UNIT – I

INTRODUCTION TO PYTHON

1.1 INTRODUCTION

Python is a high-level, general-purpose, versatile programming language created by Guido van Rossum in the early 1990s. It was designed with one major goal: *make programming simple and readable*.
Unlike many programming languages that require complex syntax, Python uses clean, English-like statements. Because of its simplicity and wide library support, Python has become a leading language in multiple fields:

- Software development
- Data Science
- Artificial Intelligence
- Machine Learning
- Web Development
- Robotics
- Automation
- Networking
- Scientific computing

Today, Python is considered an essential language in the IT industry and academic research.

1.2 WHY PYTHON?

The popularity of Python is due to several powerful reasons:

1. Easy to learn and write
2. Highly readable syntax
3. Large standard library
4. Works on all operating systems
5. Supports multiple programming paradigms
6. Suitable for modern technologies (AI, ML, Data Science)
7. Strong community support
8. Free and open source

These features make Python a universal programming language suitable for beginners as well as professionals.

1.3 HISTORY OF PYTHON

Python was developed by Guido van Rossum at the **CWI (Centrum Wiskunde & Informatica)** in the Netherlands.

Major milestones:

- **1991** – First public version Python 0.9.0 released
- **1994** – Python 1.0 released
- **2000** – Python 2.0 with garbage collection & list comprehensions
- **2008** – Python 3.0 released (clearer, more powerful version)
- **Today** – Python 3.x used globally

Python was named after the British comedy show **"Monty Python's Flying Circus"**, not the snake.

1.4 FEATURES OF PYTHON

Python provides several advanced features that make programming easy and powerful.

1.4.1 Simple and Easy to Learn

Python syntax is almost like English.

Example:

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

1.4.2 Interpreted Language

Python runs the program **line-by-line**, making debugging easier.

1.4.3 High-Level Language

Programmers need not worry about memory management, hardware details, etc.

1.4.4 Platform Independent

Python runs on Windows, MacOS, Linux without changes.

1.4.5 Object-Oriented

Supports classes, objects, inheritance, polymorphism, encapsulation.

1.4.6 Dynamically Typed

No need to specify data types; Python automatically assigns them.

1.4.7 Extensive Standard Library

Provides modules for:

- Mathematics
- Networking
- Database handling
- File processing
- Graphics
- Data management

1.4.8 Free and Open Source

Python is freely available for use and modification.

1.4.9 Extensible and Embeddable

Python can be combined with C, C++, Java for advanced applications.

1.5 COMPONENTS OF A PYTHON PROGRAM

A Python program contains various components that work together:

1.5.1 Statements

Individual instructions (e.g., input, print, assignment).

1.5.2 Variables

Names that store data in memory.

1.5.3 Expressions

Combination of operators and operands.

1.5.4 Comments

Non-executable lines for documentation.
Single-line → #
Multi-line → triple quotes

1.5.5 Functions

Blocks of reusable code.

1.5.6 Blocks and Indentation

Python uses indentation instead of braces.

Example:

```
if x > 10:
    print("Greater")
```

1.6 PYTHON VIRTUAL MACHINE (PVM)

Python code does not directly run on hardware.
The execution process is:

1. Python source code (.py)
2. Interpreter converts it into **bytecode** (.pyc)
3. **PVM (Python Virtual Machine)** executes the bytecode
4. Output displayed

PVM responsibilities:

- Bytecode interpretation
- Memory allocation
- Garbage collection
- Exception handling

Just like Java has JVM, Python has PVM.

1.7 MEMORY MANAGEMENT IN PYTHON

Python handles memory automatically through:

1. **Heap Memory** – stores objects
2. **Stack Memory** – stores function calls, local variables
3. **Python Memory Manager** – allocates memory internally
4. **Garbage Collector** – removes unused objects

Python developers do **not** need to manually allocate or free memory.

1.8 GARBAGE COLLECTION IN PYTHON

Garbage Collection removes unused objects from memory to avoid memory leaks.

Python uses:

1. **Reference Counting**
   - Every object has a reference counter
   - When count becomes zero → deleted
2. **Generational Garbage Collector**
   - Divides memory into generations
   - Older objects cleaned less frequently
   - Faster and efficient

Garbage collection improves performance and prevents memory overflow.

1.9 INSTALLING PYTHON

Python is available freely on its official website.
The installation procedure is simple and suitable for all major operating systems.

1.9.1 Downloading Python

Follow these steps:

Step 1: Open the official website

https://www.python.org

Step 2: Click on Downloads

Python automatically suggests the best version for your OS (Windows/Mac/Linux).

Step 3: Choose the latest stable version

For example: Python 3.12.x (or whichever is latest)

Step 4: Click "Download"

1.9.2 INSTALLING PYTHON ON WINDOWS

Once the installer is downloaded, follow these steps:

Step 1: Run the Installer

Double-click the downloaded file (example: python-3.x.x.exe)

Step 2: VERY IMPORTANT

Tick the checkbox:
Add Python to PATH

(If PATH is not added, Python won't run from command prompt)

Step 3: Click Install Now

The installer sets up:

- Python interpreter
- pip package manager
- IDLE (Python editor)
- Documentation
- Standard library

Step 4: Wait until installation completes

A confirmation message appears: *"Setup was successful."*

1.9.3 VERIFYING PYTHON INSTALLATION

Open **Command Prompt** and type:

python --version
or
py --version

If Python is installed correctly, you will see a version number.

Example:
Python 3.12.1

1.10 VERIFYING THE PATH TO PYTHON

PATH is an environment variable that decides where the OS will look for executable files.

If PATH is set correctly:

python
(or)
py

should open the Python interpreter.

To check:

1. Open Command Prompt
2. Type:

   where python

Windows will display the full path, for example:
C:\Users\Admin\AppData\Local\Programs\Python\Python312\python.exe

If no path is shown, Python PATH is not configured.

1.11 RUNNING PYTHON PROGRAMS

Python code can be run in three ways:

1.11.1 Using the Python Interactive Mode

Type:

python

print("Hello World")

The ">>>" prompt indicates you are in interactive mode.

Used for:

- Testing
- Small expressions
- Quick calculations

1.11.2 Using IDLE

IDLE is Python's Internal Development Environment.

Steps:

1. Search for **IDLE** in Start Menu
2. Open it
3. Type code
4. Press **F5** to run

1.11.3 Running Python Files (.py)

Create a file:

example.py

Add code:

print("Python Programming")

Run using command prompt:

python example.py

1.12 INSTALLING pandas

pandas is a powerful library used for:

- Data analysis
- Reading/writing CSV
- Handling tables
- Data cleaning

pandas does **not** come with default Python installation.

You must install it manually.

1.12.1 Installing pandas Using pip

Open Command Prompt:

pip install pandas

This command:

- Downloads pandas
- Installs required dependencies (like numpy)
- Registers it in the Python environment

1.12.2 Checking pandas Installation

python
import pandas
print(pandas.**version**)

If no error → installation successful.

1.12.3 Alternative: pip3 for Linux/Mac

pip3 install pandas

1.12.4 Installing pandas Offline

If internet is not available:

1. Download pandas .whl file from PyPI
2. Open CMD at the folder
3. Run:

pip install pandas-version.whl

1.13 VERIFYING INSTALLED PACKAGES

Python provides tools to see all installed packages.

1.13.1 Using pip list

pip list

Shows:

- Installed libraries
- Versions

Example:

numpy 1.26.4
pandas 2.2.0

1.13.2 Using pip show

pip show pandas

Shows:

- Name
- Version
- Location
- Dependencies
- Author

1.13.3 Using pip freeze

pip freeze

Outputs complete installed package list
(This is used to create requirements.txt files)

1.13.4 Checking Package Version Inside Python

python
import pandas
print(pandas.**version**)

1.13.5 Checking Installation Path

```
python
import pandas
print(pandas.file)
```

1.14 SAMPLE PYTHON PROGRAMS (BEGINNER LEVEL)

Program 1: Simple Output

```
print("Hello Python")
```

Program 2: Addition of Two Numbers

```
a = 10
b = 20
print("Sum =", a + b)
```

Program 3: Using Variables

```
name = "Hari Krishna"
age = 28
print("Name:", name)
print("Age:", age)
```

Program 4: Using Input

```
x = int(input("Enter a number: "))
y = int(input("Enter another number: "))
print("Result =", x + y)
```

Program 5: Area of Circle

```
PI = 3.14
radius = float(input("Enter radius: "))
area = PI * radius * radius
print("Area =", area)
```

UNIT – II

DATA TYPES & OPERATORS

2.1 COMMENTS IN PYTHON

Comments are non-executable statements written inside a program to increase readability. They help programmers understand logic, purpose, and functioning of code.

Python supports **two types** of comments:

2.1.1 Single-Line Comments

Begin with #

Example:

This is a comment

x = 10 # assigning value

Everything after # on the same line is ignored by Python.

2.1.2 Multi-Line Comments

Python does not have a special multi-line comment symbol like /* */ in C.
But we use **triple quotes** for multi-line comments.

Example:

"""
This is a multi-line comment
Used for documentation
"""

These are ignored during execution.

