
mult = lambda a, b: a * b

print(add(10, 20))    # 30
print(mult(5, 6))     # 30
```

(c) Lambda with no arguments

```
greet = lambda: "Hello, BCA Students!"
print(greet())       # Hello, BCA Students!
```

(d) Lambda with conditional expression (if-else)

```
is_even = lambda n: "Even" if n % 2 == 0 else "Odd"

print(is_even(4))     # Even
print(is_even(7))     # Odd
```

(e) Lambda used immediately (rarely needed)

```
# Define and call in the same line
print((lambda x, y: x + y)(15, 25))   # 40
```

3. Higher-Order Functions

A **higher-order function** is a function that does at least **one of these two things**:

- Takes another function as an argument
- Returns a function as its result

(a) Passing a function as an argument

```
def square(n):
    return n * n

def apply(func, value):       # 'func' will hold a function
    return func(value)

print(apply(square, 5))       # 25
print(apply(lambda x: x+10, 5)) # 15
```

(b) Returning a function from another function

```
def make_multiplier(n):
    return lambda x: x * n     # returns a function

double = make_multiplier(2)
triple = make_multiplier(3)

print(double(10))             # 20
print(triple(10))             # 30
```

Python provides three built-in higher-order functions that we use very often: map(), filter(), and reduce(). The remaining sections of this unit cover each one in detail.

4. The map() Function

The **map()** function applies a given function to **every item** of an iterable (like a list or tuple) and returns a **map object** which we usually convert into a list.

Syntax:

```
map( function , iterable )
```

How it works (visual):

```
Input  list :  [ 1 ,  2 ,  3 ,  4 ,  5 ]
                 |    |    |    |    |
```

```
function f(x):   *   *   *   *   *      f(x) = x * x
                 |   |   |   |   |
Output list:   [ 1 ,  4 ,  9 , 16 , 25 ]
```

(a) Using map() with a normal function

```
def square(x):
    return x * x

nums = [1, 2, 3, 4, 5]
result = map(square, nums)
print(list(result))     # [1, 4, 9, 16, 25]
```

(b) Using map() with a lambda (most common style)

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

squares = list(map(lambda x: x*x, nums))
cubes   = list(map(lambda x: x**3, nums))

print(squares)   # [1, 4, 9, 16, 25]
print(cubes)     # [1, 8, 27, 64, 125]
```

(c) Using map() with multiple iterables

```
a = [1, 2, 3]
b = [10, 20, 30]

sums = list(map(lambda x, y: x + y, a, b))
print(sums)      # [11, 22, 33]
```

(d) String example with map()

```
names = ["ravi", "amit", "priya"]
caps = list(map(str.upper, names))
print(caps)      # ['RAVI', 'AMIT', 'PRIYA']
```

map() returns a **map object**, not a list. Use list() or tuple() to see the actual values.

5. The filter() Function

The **filter()** function **selects only those items** from an iterable for which the given function returns **True**. Items returning False are removed.

Syntax:

```
filter( function , iterable )
```

How it works (visual):

```
Input list :  [ 1 ,  2 ,  3 ,  4 ,  5 ,  6 ]
                 |    |    |    |    |    |
function:     f(x) returns True if x is even
                 F    T    F    T    F    T
                      |         |         |
Output list:     [    2 ,       4 ,       6 ]
```

(a) Using filter() with a normal function

```
def is_even(x):
    return x % 2 == 0

nums = [1, 2, 3, 4, 5, 6]
evens = list(filter(is_even, nums))
print(evens)     # [2, 4, 6]
```

(b) Using filter() with a lambda

```
nums = [10, 25, 30, 45, 60, 75]

mt30 = list(filter(lambda x: x > 30, nums))
print(mt30)      # [45, 60, 75]
```

(c) Filter strings

```
names = ["Ravi", "Am", "Priya", "Su", "Deepak"]

# keep only names with length > 3
long_names = list(filter(lambda n: len(n) > 3, names))
print(long_names)   # ['Ravi', 'Priya', 'Deepak']
```

(d) Remove falsy values

If you pass None as the function, filter() automatically removes all values that are 'falsy' (0, '', None, False).

```
data = [0, 1, "", "Hi", None, "Python", False, 5]
clean = list(filter(None, data))
print(clean)       # [1, 'Hi', 'Python', 5]
```

6. The reduce() Function

The **reduce()** function applies a function to the items of an iterable **two at a time**, combining them step-by-step into a **single final value**.