2.2 IDENTIFIERS

Identifiers are the **names** given to variables, functions, classes, lists, modules, etc.

Example identifiers:

- student
- total_marks
- addNumbers
- List1
- EmployeeData

Identifiers should follow certain rules.

2.2.1 Rules for Identifiers

1. Identifiers can contain **letters**, **digits**, and **underscore**.
   Example: roll_no, total1
2. Identifier **must not begin with a digit**.
   Wrong: 1value
   Correct: value1
3. No special characters allowed except underscore.
   Wrong: total%
   Wrong: name@user
4. Python identifiers are **case-sensitive**.
   variable, Variable, VARIABLE are different.
5. Keywords cannot be used as identifiers.
   Wrong: if = 10
   Wrong: class = "python"
6. Identifier length has no limit.

2.3 KEYWORDS (RESERVED WORDS)

Keywords are predefined words that have special meaning to Python.
They cannot be used as variable names or function names.

Python has **35+ keywords** (exact number depends on version).

Some important keywords:

False
True
None
and
or
not
if
elif
else
for
while
break
continue
return
class
def
with
try
except
import
from
pass
raise
global
lambda

Example (wrong use):

return = 5 → Error (reserved word)

2.4 VARIABLES IN PYTHON

A variable is a name that stores a value in memory.
In Python, variables are created **automatically** when a value is assigned.

Example:

```
x = 10
name = "Hari"
pi = 3.14
```

2.4.1 Characteristics of Python Variables

1. Variables do not require data type declaration.
   Python automatically detects type.

2. Variable type can change during execution (dynamic typing).
   x = 10
   x = "Hello"
3. Variables reference objects stored in memory.

2.4.2 Assigning Values to Variables

You can assign:

Single variable:

x = 5

Multiple variables in one line:

a, b, c = 10, 20, 30

Same value to multiple variables:

x = y = z = 0

2.5 CONSTANTS IN PYTHON

(Already explained earlier but included here again as per syllabus sequence)

A constant is a value that should not change during program execution.
Python does not support true constants, but developers write constant names in **uppercase**.

Examples:

PI = 3.14159
MAX = 100
SCHOOL_NAME = "ABC College"

2.6 LITERALS

Literals are the **actual values** assigned to variables or constants.

Example:

x = 10
name = "Hari"
status = True

Python supports several types of literals:

2.6.1 Numeric Literals

Integer Literals:

x = 10

Float Literals:

pi = 3.14

Complex Literals:

c = 5 + 3j

2.6.2 String Literals

A sequence of characters inside quotes:

name = "Python"
message = 'Welcome'

2.6.3 Boolean Literals

True
False

Example:

flag = True

2.6.4 Special Literal: None

None means no value or empty value.

Example:

x = None

2.7 DATA TYPES IN PYTHON

A *data type* defines what kind of value a variable can store and what operations can be performed on it.
Python supports **six major built-in data types**:

1. Numbers
2. Strings
3. Lists
4. Tuples
5. Sets
6. Dictionaries

Each of these is explained in detail below.

2.7.1 NUMBERS

Numbers represent numeric values.
Python supports 3 types of numeric data:

(a) Integer (int)

Whole numbers, positive or negative.
Examples:
x = 10
age = 25
temp = -5

(b) Floating point (float)

Decimal numbers.
Examples:
pi = 3.14
percentage = 87.5

(c) Complex numbers (complex)

Have real + imaginary part.
Examples:
z = 5 + 3j

You can access real and imaginary parts:
z.real → 5.0
z.imag → 3.0

Type Checking

Use type() function:
type(10) → int
type(3.14) → float
type(5+3j) → complex

2.7.2 STRINGS

A string is a sequence of characters enclosed in quotes.

Examples:
name = "Hari"
college = 'SV University'

2.7.2.1 Creating Strings

- Single quotes: 'Hello'
- Double quotes: "Python"
- Triple quotes (multi-line):
  """
  This is
  multi-line string
  """

2.7.2.2 Accessing Characters (Indexing)

Python uses zero-based indexing.

text = "Python"
text[0] → 'P'
text[3] → 'h'

2.7.2.3 Slicing Strings

text = "Python"
text[0:3] → 'Pyt'
text[2:] → 'thon'

2.7.2.4 String Operations

- Concatenation:
  "Hello " + "World"
- Repetition:
  "Hi" * 3 → "HiHiHi"
- Membership:
  'a' in "Hari"

2.7.2.5 Important String Methods

lower(), upper(), title(), strip(), replace(), split(), join(), find(), count()

2.7.3 LISTS

A list is an **ordered**, **mutable**, **indexed** collection of items.
Lists are written using square brackets [ ].

Example:
fruits = ["apple", "banana", "orange"]

2.7.3.1 Characteristics

- Ordered
- Changeable (mutable)
- Allows duplicates
- Supports mixed data types

2.7.3.2 Creating Lists

empty = []
numbers = [10, 20, 30]
mixed = [10, "Hari", 3.5]

2.7.3.3 Accessing Elements

numbers[0] → 10
numbers[-1] → last element

2.7.3.4 List Operations

- Concatenation:
  a + b
- Repetition:
  x * 3
- Membership:
  10 in numbers

2.7.3.5 Important List Methods

append(), insert(), remove(), pop(), sort(), reverse(), count(), index(), extend()

2.7.3.6 Slicing

numbers[1:3]

2.7.3.7 Nested Lists

matrix = [[1,2,3],[4,5,6]]
matrix[1][2] → 6

2.7.4 TUPLES

Tuples are **ordered**, **immutable** collections.
Written using parentheses ( ).

Example:
t = (10, 20, "Hari")

2.7.4.1 Characteristics

- Ordered
- Cannot be changed (immutable)
- Faster than lists
- Useful for fixed data

2.7.4.2 Creating Tuples

t1 = ()
t2 = (10,) ← single element
t3 = (10, 20, 30)

2.7.4.3 Accessing Elements

t3[1] → 20

2.7.4.4 Tuple Operations

- Concatenation
- Repetition
- Membership

2.7.4.5 Functions with Tuples

len(), max(), min(), sum()

2.7.4.6 Nested Tuples

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

Sequence in Python

In Python, a **sequence** is a data type that stores a collection of **ordered elements**.
Each element in a sequence has a **fixed position called index**, starting from **0**.

Types of Sequences in Python

1. String (`str`)

A string is a sequence of characters enclosed in quotes.

Example:

```
s = "Python"
print(s[0])      # P
print(s[1:4])    # yth
```

2. List (`list`)

A list is a mutable sequence that can store different types of elements.

Example:

```
lst = [10, 20, 30, 40]
lst.append(50)
print(lst)
```

3. Tuple (`tuple`)

A tuple is an immutable sequence. Once created, elements cannot be changed.

Example:

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

4. Range (`range`)

Range represents a sequence of numbers.

Example:

```
r = range(1, 6)
print(list(r))
```

Sequence Operations

Operation	Description	Example
`+`	Concatenation	`[1,2] + [3,4]`
`*`	Repetition	`[1,2] * 2`
`in`	Membership	`2 in [1,2,3]`
`len()`	Length	`len([1,2,3])`

Indexing and Slicing

Indexing Example:

```
x = "Computer"
print(x[0])    # C
print(x[-1])   # r
```

Slicing Example:

```
print(x[1:4])   # omp
print(x[:3])    # Com
print(x[3:])    # puter
```

2.7.5 SETS

A set is an **unordered**, **unindexed**, **mutable**, **unique-element** collection.

Example:
s = {10, 20, 30}

2.7.5.1 Characteristics

- Duplicate values removed
- Elements not accessed by index
- Faster membership testing

2.7.5.2 Creating Sets

s = {1, 2, 3}
empty_set = set()

2.7.5.3 Set Operations

- Union: s1 | s2
- Intersection: s1 & s2
- Difference: s1 - s2
- Symmetric Difference: s1 ^ s2

2.7.5.4 Important Set Methods

add(), remove(), discard(), clear(), update()

2.7.6 DICTIONARIES

A dictionary is an **unordered collection** of **key-value pairs**.

Example:
student = {"name": "Hari", "age": 25, "course": "MCA"}

2.7.6.1 Characteristics

- Stores data as key:value
- Keys must be unique
- Values can be duplicates
- Mutable and dynamic

2.7.6.2 Accessing Values

student["name"] → "Hari"

2.7.6.3 Adding/Updating

student["age"] = 26

2.7.6.4 Removing

student.pop("course")

2.7.6.5 Dictionary Methods

keys(), values(), items(), update(), pop(), get(), clear()

2.7.7 TYPE CONVERSION

Python can convert one data type to another.

Implicit Conversion

Automatic by Python.
Example:
x = 10
y = 3.5
z = x + y → float

Explicit Conversion

int(), float(), str(), list(), tuple(), set(), dict()

2.8 OPERATORS

Operators are symbols used to perform operations on variables and values.
Python supports various kinds of operators categorized into:

1. Arithmetic Operators
2. Unary Operators
3. Assignment Operators
4. Relational Operators
5. Logical Operators
6. Boolean Operators
7. Bitwise Operators
8. Membership Operators
9. Identity Operators

Each operator type is explained below with examples.