Important: reduce() is not a built-in function in Python 3 — you must import it from the **functools** module.

Syntax:

```
from functools import reduce

reduce( function , iterable [, initial_value] )
```

How it works (visual) — sum of [1, 2, 3, 4]:

```
[ 1 , 2 , 3 , 4 ]
   \ /
    3
    \ /
     6
     \ /
     10   <- final reduced value

Step 1:  1 + 2  = 3
Step 2:  3 + 3  = 6
Step 3:  6 + 4  = 10
```

(a) Sum of all numbers

```
from functools import reduce

nums = [1, 2, 3, 4, 5]
total = reduce(lambda a, b: a + b, nums)
print(total)     # 15
```

(b) Product (multiplication) of all numbers

```
from functools import reduce

nums = [1, 2, 3, 4, 5]
prod = reduce(lambda a, b: a * b, nums)
print(prod)      # 120
```

(c) Find maximum value using reduce()

```
from functools import reduce

nums = [10, 45, 22, 78, 33]
biggest = reduce(lambda a, b: a if a > b else b, nums)
print(biggest)   # 78
```

(d) Concatenate strings

```
from functools import reduce

words = ["I ", "love ", "Python ", "Programming"]
sentence = reduce(lambda a, b: a + b, words)
print(sentence)  # I love Python Programming
```

(e) Using an initial value

```
from functools import reduce

nums = [1, 2, 3, 4]
total = reduce(lambda a, b: a + b, nums, 100)   # start from 100
print(total)     # 110
```

7. Comparison: map() vs filter() vs reduce()

Feature	map()	filter()	reduce()
Purpose	Apply function to every item	Select items where condition is True	Combine all items into one value
Output size	Same as input	Less than or equal to input	A single value
Returns	map object	filter object	Single value
Function returns	Any value	True / False	Combined value of two args
Needs import?	No (built-in)	No (built-in)	Yes (from functools)
Common use	Convert / transform data	Filter / clean data	Total, product, max, etc.

8. Using Lambda with map(), filter(), reduce() Together

Example 1: Sum of squares of even numbers

```
from functools import reduce

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

# Step 1: keep only even numbers      -> filter
# Step 2: square each of them         -> map
# Step 3: add them all                -> reduce

evens   = filter(lambda x: x % 2 == 0, nums)   # 2, 4, 6
squares = map(lambda x: x*x, evens)            # 4, 16, 36
total   = reduce(lambda a, b: a + b, squares)  # 56

print(total)     # 56
```

Example 2: One-line version (chained)

```
from functools import reduce

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

total = reduce(lambda a, b: a + b,
               map(lambda x: x*x,
                   filter(lambda x: x % 2 == 0, nums)))

print(total)     # 56
```

Example 3: Marks Processing Program

```
from functools import reduce

marks = [78, 45, 32, 90, 55, 28, 67]

# Passed students (>= 40)
passed = list(filter(lambda m: m >= 40, marks))
print("Passed Marks :", passed)

# Add 5 grace marks to everyone
updated = list(map(lambda m: m + 5, marks))
print("After Grace  :", updated)

# Total marks scored by passed students
total = reduce(lambda a, b: a + b, passed)
print("Passed Total :", total)

# Average of passed students
avg = total / len(passed)
print("Passed Avg   :", avg)
```

Quick Revision — Important Points

1. Lambda is a small one-line anonymous function: lambda args : expression

2. A lambda can have many arguments but only one expression.

3. Lambda is mostly used with map(), filter(), and reduce().

4. A higher-order function takes a function as input or returns a function.

5. map(func, iterable) applies func to every item; output size = input size.

6. filter(func, iterable) keeps items where func returns True.

7. reduce(func, iterable) reduces all items into a single value (needs functools).

8. map() and filter() return iterator objects — wrap with list() to display.

9. filter(None, data) removes all falsy values (0, '', None, False).

10. map / filter / reduce can be combined to write powerful one-line programs.

Practice Questions

Short Answer:

Q1. Define lambda function with syntax and one example.

Q2. What is a higher-order function? Give one example.

Q3. Differentiate between a normal function and a lambda function.

Q4. Differentiate between map(), filter(), and reduce().

Q5. Why must reduce() be imported from the functools module?

Q6. Can a lambda contain multiple statements? Justify your answer.

Q7. What does filter(None, data) do?

Programming:

Q1. Write a lambda function that returns the cube of a number.

Q2. Write a lambda function to find the larger of two numbers.

Q3. Use map() with a lambda to convert a list of temperatures from Celsius to Fahrenheit.

Q4. Use filter() with a lambda to keep only positive numbers from a given list.

Q5. Use map() to convert a list of names to uppercase.

Q6. Use reduce() to calculate the factorial of N.

Q7. Use reduce() to find the maximum number in a list (without using max()).

Q8. Use map() and filter() together to print the squares of all odd numbers from 1 to 20.

Q9. Write a program that uses filter, map and reduce together to find the sum of cubes of all even numbers from 1 to 10.

Q10. Use a lambda inside sorted() to sort a list of tuples [(name, marks), ...] by marks in descending order.

UNIT 6: MODULES, PACKAGES AND NAMESPACES

PART A : MODULES

1. Introduction to Modules

A **module** in Python is simply a **.py file** that contains Python code — variables, functions, classes, or runnable statements — which can be reused in other programs.

Think of a module as a **toolbox**. Instead of writing the same code in every program, we keep useful tools (functions) in a module and **import** them whenever needed.

Why use modules?

- Code reusability — write once, use many times
- Better organization of large programs
- Easier maintenance and debugging
- Avoid name conflicts (each module has its own namespace)
- Allow team members to work on separate modules

Three categories of modules in Python:

- **Built-in modules** — already included with Python (math, random, os, sys, datetime, etc.)
- **Third-party modules** — installed using pip (numpy, pandas, requests, etc.)
- **User-defined (custom) modules** — created by the programmer for their own program

2. The Main Module

The file you **run directly** is called the **main module**. When Python runs a file, it gives it a special name: **__main__**.

Every Python file has a built-in variable called __name__:

- If the file is run directly, __name__ = **"__main__"**
- If the file is imported into another program, __name__ = **name of the file (without .py)**

Example — file test.py:

```
def hello():
    print("Hello from test.py")

print("__name__ value is:", __name__)

if __name__ == "__main__":
    print("This file is run directly")
    hello()
else:
    print("This file was imported as a module")
```

Run it directly: python test.py

```
__name__ value is: __main__
This file is run directly
Hello from test.py
```

If the same file is imported elsewhere:

```
import test

# Output:
# __name__ value is: test
# This file was imported as a module
```

This **if __name__ == "__main__":** trick is very common — it lets a file work both as a standalone program AND as an importable module.

3. Built-in Modules

Python ships with a large 'Standard Library' — hundreds of pre-written modules ready to use. Here are some commonly used ones:

Module	Purpose	Sample function

math	Mathematical functions	math.sqrt(25)
random	Random numbers and choices	random.randint(1, 10)
os	Interact with operating system	os.getcwd()
sys	System-specific parameters	sys.version
datetime	Work with dates and times	datetime.date.today()
time	Time-related functions	time.sleep(2)
statistics	Mean, median, mode, stdev	statistics.mean([1,2,3])
json	Read/write JSON data	json.dumps(data)

Example using the math module

```
import math

print(math.sqrt(25))      # 5.0
print(math.pi)           # 3.141592653589793
print(math.factorial(5))  # 120
print(math.pow(2, 10))    # 1024.0
print(math.ceil(4.2))     # 5
print(math.floor(4.8))    # 4
```

Example using the random module

```
import random

print(random.randint(1, 100))          # any number from 1 to 100
print(random.choice(["a", "b", "c"]))   # picks one randomly
print(random.random())                 # 0.0 to 1.0

nums = [10, 20, 30, 40, 50]
random.shuffle(nums)
print(nums)                            # shuffled order
```

Example using the datetime module

```
import datetime

today = datetime.date.today()
now   = datetime.datetime.now()

print("Today:", today)        # e.g., 2025-08-12
print("Now  :", now)          # e.g., 2025-08-12 09:30:45.123
```

4. Custom (User-Defined) Modules

Any Python file that you create can be used as a module — just save your functions/variables in a .py file and import them in another file.

Step 1: Create the module file

Create a file named mymath.py:

```
# File: mymath.py

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

def sub(a, b):
    return a - b

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

PI = 3.14159
```

Step 2: Use the module in another file

Create another file main.py in the **same folder**:

```
# File: main.py
import mymath

print(mymath.add(10, 5))    # 15
print(mymath.sub(10, 5))    # 5
print(mymath.mul(10, 5))    # 50
print(mymath.PI)            # 3.14159
```

Both files must be in the **same folder** (or the module folder must be on Python's search path) for the import to work.