2.8.1 ARITHMETIC OPERATORS

Used to perform mathematical operations.

Operator	Meaning	Example	Result
+	Addition	10 + 5	15
-	Subtraction	10 - 5	5
*	Multiplication	10 * 5	50
/	Division	10 / 5	2.0
//	Floor Division	10 // 3	3
%	Modulus	10 % 3	1
**	Exponent	2 ** 3	8

Example:

```
a = 15
b = 4
print(a + b) # 19
print(a // b) # 3
print(a % b) # 3
```

2.8.2 UNARY OPERATORS

Unary operators work on a **single operand**.

Unary Plus (+)

Indicates a positive value.

Example:
x = +10

Unary Minus (-)

Used to change the sign of a number.

Example:
x = 7
print(-x)
Output: -7

2.8.3 ASSIGNMENT OPERATORS

Used to assign values to variables.

Operator	Example	Meaning
=	x = 5	Assign 5 to x
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
//=	x //= 3	Floor divide
%=	x %= 3	Modulus assign
**=	x **= 2	Power assign

Example:

```
x = 10
x += 5 # 15
x *= 2 # 30
```

2.8.4 RELATIONAL OPERATORS

Used to compare two values. Always return **True** or **False**.

Operator	Example	Meaning
>	a > b	Greater than
<	a < b	Less than
>=	a >= b	Greater than or equal
<=	a <= b	Less than or equal
==	a == b	Equal
!=	a != b	Not equal

Example:

```
a = 10
b = 20
print(a < b) # True
```

2.8.5 LOGICAL OPERATORS

Used to combine multiple conditions.

Operator	Meaning	Example
and	True if both true	a > 5 and a < 20
or	True if any one true	a > 5 or a < 3
not	Reverses truth	not(a > 5)

Example:

```
x = 10
print(x > 5 and x < 20) # True
```

2.8.6 BOOLEAN OPERATORS

Boolean values: **True**, **False**

Example:
a = True
b = False
print(a and b) # False
print(a or b) # True
print(not a) # False

2.8.7 BITWISE OPERATORS

Operate on **binary bits**.

Operator	Meaning	Example
&	Bitwise AND	5 & 3 → 1
|	Bitwise OR	5 | 3 → 7
^	Bitwise XOR	5 ^ 3 → 6
~	Bitwise NOT	~5 → -6
<<	Left Shift	5 << 1 → 10
>>	Right Shift	5 >> 1 → 2

Example:

5 → 101
3 → 011

5 & 3 → 001 → 1
5 | 3 → 111 → 7

2.8.8 MEMBERSHIP OPERATORS

Used to test whether a value is present in a sequence.

Operator	Example	Result
in	3 in [1,2,3]	True
not in	5 not in [1,2,3]	True

2.8.9 IDENTITY OPERATORS

Check if two variables refer to the **same memory object**.

Operator	**Meaning**
is	Same object
is not	Different object

Example:

a = [1,2,3]
b = a
c = [1,2,3]

a is b → True
a is c → False

(because c is a different object even though values same)

2.9 OPERATOR PRECEDENCE

Determines which operator is evaluated first.

Order (highest to lowest):

1. **() Parentheses**
2. **Exponentiation ( )**
3. **Unary +, -**
4. **Multiplication, Division, Floor Division, Modulus**
5. **Addition, Subtraction**
6. **Relational Operators**
7. **Logical Operators**

Example:
x = 10 + 5 * 2
Output: 20 (multiplication first)

2.10 OPERATOR ASSOCIATIVITY

Defines direction of evaluation.

- Left to Right: +, -, *, /, <, >
- Right to Left: ** (power)

Example:
2 ** 3 ** 2

= 2 ** (3 ** 2)
= 2 ** 9
= 512

UNIT – III

CONTROL STATEMENTS & ARRAYS

3.1 INTRODUCTION

In any programming language, controlling the flow of execution is very important.
A program does **not** always run sequentially from top to bottom.
Depending on conditions, decisions, loops, and user inputs, the program may:

- Skip certain statements
- Repeat statements
- Break out of loops
- Choose different execution paths

Python provides **control flow statements** that make programs intelligent and interactive.

Before learning control statements, we must understand how Python handles **input**, **output**, and **command-line arguments**, because these are used inside control flow.

3.2 INPUT STATEMENTS

Input statements allow users to enter data during program execution.

Python uses:

input() function

Syntax:

variable = input("Message to user")

The input is **always read as a string**, so conversion may be required.

Example:

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

Example with type conversion:

```
x = int(input("Enter a number: "))
y = float(input("Enter a decimal value: "))
print("Sum =", x + y)
```

Important Points:

1. input() always returns **string**
2. Must convert using int(), float() when needed
3. Works in both interactive and file-based programs

3.3 OUTPUT STATEMENTS

Python prints output using:

print() function

Syntax:

```
print(value1, value2, ..., sep=" ", end="\n")
```

Features of print():

- Prints multiple values
- Adds space by default
- Ends with newline unless changed

Example 1:

```
print("Hello Python")
```

Example 2 (multiple values):

```
print("Sum =", 10 + 20)
```

Example 3 (custom separator):

```
print("A", "B", "C", sep="-")
# Output: A-B-C
```

Example 4 (custom ending):

```
print("Hello", end=" ")
print("World")
# Output: Hello World
```

3.4 COMMAND LINE ARGUMENTS

Command-line arguments are values passed to a program from the command prompt.

Python uses the **sys** module.

Step 1: Import sys

```
import sys
```

Step 2: Access arguments using:

sys.argv

- sys.argv[0] → program name
- sys.argv[1] → first argument
- sys.argv[2] → second argument

Example Program (cmd_args.py):

```
import sys
print("Program Name:", sys.argv[0])
print("First Argument:", sys.argv[1])
print("Second Argument:", sys.argv[2])
```

Running:

```
python cmd_args.py hello 123
```

Output:

```
Program Name: cmd_args.py
First Argument: hello
Second Argument: 123
```

Notes:

- All command-line arguments are strings
- Need conversion for numeric values

3.5 CONTROL STATEMENTS INTRODUCTION

Control statements alter the normal flow of execution.
Python supports:

1. **Selection (Decision) Statements**
    - if
    - if-else
    - if-elif
    - nested-if
2. **Looping (Iteration) Statements**
    - while
    - for
    - nested loops
3. **Jump (Transfer) Statements**
    - break
    - continue
    - pass
    - return

Control statements help in:

- Making decisions
- Repeating tasks
- Skipping iterations
- Exiting loops early
- Creating interactive programs

3.6 NEED FOR CONTROL FLOW

Without control statements, a program:

- Runs only in sequential order
- Cannot make decisions
- Cannot repeat tasks
- Cannot skip conditions

Examples where control flow is needed:

- ATM PIN verification
- Student grade calculation
- Salary computation
- Games (conditions + loops)
- Billing systems
- Searching for elements

3.7 TYPES OF CONTROL STATEMENTS

1. Selection Statements

- if
- if-else
- if-elif ladder
- nested if

2. Looping Statements

- while loop
- for loop
- infinite loops
- nested loops

3. Jump Statements

- break
- continue
- pass
- return

3.8 BASIC FLOW OF A PYTHON PROGRAM

A Python program generally follows:

1. Input from user
2. Processing using control statements
3. Printing output

Example:

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

if age >= 18:
   print("Eligible to vote")
else:
   print("Not eligible")
```

3.9 SELECTION STATEMENTS

Selection (decision-making) statements allow a program to **choose a specific block of code** based on a condition.
These conditions generally evaluate to **True** or **False**.

Python supports four types of decision statements:

1. if
2. if–else
3. if–elif–else
4. nested if

Let us study each in detail.

3.9.1 IF STATEMENT

The **if** statement executes a block of code **only when the condition is True**.

Syntax

```
if condition:
   statement(s)
```

Flow

- If condition is True → the block executes
- If condition is False → block is skipped

Example 1

```
age = 20
if age >= 18:
   print("Eligible for voting")
```

Example 2

```
num = 10
if num > 0:
   print("Positive number")
```

3.9.2 IF–ELSE STATEMENT

If the condition is True → executes **if block**
If False → executes **else block**

Syntax

```
if condition:
   statements
else:
   statements
```

Example

```
marks = 45

if marks >= 50:
   print("Pass")
else:
   print("Fail")
```

Explanation

- marks = 45 → condition False
- So else block executes → "Fail"

3.9.3 IF–ELIF–ELSE (MULTI-WAY DECISION)

Used when multiple conditions must be checked.
Only **one block** will execute.

Syntax

```
if condition1:
   block1
elif condition2:
   block2
elif condition3:
   block3
else:
   block4
```

Example: Grade System

```
marks = 82

if marks >= 90:
   print("A Grade")
elif marks >= 80:
   print("B Grade")
elif marks >= 70:
   print("C Grade")
else:
   print("D Grade")
```

Explanation

marks = 82 → B Grade

3.9.4 NESTED IF STATEMENT

One **if** inside another **if** is called **nested if**.
Used when a decision depends on another decision.

Syntax

```
if condition1:
   if condition2:
      statement
   else:
      statement
else:
   statement
```

Example

```
age = 25
nationality = "Indian"

if age >= 18:
   if nationality == "Indian":
      print("Eligible for Voter ID")
   else:
      print("Not an Indian Citizen")
else:
   print("Underage")
```

Explanation

- age >= 18 → True
- nationality == “Indian” → True
- Both True → Eligible

3.9.5 COMPOUND CONDITIONS

Python allows combining multiple conditions using:

- and
- or
- not

Example 1: Using and

```
age = 30
has_id = True
```

```
if age >= 18 and has_id:
    print("Entry allowed")
```

Example 2: Using or

```
day = "Sunday"

if day == "Saturday" or day == "Sunday":
    print("Weekend")
```

Example 3: Using not

```
logged_in = False

if not logged_in:
    print("Please login")
```

3.9.6 SHORT-HAND IF STATEMENT

One-line if statement.

Example

```
x = 10
if x > 5: print("Greater")
```

3.9.7 SHORT-HAND IF–ELSE (TERNARY OPERATOR)

Python allows a one-line if–else expression.

Syntax

```
variable = value_if_true if condition else value_if_false
```

Example

```
age = 17
result = "Adult" if age >= 18 else "Minor"
print(result)
```

Output: Minor

3.9.8 COMMON ERRORS IN IF STATEMENTS

1. Missing colon
2. Wrong indentation
3. Using = instead of ==
4. Wrong logical operator
5. Forgetting parenthesis in complex conditions

Example error:

```
if x = 10:        # wrong
```

Correct:

```
if x == 10:
```

3.10 REAL-TIME EXAMPLES USING SELECTION

Example 1: Even or Odd

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

Example 2: Student Result

```
marks = int(input("Enter marks: "))
if marks >= 35:
   print("Pass")
else:
   print("Fail")
```

Example 3: Largest of 3 Numbers

```
a = 10
b = 50
c = 30

if a > b and a > c:
   print("A is largest")
elif b > c:
   print("B is largest")
else:
   print("C is largest")
```

3.11 LOOPING STATEMENTS

Looping (iteration) statements allow a block of code to run **multiple times**.
Loops are used when we want to repeat actions until a condition is met.

Python supports two main loops:

1. **while loop**
2. **for loop**

Additionally, we have:

- Nested loops
- Infinite loops
- Loop control statements (break, continue, pass)

3.12 WHILE LOOP

The **while loop** repeats a block of code **as long as the condition remains True**.

Syntax

```
while condition:
   statements
```

Flow

1. Check the condition
2. If True → execute block
3. Repeat
4. If False → exit loop

3.12.1 Example: Print 1 to 5

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

Output:

```
1
2
3
```

```
4
5
```

3.12.2 Example: Sum of first N numbers

```
n = int(input("Enter N: "))
i = 1
s = 0

while i <= n:
    s = s + i
    i = i + 1

print("Sum =", s)
```

3.12.3 Example: Reverse a number

```
n = 1234
rev = 0

while n > 0:
    digit = n % 10
    rev = rev * 10 + digit
    n = n // 10

print("Reversed =", rev)
```

3.12.4 Infinite while Loop

A loop becomes infinite when the condition **never becomes False**.

Example:

```
while True:
    print("Running...")
```

3.12.5 Using else with while

Python allows an **else block** with while.

```
i = 1
while i <= 3:
    print(i)
```

```
    i += 1
else:
    print("Loop completed")
```

3.13 FOR LOOP

The **for loop** is used to iterate over sequences like:

- Strings
- Lists
- Tuples
- Sets
- Dictionaries
- Range()

Syntax

```
for variable in sequence:
    statements
```

3.13.1 Example: Print characters in a string

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

3.13.2 Example: Print elements of a list

```
numbers = [10, 20, 30]
for n in numbers:
    print(n)
```

3.13.3 Using range() with for

range() generates a sequence of numbers.

Syntax

- range(stop)
- range(start, stop)
- range(start, stop, step)

Examples:

```
for i in range(5):        # 0 to 4
```

```
    print(i)

for i in range(1, 6):     # 1 to 5
    print(i)

for i in range(1, 10, 2): # odd numbers
    print(i)
```

3.13.4 Sum of first N numbers

```
n = int(input("Enter N: "))
s = 0

for i in range(1, n+1):
    s += i

print("Sum =", s)
```

3.13.5 Nested for loops

Loop inside another loop.

Example: 3×3 Matrix

```
for i in range(3):
    for j in range(3):
        print(i, j)
```

3.13.6 Looping through a dictionary

```
student = {"name": "Hari", "age": 25}

for key in student:
    print(key, ":", student[key])
```

3.13.7 Using else with for

```
for i in range(3):
    print(i)
else:
    print("Loop completed")
```

3.14 NESTED LOOPS

A nested loop means loop **inside** a loop.
Used for:

- Patterns
- Matrices
- Tables
- Multi-level iteration

Example: Printing a Square Pattern

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

3.15 INFINITE FOR LOOPS

Unlike while, for loops generally do not become infinite,
but using range with no stop can create infinite iterators (rarely used).

3.16 LOOP CONTROL STATEMENTS

Used inside loops to alter normal flow.

3.16.1 break Statement

Terminates the loop immediately.

```
for i in range(1, 10):
    if i == 5:
        break
    print(i)
```

Output: 1 2 3 4

3.16.2 continue Statement

Skips the current iteration and goes to next.

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

```
print(i)
```

Output: 1 2 4 5

3.16.3 pass Statement

Used as a placeholder when a statement is required syntactically but you don't want to execute anything.

```
for i in range(5):
    pass
```

3.16.4 return Statement (used inside functions)

Ends function execution and returns a value.

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

3.17 INTRODUCTION TO ARRAYS

An **array** is a data structure that stores **multiple values of the same type** under a single variable name.
Arrays allow fast data access, efficient storage, and mathematical operations.

In Python, arrays can be created in two ways:

1. Using the **array module** (built-in)
2. Using **NumPy arrays** (most widely used in programming, data science, ML)

According to syllabus (MCA Python), we study **array module arrays** + **NumPy arrays basics** for dimensions & operations.

3.18 ADVANTAGES OF ARRAYS

1. **Efficient Storage**
   Stores similar data types compactly.
2. **Faster Access**
   Accessing elements using index is very fast.
3. **Easy Traversal**
   Can iterate using loops easily.
4. **Supports Mathematical Operations**
   Addition, subtraction, multiplication, slicing.
5. **Better Performance than Lists for Numeric Data**

6. **Useful for matrices, tables, vectors in Data Science**

3.19 CREATING ARRAYS

Python's built-in **array module** is used to create arrays.

Importing array Module

```
from array import *
```

Syntax for Creating an Array

```
array(typecode, [elements])
```

Typecodes

Typecode	Data Type	Example
'i'	integer	array('i', [1,2,3])
'f'	float	array('f', [1.2, 3.4])
'd'	double	array('d', [2.34, 5.67])
'u'	Unicode char	array('u', ['a','b'])

3.19.1 Example: Creating an Integer Array

```
from array import *

a = array('i', [10, 20, 30, 40])
print(a)
```

3.19.2 Taking Array Input from User

```
from array import *

n = int(input("Enter size: "))
arr = array('i', [])

for i in range(n):
    x = int(input("Enter element: "))
    arr.append(x)

print("Array =", arr)
```

3.20 ACCESSING ARRAY ELEMENTS (INDEXING)

Array indexing starts from **0**.

Example:

```
a = array('i', [10, 20, 30])
print(a[0])   # 10
print(a[2])   # 30
```

3.21 MODIFYING ARRAY ELEMENTS

```
a = array('i', [10, 20, 30])
a[1] = 50
print(a)
```

Output: array('i', [10, 50, 30])

3.22 ARRAY METHODS

Python provides useful methods to modify arrays.

1. append()

Adds element at the end.

```
a.append(60)
```

2. insert()

Insert at a specific position.

```
a.insert(2, 25)
```

3. remove()

Remove first occurrence.

```
a.remove(20)
```

4. pop()

Remove element at index.

```
a.pop(1)
```

5. index()

Returns index of element.

a.index(30)

6. reverse()

Reverses the array.

a.reverse()

3.23 TYPES OF ARRAYS

Python supports:

1. Single-Dimensional Arrays

One row of elements.
Example:

a = array('i', [1,2,3,4,5])

2. Multi-Dimensional Arrays (NumPy)

Matrices → rows & columns
Examples:

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

3.24 MATHEMATICAL OPERATIONS ON ARRAYS

With **NumPy arrays**, mathematical operations become easy.

Example:

```
import numpy as np

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

print(a + b)
print(a * b)
print(a ** 2)
```

Output:
[5 7 9]

```
[4 10 18]
[1 4 9]
```

3.25 DIMENSIONS OF ARRAYS

NumPy arrays support multiple dimensions.

1D Array

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

2D Array

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

3D Array

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

3.26 ARRAY ATTRIBUTES (NumPy)

1. shape

Returns row × column size.

```
a.shape
```

2. ndim

Number of dimensions.

```
a.ndim
```

3. size

Total number of elements.