5. Importing a Module — Different Ways

(a) import module_name

Imports the entire module. Use module name + dot to access its members.

```
import math

print(math.sqrt(36))   # 6.0
print(math.pi)         # 3.14159...
```

(b) from module import name

Imports only specific names from the module. Use them directly without prefix.

```
from math import sqrt, pi

print(sqrt(49))    # 7.0
print(pi)          # 3.14159...
```

(c) from module import *

Imports **all** public names from the module. Quick but may cause name conflicts — not recommended for large programs.

```
from math import *

print(sqrt(81))    # 9.0
print(factorial(5))# 120
print(pi)          # 3.14159...
```

(d) import module as alias

Renames a long module name into a short alias for convenience.

```
import math as m

print(m.sqrt(64))     # 8.0
print(m.pow(2, 8))    # 256.0

# Common in real-world code:
# import numpy as np
# import pandas as pd
```

(e) from module import name as alias

```
from math import factorial as f

print(f(6))    # 720
```

Comparison of import styles

Statement	How to use	When to use
import math	math.sqrt(25)	When using many names from the module
from math import sqrt	sqrt(25)	When using only a few specific names
from math import *	sqrt(25)	Quick scripts only — risky in large code
import math as m	m.sqrt(25)	When the name is long or used very often

6. Exploring a Module — dir() and help()

Python provides two helpful built-in functions to discover what a module contains:

dir(module) — list of all names

```
import math
```

```
print(dir(math))
# ['acos', 'asin', 'ceil', 'cos', 'e', 'factorial', 'floor', 'pi', 'pow', 'sqrt', ...]
```

help(module or function) — detailed documentation

```
import math

help(math.sqrt)

# Output:
# Help on built-in function sqrt in module math:
# sqrt(x, /)
#     Return the square root of x.
```

PART B : PACKAGES

7. Packages

A **package** is a **folder** that contains multiple related Python modules. It is a way to organize many modules into one place — like keeping related tools in different drawers of the same toolbox.

A folder becomes a Python package when it contains a special file named __init__.py (it can be empty).

Visualising a package

```
myproject/              <- main project folder
|
+-- main.py             <- the program you run
|
+-- school/             <- PACKAGE  (a folder)
    |
    +-- __init__.py     <- marks this folder as a package
    +-- student.py      <- module 1
    +-- teacher.py      <- module 2
    +-- subjects.py     <- module 3
```

Comparison:

Module	Package
A single .py file	A folder of related .py files
Contains functions, variables, classes	Contains modules (and possibly sub-packages)
Example: math.py	Example: numpy/, scikit-learn/
No __init__.py needed	Must contain __init__.py

8. Creating and Using a Package

Step 1: Create the folder structure

```
myproject/
    |
+-- main.py
    |
+-- school/
        |
+-- __init__.py
+-- student.py
+-- teacher.py
```

Step 2: Write the modules

File: school/student.py

```
def show():
    print("This is the student module")

def total_marks(marks):
    return sum(marks)
```

File: school/teacher.py

```
def show():
    print("This is the teacher module")

def greet(name):
    return "Welcome " + name
```

File: school/__init__.py (can be empty, or used to set up the package)

```
# Optional: code to run when the package is imported
print("School package loaded")
```

Step 3: Use the package in main.py

```
# File: main.py

from school import student, teacher

student.show()                  # This is the student module
print(student.total_marks([80, 90, 75]))   # 245

teacher.show()                  # This is the teacher module
print(teacher.greet("Mr. Deepak"))   # Welcome Mr. Deepak
```

Other ways to import from a package

```
# Import specific function only
from school.student import total_marks
print(total_marks([10, 20, 30]))     # 60

# Import the whole sub-module
import school.teacher
school.teacher.show()
```

PART C : NAMESPACE & SCOPE

9. Namespace

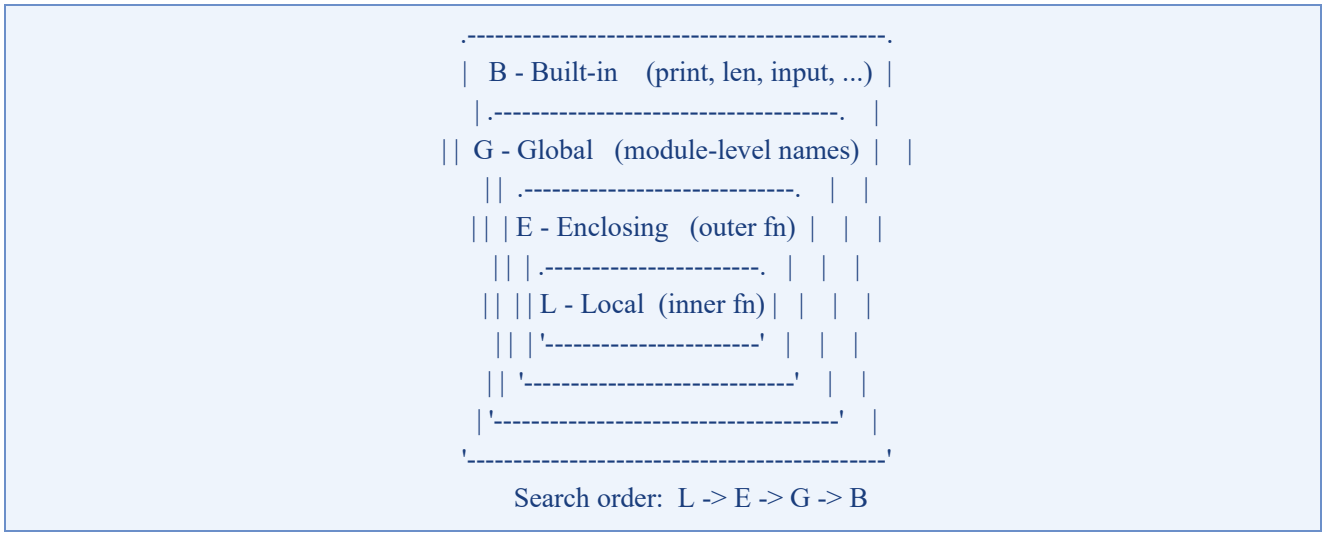
A **namespace** is a container that holds the mapping of **names to objects**. It tells Python where to find a variable or function when it is referenced.

Think of a namespace as a **dictionary** where keys are names (variable names, function names) and values are the actual objects.

Python has three main types of namespaces:

- **Built-in namespace** — contains names of built-in functions like print(), len(), input(). Available everywhere.
- **Global namespace** — contains names defined at the top level of a module/script.
- **Local namespace** — contains names defined inside a function. Created when the function is called and destroyed when it returns.

Visualising namespaces (LEGB Rule)



When Python sees a name, it looks for it in this order: Local, then Enclosing, then Global, then Built-in. This is called the LEGB rule.

10. globals() and locals()

Python provides two built-in functions to **see what's inside a namespace**:

- **globals()** — returns a dictionary of all names in the current global namespace
- **locals()** — returns a dictionary of all names in the current local namespace

Example: globals()

```
x = 10
name = "Deepak"

def hello():
    print("Hi")

print(globals())
# {... 'x': 10, 'name': 'Deepak', 'hello': <function hello>, ...}
```

Example: locals()

```
def show():
    a = 5
    b = "Python"
    print(locals())

show()
# {'a': 5, 'b': 'Python'}
```

Modifying a global variable from inside a function

Use the **global** keyword to tell Python that you want to modify a global variable, not create a new local one.

```
count = 0

def increment():
    global count        # tells Python to use the global 'count'
    count = count + 1

increment()
increment()
print(count)            # 2
```

> Without **global**, any assignment inside a function creates a **new local variable** — the global one stays unchanged.

11. Inner Functions (Nested Functions)

An **inner function** is a function defined **inside** another function. The outer function is sometimes called the *enclosing function*.

Why use inner functions?

- To group helper code that is needed only inside one specific function
- To hide functionality from the rest of the program
- To remember values from the outer function (closures)

Basic inner function

```
def outer():
    print("Outer function")

    def inner():
        print("Inner function")

    inner()        # call inner from inside outer

outer()
# Outer function
# Inner function

# inner() cannot be called from outside outer()
```

Inner function accessing outer variable

```
def greet(name):
    msg = "Hello, "

    def display():
        print(msg + name)   # uses outer 'msg' and 'name'

    display()

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

Modifying an outer variable using nonlocal

To modify a variable from the **enclosing** function (not the global one), use the **nonlocal** keyword.