```
a.size
```

4. dtype

Data type of elements.

a.dtype

3.27 ARRAY SLICING

Selecting a portion of the array.

1D Slicing

```
a = np.array([10,20,30,40,50])
print(a[1:4])     # 20 30 40
```

2D Slicing

```
a = np.array([[1,2,3],[4,5,6],[7,8,9]])
print(a[0:2, 1:3])
```

Output:
[[2 3]
[5 6]]

3.28 REAL-TIME ARRAY APPLICATIONS

Arrays are used in:

- Data Science (tables, datasets)
- Scientific computing
- Machine Learning
- Image processing (images = 3D arrays)
- Banking / Billing systems
- Student marks management
- Matrix operations

UNIT – IV

COMPONENTS & FUNCTIONS

4.1 INTRODUCTION TO LISTS

A **List** in Python is an *ordered*, *mutable*, and *indexed* collection of items.
Lists are one of the most powerful and commonly used data structures in Python.

Lists can store:

- Numbers
- Strings
- Decimal values
- Boolean values
- Other lists
- Mixture of different data types

Lists use **square brackets [ ]**.

Example:

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

4.2 CHARACTERISTICS OF LISTS

1. **Ordered**
   Elements preserve insertion order.
2. **Mutable**
   Elements can be changed or modified.
3. **Indexed**
   Each element has an index starting from 0.
4. **Allow Duplicates**
   Example: [10, 20, 10]
5. **Allow Mixed Data Types**
   Example: [10, "Hari", 5.5, True]
6. **Dynamic Size**
   List grows/shrinks as needed.

4.3 CREATING LISTS

4.3.1 Empty List

list1 = []

4.3.2 List with Elements

numbers = [10, 20, 30]

4.3.3 Mixed Data Type List

data = [10, "Hari", 3.5, True]

4.3.4 Using list() Constructor

```
chars = list("python")
```

4.4 ACCESSING LIST ELEMENTS

Lists use indexing.

4.4.1 Positive Indexing

```
fruits = ["apple", "banana", "cherry"]
print(fruits[0])   # apple
print(fruits[2])   # cherry
```

4.4.2 Negative Indexing

```
print(fruits[-1])  # cherry
print(fruits[-2])  # banana
```

4.5 LIST SLICING

Used to retrieve a part of the list.

Syntax

```
list[start:end]
```

Example

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

4.6 UPDATING LIST ELEMENTS

Since lists are mutable, values can be changed.

4.6.1 Changing a Single Element

```
numbers = [10, 20, 30]
numbers[1] = 50
# [10, 50, 30]
```

4.6.2 Changing Multiple Elements

```
marks = [50, 60, 70, 80]
marks[1:3] = [65, 75]
# [50, 65, 75, 80]
```

4.7 LIST OPERATIONS

4.7.1 Concatenation

```
a = [1, 2]
b = [3, 4]
c = a + b    # [1, 2, 3, 4]
```

4.7.2 Repetition

```
x = [10]
print(x * 3)   # [10, 10, 10]
```

4.7.3 Membership

```
fruits = ["apple", "mango"]
"apple" in fruits      # True
"orange" not in fruits # True
```

4.8 IMPORTANT LIST METHODS

Python provides several built-in methods to work with lists.

4.8.1 append() → Add element at end

```
numbers.append(100)
```

4.8.2 insert() → Insert at position

```
numbers.insert(1, 55)
```

4.8.3 extend() → Add multiple items

```
numbers.extend([200, 300])
```

4.8.4 remove() → Remove first occurrence

```
numbers.remove(20)
```

4.8.5 pop() → Remove by index

```
numbers.pop(2)
```

4.8.6 index() → Get index of element

```
numbers.index(30)
```

4.8.7 count() → Count occurrences

numbers.count(10)

4.8.8 reverse() → Reverse list

numbers.reverse()

4.8.9 sort() → Ascending sort

numbers.sort()

4.8.10 sort(reverse=True) → Descending sort

numbers.sort(reverse=True)

4.9 NESTED LISTS

A list inside another list.

Example:

```
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
```

Accessing nested elements

```
print(matrix[1][2])   # 6
```

4.10 TRAVERSING A LIST

Using for loop

```
for item in fruits:
    print(item)
```

Using index

```
for i in range(len(fruits)):
    print(fruits[i])
```

4.11 LIST COMPREHENSIONS (Advanced)

(Very useful for short, powerful list creation)

Syntax

new_list = [expression for item in list]

Example

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

4.12 INTRODUCTION TO TUPLES

A **tuple** is an ordered, indexed collection of elements, similar to a list, but **immutable** (unchangeable).

Key features:

- Ordered
- Supports indexing
- Allows duplicates
- Allows mixed data types
- **Immutable** (cannot modify elements after creation)
- Faster than lists
- Memory-efficient

Tuples are written using **parentheses ( )**.

Example:

t = (10, 20, 30)

4.13 NEED FOR TUPLES

1. **Data protection**
   – Since tuples cannot be changed, they protect data from accidental modification.
2. **Performance**
   – Faster than lists in iteration.
3. **Hashable**
   – Can be used as keys in dictionaries (lists cannot).
4. **Used for fixed collections**
   – Coordinates, dates, database records.
5. **Less memory**
   – Ideal for large datasets.

4.14 CREATING TUPLES

4.14.1 Empty Tuple

```
t = ()
```

4.14.2 Tuple with Elements

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

4.14.3 Mixed Data Types

```
t = (10, "Hari", 3.5)
```

4.14.4 Without Parentheses (Tuple Packing)

```
t = 10, 20, 30
```

4.14.5 Single-element Tuple

Must add comma.

```
t = (10,)    # correct
t = (10)     # NOT a tuple
```

4.14.6 Using tuple() Constructor

```
t = tuple([10, 20, 30])
```

4.15 ACCESSING TUPLE ELEMENTS

4.15.1 Positive Indexing

```
t = (10,20,30)
print(t[0])  # 10
print(t[2])  # 30
```

4.15.2 Negative Indexing

```
print(t[-1])  # 30
print(t[-2])  # 20
```

4.16 SLICING TUPLES

Tuples support slicing just like lists.

Example:

```
t = (10,20,30,40,50)
print(t[1:4])   # (20,30,40)
```

4.17 TUPLE OPERATIONS

4.17.1 Concatenation

```
t1 = (1,2)
t2 = (3,4)
t3 = t1 + t2    # (1,2,3,4)
```

4.17.2 Repetition

```
t = (10,)
print(t * 3)   # (10,10,10)
```

4.17.3 Membership

```
10 in (10,20,30)    # True
```

4.18 BUILT-IN FUNCTIONS FOR TUPLES

len()

Returns length.

```
len((10,20,30))   # 3
```

max() / min()

```
max((10,20,30))   # 30
```

sum()

```
sum((1,2,3))      # 6
```

tuple()

Convert list to tuple.

```
tuple([10,20,30])
```

4.19 LOOPING THROUGH TUPLES

Using for loop:

```
for i in (10,20,30):
   print(i)
```

4.20 NESTED TUPLES

Tuples inside tuples.

Example:

```
t = (10, (20,30), 40)
print(t[1][1])   # 30
```

4.21 UNPACKING TUPLES

Assign tuple elements into variables.

```
a, b, c = (10,20,30)
print(a, b, c)
```

4.22 IMMUTABILITY OF TUPLES

Tuples **do not allow modification**.

Wrong:

```
t[1] = 100    # Error
```

To modify, convert to list:

```
t = (10,20,30)
lst = list(t)
lst[1] = 50
t = tuple(lst)
```

4.23 REAL-TIME USES OF TUPLES

- Storing database records
- Coordinates (x,y,z)
- Dates (day, month, year)
- Fixed values like configuration
- Returning multiple values from a function

4.24 INTRODUCTION TO DICTIONARIES

A **Dictionary** in Python is a powerful, flexible, and efficient data structure used to store data in the form of **key–value pairs**.

Example:

student = {"name": "Hari", "age": 25, "course": "MCA"}

Key Features

1. **Unordered** (in older Python versions; in Python 3.7+, preserves order)
2. **Mutable** (values can be changed)
3. **Keys must be unique**
4. **Values can be duplicated**
5. **Keys must be immutable** (int, string, tuple)
6. **Very fast lookups** using hashing

Dictionaries are ideal for storing data like:

- Student details
- Employee records
- Settings/configurations
- JSON-type data

4.25 CREATING DICTIONARIES

4.25.1 Empty Dictionary

d = {}

4.25.2 Using Key–Value Pairs

student = {"name": "Hari", "age": 20, "dept": "MCA"}

4.25.3 Using dict() Constructor

d = dict(name="Hari", age=20)

4.25.4 Using List of Tuples

d = dict([("name", "Hari"), ("age", 20)])

4.26 ACCESSING DICTIONARY VALUES

4.26.1 Using Key

student["name"]

4.26.2 Using get() Method

student.get("age")

Difference:
If key does not exist:

- student["abc"] → Error
- student.get("abc") → None

4.27 MODIFYING DICTIONARY VALUES

4.27.1 Changing Existing Value

student["age"] = 21

4.27.2 Adding a New Key–Value Pair

student["address"] = "Tirupati"

4.28 DELETING ELEMENTS FROM DICTIONARIES

del statement

del student["age"]

pop() → Remove by key

student.pop("dept")

popitem() → Remove last inserted item

student.popitem()

clear() → Remove all items

student.clear()

4.29 DICTIONARY OPERATIONS

4.29.1 Membership

"name" in student # checks key

4.29.2 Length

```
len(student)
```

4.29.3 Iterating Through Keys

```
for key in student:
    print(key)
```

4.29.4 Iterating Through Values

```
for value in student.values():
    print(value)
```

4.29.5 Iterating Through Key–Value Pairs

```
for key, value in student.items():
    print(key, value)
```

4.30 IMPORTANT DICTIONARY METHODS

keys() → Returns all keys

```
student.keys()
```

values() → Returns all values

```
student.values()
```

items() → Returns key–value pairs

```
student.items()
```

update() → Merge dictionaries

```
d1.update(d2)
```

setdefault() → Return value if key exists; else insert key

```
student.setdefault("branch", "CS")
```

4.31 COPYING DICTIONARIES

Using copy()

```
d2 = student.copy()
```

Using dict()

```
d3 = dict(student)
```

4.32 NESTED DICTIONARIES

A dictionary inside another dictionary.

```
student = {
   "name": "Hari",
   "marks": {"math": 90, "science": 88}
}
```

Accessing:

```
student["marks"]["science"]
```

4.33 PASSING DICTIONARIES TO FUNCTIONS

Dictionaries can be passed directly to functions.

Example:

```
def display(info):
   for k, v in info.items():
      print(k, ":", v)

student = {"name": "Hari", "age": 21}
display(student)
```

4.34 ORDERED DICTIONARIES (OrderedDict)

(From **collections** module)

Python's normal dictionaries (Python 3.7+) maintain insertion order,
but **OrderedDict** offers extra features, like:

- reorder items
- move elements
- compare order of dictionaries

Import

```
from collections import OrderedDict
```

Creating OrderedDict

```
d = OrderedDict()
```

```
d["a"] = 1
d["b"] = 2
d["c"] = 3
```

move_to_end()

```
d.move_to_end("a")
```

popitem(last=False) → Remove first inserted item

```
d.popitem(last=False)
```

4.35 REAL-TIME USES OF DICTIONARIES

1. Student database
2. Contact lists
3. JSON data
4. API response processing
5. Configuration files
6. Data analysis (mapping values)
7. Caches & lookup tables

4.36 INTRODUCTION TO FUNCTIONS

A **function** is a block of organized, reusable, and logically grouped statements that perform a specific task.

Functions help in:

1. Reducing code repetition
2. Improving readability
3. Dividing large programs into smaller modules
4. Debugging and testing easily
5. Reusing code in multiple programs

Python provides:

- Built-in functions (print(), len(), type(), etc.)
- User-defined functions (created by programmers)

4.37 DEFINING A FUNCTION

A user-defined function is created using the **def** keyword.

Syntax

```
def function_name(parameters):
    statements
```

Example

```
def greet():
    print("Welcome to Python Programming")
```

4.38 CALLING A FUNCTION

To execute a function, you must call it.

```
greet()
```

Output:
Welcome to Python Programming

4.39 FUNCTION PARAMETERS AND ARGUMENTS

Parameters → variables inside the function definition
Arguments → values passed during function call

Example

```
def add(a, b):      # parameters
    print(a + b)

add(10, 20)         # arguments
```

4.40 TYPES OF ARGUMENTS

Python supports several types of arguments.

4.40.1 Positional Arguments

Values passed in order.

```
def info(name, age):
    print(name, age)

info("Hari", 25)
```

4.40.2 Keyword Arguments

Arguments passed using parameter names.

info(age=25, name="Hari")

4.40.3 Default Arguments

Provide default values.

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

greet()
greet("Hari")
```

4.40.4 Variable-Length Arguments

Used when number of arguments is unknown.

**(a) args → multiple positional arguments*

```
def add(*nums):
    s = 0
    for n in nums:
        s += n
    print("Sum =", s)

add(1,2,3,4)
```

**(b) kwargs → multiple keyword arguments

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

info(name="Hari", age=25, branch="MCA")
```

4.41 RETURNING VALUES FROM FUNCTIONS

Functions can return one or more values using **return**.

Example (single return)

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

result = add(10, 20)
```

4.42 RETURNING MULTIPLE VALUES

Python allows returning multiple values as a tuple.

```
def calc(a, b):
    return a + b, a - b, a * b

x, y, z = calc(10, 5)
print(x, y, z)
```

4.43 PASS BY OBJECT REFERENCE

Python uses **Pass-by-Object-Reference** (neither pass-by-value nor pass-by-reference).

Meaning:

- If the object passed is **mutable** (list, dict) → changes made inside function affect original.
- If the object is **immutable** (int, float, string, tuple) → changes inside function DO NOT affect original.

4.43.1 Immutable Example (int)

```
def modify(x):
    x = x + 10

a = 5
modify(a)
print(a)     # 5 (unchanged)
```

4.43.2 Mutable Example (list)

```
def update(lst):
    lst.append(100)

numbers = [10,20]
update(numbers)
print(numbers)  # [10,20,100]
```

4.44 ANONYMOUS FUNCTIONS (LAMBDA)

A **lambda function** is a small, one-line function without a name.

Syntax

lambda arguments: expression

Example:

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

4.45 SCOPE OF VARIABLES

Two types:

4.45.1 Local Variables

Declared inside function; accessible only within function.

```
def test():
    x = 10
    print(x)
```

4.45.2 Global Variables

Declared outside; accessible everywhere.

```
x = 100

def show():
    print(x)
```

4.46 RECURSIVE FUNCTIONS

A function calling itself is **recursion**.

Example:

```
def fact(n):
    if n == 1:
        return 1
    return n * fact(n-1)

print(fact(5))   # 120
```

Used for:

- Factorial
- Fibonacci
- Tree traversal
- Searching algorithms

4.47 FUNCTION VS METHOD

Function	Method
Defined independently using def	Associated with an object
Called by name → func()	Called using object → object.method()
Example: len(list)	Example: list.append( )

4.48 REAL-TIME USES OF FUNCTIONS

1. Modular programming
2. Reusing repeated logic
3. Handling user inputs
4. Mathematical calculations
5. Validation and verification
6. File handling tasks

UNIT – V

OBJECT ORIENTED PROGRAMMING SYSTEM & DATA STRUCTURES

5.1 INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING (OOP)

Object-Oriented Programming (OOP) is a programming approach that organizes software design around **objects** rather than functions and logic.
An object represents **real-world entities** such as:

- Student
- Employee

- Car
- Bank Account
- Mobile

Each object has:

1. **Data (attributes)** – properties
2. **Functions (methods)** – actions performed by the object

Python is a *fully object-oriented programming language*, meaning:

- Everything is an object
- Supports inheritance, encapsulation, polymorphism, abstraction

OOP makes programs:

- Easy to understand
- Easy to maintain
- More secure
- More reusable

5.2 NEED FOR OOP

1. **Reduces complexity**
   Programs reflect real-world modeling.
2. **Reusability**
   Classes can be reused across applications.
3. **Data Security**
   Encapsulation protects data from unauthorized access.
4. **Improved maintainability**
   Changes can be made easily without disturbing other parts.
5. **Modularity**
   Large programs can be divided into small, manageable parts.
6. **Extensibility**
   New features can easily be added through inheritance.

5.3 BASIC OOP TERMINOLOGY

1. Class

A blueprint or template that defines properties and behaviors.

2. Object

A real instance created from a class.

3. Attribute

Variables that store object data.

4. Method

Function defined inside a class.

5. Constructor

Special method to initialize object values.

6. Inheritance

Child class acquiring features from parent class.

7. Polymorphism

Same operation behaving differently in different contexts.

8. Encapsulation

Protecting data using access control.

9. Abstraction

Showing essential details and hiding internal complexity.

5.4 CLASSES IN PYTHON

A **class** is used to define a user-defined data type.
It consists of:

- Class name
- Attributes (variables)
- Methods (functions)

Syntax

```
class ClassName:
   # attributes
   # methods
```

Example:

```
class Student:
   pass
```

5.5 OBJECTS IN PYTHON

An **object** is an instance of a class.

Creating an object:

```
s1 = Student()
```

Accessing attributes/methods:

```
objectname.attribute
objectname.method()
```

5.6 init() METHOD (CONSTRUCTOR)

__init__() is a **special method** called automatically when an object is created.
Used for initialization (assigning initial values).

Example:

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

s1 = Student("Hari", 22)
print(s1.name)
```

5.7 SELF PARAMETER

- Represents the **current object**
- Automatically passed to every method
- Used to access attributes and methods of the class

Example:

```
class Test:
   def show(self):
      print("This is:", self)

obj = Test()
obj.show()
```

5.8 ATTRIBUTES (INSTANCE & CLASS VARIABLES)

Python supports two types of attributes:

5.8.1 Instance Attributes

- Unique for every object
- Defined inside __init__() using self

Example:

```
class Employee:
    def __init__(self, id, name):
        self.id = id
        self.name = name
```

5.8.2 Class Attributes

- Common to all objects
- Defined inside class but outside methods

Example:

```
class Employee:
    company = "Infosys"    # class attribute
```

5.9 METHODS IN PYTHON

Python supports three types of methods:

5.9.1 Instance Methods

Work with object attributes.