```
def outer():
    count = 0

    def inner():
        nonlocal count
        count = count + 1
        print("count =", count)

    inner()    # count = 1
    inner()    # count = 2
    inner()    # count = 3

outer()
```

12. Scope of a Variable

The **scope** of a variable is the region of the program where that variable can be accessed.

Python has four scopes (LEGB rule):

Scope	Where it lives	Example
Local (L)	Inside the current function	x = 10 inside def fn()
Enclosing (E)	Inside an outer function (for nested fns)	outer's variables seen by inner
Global (G)	Top level of the module/script	x = 10 at file top
Built-in (B)	Provided by Python itself	print, len, input

Local vs Global scope example

```
x = 100            # global variable

def show():
    x = 50         # local variable - separate from global
    print("Inside  :", x)   # 50

show()
print("Outside :", x)       # 100
```

All four scopes in one example

```
x = "global"

def outer():
    x = "enclosing"

    def inner():
        x = "local"
        print(x)       # local

    inner()
    print(x)           # enclosing

outer()
print(x)               # global
print(len("hello"))    # built-in (len)
```

Output:

```
local
enclosing
global
5
```

13. global vs nonlocal Keywords

Keyword	Refers to	When to use
global	A variable in the global namespace	To modify a top-level variable from inside a function
nonlocal	A variable in the nearest enclosing function	To modify a variable of an outer function from inside an inner function

Combined example:

```
msg = "Hello"           # global

def outer():
    msg = "Hi"          # enclosing

    def inner_global():
        global msg
        msg = "Namaste"  # changes the GLOBAL msg

    def inner_nonlocal():
        nonlocal msg
        msg = "Bonjour"  # changes the ENCLOSING msg

    inner_nonlocal()
    print("Inside outer :", msg)   # Bonjour

    inner_global()

outer()
print("Global msg   :", msg)       # Namaste
```

14. Combined Mini-Project

A small example that combines a custom module + import + namespace + scope.

File: calculator.py (custom module)

```
# 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 "Cannot divide by 0"

if __name__ == "__main__":
    print("Running calculator.py directly")
    print(add(10, 5))
```

File: main.py

```
# main.py
```

```
import calculator as calc

print("===== Simple Calculator =====")
a = float(input("Enter first number : "))
b = float(input("Enter second number: "))

print("Add :", calc.add(a, b))
print("Sub :", calc.sub(a, b))
print("Mul :", calc.mul(a, b))
print("Div :", calc.div(a, b))
```

Quick Revision — Important Points

1. A module is a single .py file containing functions/variables.

2. A package is a folder of related modules; it must contain __init__.py.

3. The file you run directly is the main module; its __name__ becomes '__main__'.

4. if __name__ == '__main__': lets a file run as both program and module.

5. Built-in modules: math, random, os, sys, datetime, etc.

6. Import styles: import m, from m import x, from m import *, import m as n.

7. dir(m) lists names in a module; help(m) shows documentation.

8. A namespace maps names to objects; Python uses Local -> Enclosing -> Global -> Built-in (LEGB).

9. globals() and locals() return the current namespace as a dictionary.

10. Use the global keyword to modify a global variable inside a function.

11. Use the nonlocal keyword to modify an enclosing function's variable from an inner function.

12. Inner functions are functions defined inside other functions.

Practice Questions

Short Answer:

Q1. Define module and package with one example each.

Q2. What is the main module? What is the value of __name__ in it?

Q3. Differentiate between built-in, third-party, and user-defined modules.

Q4. Explain four different ways to import a module.

Q5. What is a namespace? List the three types in Python.

Q6. Differentiate between local and global scope.

Q7. Differentiate between the global and nonlocal keywords.

Q8. What is an inner function? Give one use case.

Q9. What is the LEGB rule in Python?

Q10. What is the role of __init__.py in a package?

Programming:

Q1. Create a custom module myutils.py with functions to find the area of a circle, square, and rectangle. Import and use it in another file.

Q2. Write a program that uses the math module to find the square root, factorial, and value of pi.

Q3. Write a program that uses the random module to generate 5 random numbers between 1 and 100.

Q4. Write a program that prints today's date, current time, and the day of the week using the datetime module.

Q5. Create a package college containing two modules: students.py and faculty.py. Use functions from both in a main.py file.

Q6. Write a program demonstrating the use of globals() and locals().

Q7. Write a program with a global counter that is incremented inside a function using the global keyword.

Q8. Write a program with an outer and inner function where inner modifies the outer variable using nonlocal.

Q9. Write a program demonstrating the LEGB rule using local, enclosing, global, and built-in scopes.

Q10. Create a custom module that uses if __name__ == '__main__': for testing, and another file that imports its functions.