```
class Student:
    def display(self):
        print("Instance method")
```

5.9.2 Class Methods

Work with class attributes.
Use @classmethod decorator.

```
class Student:
    college = "SVU"

    @classmethod
    def showCollege(cls):
        print(cls.college)
```

5.9.3 Static Methods

General utility functions.
Use @staticmethod decorator.

```
class Math:
    @staticmethod
    def square(x):
        return x*x
```

5.10 DELETING OBJECTS

Objects are deleted when:

- No reference exists
- Using del keyword
- Garbage collector removes it automatically

```
del s1
```

5.11 REAL-TIME EXAMPLE PROGRAM

Student Class

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

    def display(self):
        print("Roll:", self.roll)
        print("Name:", self.name)
        print("Branch:", self.branch)
```

```
s = Student(101, "Hari", "MCA")
s.display()
```

5.12 ENCAPSULATION

Encapsulation is the process of **binding data and methods together** in a single unit (class), and **protecting the data** from outside access.

In simple words:

Encapsulation = Data Hiding + Data Protection

Python supports encapsulation through:

1. **Public Members**
2. **Protected Members**
3. **Private Members**

These access levels are implemented using **naming conventions**.

5.12.1 PUBLIC MEMBERS

- Accessible anywhere (inside class, outside class, other modules).
- No underscore needed.

Example:

```
class Student:
    def __init__(self):
        self.name = "Hari"   # public

s = Student()
print(s.name)     # accessible
```

5.12.2 PROTECTED MEMBERS

- Identified by **single underscore** (_)
- Access allowed inside class and subclasses
- Outside access is allowed but **not recommended**

Example:

```
class Student:
    def __init__(self):
```

```
    self._marks = 85   # protected
```

5.12.3 PRIVATE MEMBERS

- Identified by **double underscore** (__)
- Cannot be accessed directly outside the class
- Only accessible inside class

Example:

```
class Student:
    def __init__(self):
        self.__password = "abc123"

s = Student()
print(s.__password)     # ERROR
```

To access private members indirectly:

```
class Student:
    def __init__(self):
        self.__password = "abc123"

    def show(self):
        print(self.__password)

s = Student()
s.show()
```

5.12.4 GETTER AND SETTER METHODS

Used to safely access and update private variables.

Getter → read value

Setter → modify value

```
class Bank:
    def __init__(self):
        self.__balance = 5000

    def getBalance(self):
        return self.__balance

    def setBalance(self, amount):
        if amount > 0:
```

```
        self.__balance = amount

b = Bank()
print(b.getBalance())
b.setBalance(8000)
print(b.getBalance())
```

5.12.5 BENEFITS OF ENCAPSULATION

1. **Data Protection** – prevents unauthorized access
2. **Improves Security**
3. **Easier Maintenance**
4. **Prevents accidental changes**
5. **Cleaner Program Structure**

5.13 ABSTRACTION

Abstraction means **showing only essential features** and **hiding internal details**.

Example from real world:

- Car → We use steering & pedals, but don't know the internal engine mechanics.

In programming:

- Show only necessary methods
- Hide complex internal logic
- Reduce program complexity

Python provides abstraction using:

1. **Abstract Classes**
2. **Interfaces** (not exactly, but achieved using abstract base classes)

5.13.1 ABSTRACT CLASSES

An abstract class is a class that contains **one or more abstract methods**.
It cannot be instantiated directly.

To define abstract classes, we use:

```
from abc import ABC, abstractmethod
```

5.13.2 Creating an Abstract Class

```
from abc import ABC, abstractmethod

class Shape(ABC):

    @abstractmethod
    def area(self):
        pass
```

Characteristics:

- Cannot create object of Shape
- Must override area() in child class

5.13.3 Implementing Abstract Methods in Child Class

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

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

c = Circle(5)
print(c.area())
```

5.13.4 NEED FOR ABSTRACTION

1. Reduce complexity
2. Hide unnecessary processes
3. Provide framework for child classes
4. Helps implement templates
5. Improves security
6. Allows different implementations for different subclasses

5.13.5 DIFFERENCE BETWEEN ABSTRACTION & ENCAPSULATION

Encapsulation	Abstraction
Hides data	Hides implementation details
Achieved using private members	Achieved using abstract classes
Focus on "how to protect data"	Focus on "how to simplify usage"

Encapsulation	Abstraction
Encapsulates (binds) data + methods	Provides outline for methods

5.13.6 REAL-TIME EXAMPLES OF ABSTRACTION

1. ATM interface
   - We insert card & enter PIN
   - Internal transaction logic is hidden
2. Mobile Camera
   - Press button → takes photo
   - Processing is hidden
3. Car driving
   - User sees only steering, brake

Engine mechanism is hidden

5.14 INHERITANCE

Inheritance is an OOP concept where a child class (subclass) acquires the properties and methods of a parent class (superclass).

Why Inheritance?

1. **Reusability**
   – Write once, use multiple times.
2. **Extensibility**
   – Add new features to existing classes without modifying original code.
3. **Fewer errors**
   – Common logic written only once.
4. **Better organization**
   – Programs become modular and structured.

5.14.1 TERMINOLOGY

- **Parent Class / Base Class / Super Class**
  Class whose properties are inherited.
- **Child Class / Derived Class / Sub Class**
  Class which inherits.

Syntax

```
class Parent:
   ...
```

```
class Child(Parent):
    ...
```

5.15 TYPES OF INHERITANCE

Python supports **five** types of inheritance:

1. **Single Inheritance**
2. **Multiple Inheritance**
3. **Multilevel Inheritance**
4. **Hierarchical Inheritance**
5. **Hybrid Inheritance**

Each one explained below.

5.15.1 SINGLE INHERITANCE

One parent → one child.

```
class A:
    def showA(self):
        print("Class A")

class B(A):
    def showB(self):
        print("Class B")

obj = B()
obj.showA()
obj.showB()
```

5.15.2 MULTIPLE INHERITANCE

Child class inheriting from **multiple parents**.

```
class A:
    def showA(self):
        print("A")

class B:
    def showB(self):
        print("B")

class C(A, B):
```

```
    pass

obj = C()
obj.showA()
obj.showB()
```

5.15.3 MULTILEVEL INHERITANCE

Parent → Child → Grandchild

```
class A:
    def showA(self):
        print("A")

class B(A):
    def showB(self):
        print("B")

class C(B):
    def showC(self):
        print("C")

obj = C()
obj.showA()
obj.showB()
obj.showC()
```

5.15.4 HIERARCHICAL INHERITANCE

Single parent → multiple children.

```
class A:
    def showA(self):
        print("A")

class B(A):
    pass

class C(A):
    pass
```

5.15.5 HYBRID INHERITANCE

Combination of multiple types.

Example:

A → B, C → D

Python supports hybrid inheritance but resolved using **MRO (Method Resolution Order)**.

5.16 CONSTRUCTOR IN INHERITANCE

Parent class constructor does not automatically run unless called using:

1. Parent class name

```
class A:
    def __init__(self):
        print("A constructor")

class B(A):
    def __init__(self):
        A.__init__(self)
        print("B constructor")
```

2. super() method (Recommended)

```
class A:
    def __init__(self):
        print("A constructor")

class B(A):
    def __init__(self):
        super().__init__()
        print("B constructor")
```

5.17 METHOD OVERRIDING

Overriding occurs when **child class provides its own version** of a parent class method.

Example:

```
class A:
    def show(self):
        print("Parent")

class B(A):
    def show(self):
        print("Child")
```

```
obj = B()
obj.show()     # Child
```

Purpose:

- Modify existing behavior
- Add new features

5.18 METHOD RESOLUTION ORDER (MRO)

MRO decides **which method to execute first** in multiple inheritance.

Use:

ClassName.mro()

5.19 POLYMORPHISM

Polymorphism = **One name, many forms**

Python supports:

1. **Overriding** (Runtime Polymorphism)
2. **Overloading (Conceptually)** – Python does not support overloading traditionally, but supports it using:
    - default arguments
    - variable-length arguments

5.19.1 POLYMORPHISM WITH FUNCTIONS

Same function works with different types.

```
print(len("Hari"))   # 4
print(len([10,20]))  # 2
```

5.19.2 POLYMORPHISM WITH CLASSES

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

class Dog:
```

```
    def sound(self):
        print("Bark")

for animal in (Cat(), Dog()):
    animal.sound()
```

5.19.3 METHOD OVERLOADING (CONCEPT IN PYTHON)

Python does not support classical method overloading.

But can achieve using default/variable arguments:

```
class Math:
    def add(self, a=None, b=None, c=None):
        if a!=None and b!=None and c!=None:
            return a+b+c
        elif a!=None and b!=None:
            return a+b
        else:
            return a
```

5.19.4 OPERATOR OVERLOADING

Python allows defining behavior of operators for custom classes using **dunder** methods.

Example: **add()**

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

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

obj1 = A(10)
obj2 = A(20)

print(obj1 + obj2)
```

Interfaces in Python

Introduction

Python does not have a built-in `interface` keyword. Interfaces in Python are implemented using **Abstract Base Classes (ABC)**. An interface defines method declarations, and the implementation is provided by child classes.

Abstract Base Classes (ABC)

Abstract Base Classes are used to define interfaces in Python. The `abc` module provides tools to create abstract base classes.

Need for Interfaces in Python

Interfaces are used to:

- Enforce method implementation
- Achieve abstraction
- Support polymorphism
- Maintain standard structure
- Improve code maintainability

Characteristics of Python Interfaces

- Defined using Abstract Base Classes
- Contain only abstract methods
- Interface objects cannot be created
- Child classes must implement all methods
- Used to define common behavior

Creating an Interface in Python

```
from abc import ABC, abstractmethod

class Shape(ABC):

    @abstractmethod
    def area(self):
        pass
```

Implementing an Interface in Python

```
class Rectangle(Shape):

    def area(self):
        print("Area of Rectangle")
```

Interface Implemented by Multiple Classes

```
class Circle(Shape):

    def area(self):
        print("Area of Circle")
```

Interface and Polymorphism in Python

```
s1 = Rectangle()
s2 = Circle()

s1.area()
s2.area()
```

Multiple Interfaces in Python

```
from abc import ABC, abstractmethod

class Printer(ABC):

    @abstractmethod
    def print_data(self):
        pass

class Scanner(ABC):

    @abstractmethod
    def scan_data(self):
        pass

class AllInOne(Printer, Scanner):

    def print_data(self):
        print("Printing data")

    def scan_data(self):
        print("Scanning data")
```

Advantages of Interfaces in Python

- Supports multiple inheritance
- Provides abstraction
- Improves code flexibility
- Encourages modular programming
- Ensures method consistency

Limitations of Interfaces in Python

- Cannot create objects of interface classes
- All abstract methods must be implemented
- Slightly complex syntax

Real-Time Examples

- Payment Gateway Systems
- Database Connectivity
- Shape-based Applications
- Device Driver Systems

Conclusion

In Python, interfaces are implemented using **Abstract Base Classes (ABC)**. They help in achieving abstraction, polymorphism, and multiple inheritance.

Exceptions in Python

Introduction

An **exception** is an error that occurs during the execution of a program. When Python detects an error, it stops the program and displays an error message. These runtime errors are known as **exceptions**.

Exception handling allows programmers to manage errors smoothly without stopping the program.

Need for Exception Handling

- Prevents program crash
- Provides meaningful error messages
- Helps continue program execution
- Ensures safe handling of unexpected situations

Common Built-in Exceptions

Exception Name	Description
ZeroDivisionError	Occurs when dividing a number by zero
ValueError	Occurs when converting invalid data type
NameError	Occurs when a variable is not defined
TypeError	Occurs when performing an operation on incompatible types
IndexError	Occurs when accessing an invalid list index

Exception Name	Description
`FileNotFoundError`	Occurs when trying to open a non-existing file

Basic Exception Handling Syntax

```
try:
    # Code that may cause an exception
except:
    # Code to handle the exception
```

Example: Basic Handling

```
try:
    a = int(input("Enter a number: "))
    b = int(input("Enter another number: "))
    print(a / b)
except:
    print("An error occurred.")
```

Handling Specific Exceptions

```
try:
    x = 10 / 0
except ZeroDivisionError:
    print("You cannot divide by zero.")
```

Multiple Except Blocks

```
try:
    num = int(input("Enter a number: "))
    print(10 / num)
except ZeroDivisionError:
    print("Cannot divide by zero.")
except ValueError:
    print("Invalid input. Enter only numbers.")
```

Else Block

The `else` block executes **only if no exception occurs**.

```
try:
    x = int(input("Enter number: "))
except ValueError:
    print("Invalid input")
else:
    print("Valid number:", x)
```

Finally Block

The `finally` block **always executes**, whether an exception occurs or not.

```
try:
    f = open("data.txt")
except FileNotFoundError:
    print("File not found.")
finally:
    print("Execution completed.")
```

User-Defined Exceptions

Python allows creating custom exceptions.

```
class AgeError(Exception):
    pass

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

try:
    if age < 18:
        raise AgeError("Age must be 18 or above.")
    else:
        print("You are eligible.")
except AgeError as e:
    print(e)
```

5.20 INTRODUCTION TO DATA STRUCTURES

A **Data Structure** is a way of organizing and storing data efficiently so that it can be accessed and modified easily.

Common Data Structures:

- Arrays
- Linked Lists
- Stacks
- Queues
- Deques

In this part, we study **Linked List, Stack, Queue, Deque** in detail.

5.21 LINKED LIST

A **Linked List** is a linear data structure where elements (called **nodes**) are connected using **pointers**.

Each node contains:

1. **Data**
2. **Pointer/Reference** to next node

Example representation:

[Data|Next] → [Data|Next] → [Data|Next] → None

The **first node** is called **head**.

5.21.1 TYPES OF LINKED LIST

1. **Singly Linked List**
2. **Doubly Linked List**
3. **Circular Linked List**

Syllabus mainly covers **singly linked list** (conceptual + implementation).

5.21.2 ADVANTAGES OF LINKED LIST

- Dynamic size (grows and shrinks at runtime)
- Efficient insertion and deletion
- No memory wasted
- Useful for queues, stacks, graphs

5.21.3 DISADVANTAGES OF LINKED LIST

- Slow access (no direct indexing)
- Requires extra memory for pointers
- Reverse traversal difficult (in singly list)

5.21.4 IMPLEMENTATION OF SINGLY LINKED LIST

A linked list node is created using a class.

Node Class

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Linked List Class

```
class LinkedList:
```

```
def __init__(self):
    self.head = None
```

5.21.5 INSERTION OPERATIONS

Insert at Beginning

```
def insert_begin(self, data):
    new = Node(data)
    new.next = self.head
    self.head = new
```

Insert at End

```
def insert_end(self, data):
    new = Node(data)
    if self.head is None:
        self.head = new
        return
    temp = self.head
    while temp.next:
        temp = temp.next
    temp.next = new
```

5.21.6 TRAVERSAL OF LINKED LIST

```
def display(self):
    temp = self.head
    while temp:
        print(temp.data, end=" → ")
        temp = temp.next
    print("None")
```

5.22 STACK

A **Stack** is a linear data structure that follows the **LIFO (Last In First Out)** principle.

Common examples:

- Undo/Redo
- Browser back button
- Function call stack

5.22.1 STACK OPERATIONS

1. **push(x)** → Insert element
2. **pop()** → Remove top element
3. **peek()** → View top element
4. **isEmpty()** → Check empty
5. **size()** → Count elements

5.22.2 STACK IMPLEMENTATION USING LIST

Push

```
stack.append(10)
```

Pop

```
stack.pop()
```

Peek

```
stack[-1]
```

5.22.3 STACK IMPLEMENTATION USING CLASS

```
class Stack:
    def __init__(self):
        self.stack = []

    def push(self, x):
        self.stack.append(x)

    def pop(self):
        return self.stack.pop()

    def peek(self):
        return self.stack[-1]

    def isEmpty(self):
        return self.stack == []
```

5.23 QUEUE

A **Queue** is a linear data structure that follows **FIFO (First In First Out)**.

Examples:

- Ticket booking
- CPU scheduling
- Waiting lines

5.23.1 QUEUE OPERATIONS

1. **enqueue(x)** → Insert
2. **dequeue()** → Remove
3. **front()** → First element
4. **rear()** → Last element
5. **isEmpty()**

5.23.2 QUEUE IMPLEMENTATION USING LIST

```
queue.append(10)     # enqueue
queue.pop(0)         # dequeue
```

5.23.3 QUEUE USING collections.deque

Efficient implementation:

```
from collections import deque

q = deque()

q.append(10)
q.append(20)
q.popleft()
```

5.24 DEQUE (DOUBLE ENDED QUEUE)

Deque = **Queue that allows insertion & deletion at both ends**.

It is provided by:

```
from collections import deque
```

5.24.1 DEQUE OPERATIONS

Insert at Right

dq.append(10)

Insert at Left

dq.appendleft(20)

Remove from Right

dq.pop()

Remove from Left

dq.popleft()

5.24.2 Why Deque?

- Faster than list
- Supports both stack and queue operations
- Used in BFS algorithms
- Sliding window problems
- Task schedulers

5.25 DIFFERENCES TABLE

Stack vs Queue

Stack	Queue
LIFO	FIFO
push/pop	enqueue/dequeue
Top access	Front access

Queue vs Deque

Queue	Deque
One end insertion, one end deletion	Both ends insertion & deletion

5.26 INTRODUCTION TO DATE & TIME MODULE

Python provides a powerful **datetime** module to work with:

- Current date
- Current time
- Date & time together
- Differences between dates
- Formatting output
- Converting strings to dates

- Performing arithmetic on dates

We must import:

import datetime

The datetime module contains several useful classes:

1. **date**
2. **time**
3. **datetime**
4. **timedelta**

All are covered below in detail (as per MCA syllabus).

5.27 date CLASS

Creating a date object

```
import datetime

d = datetime.date(2025, 1, 24)
print(d)
```

Attributes

```
d.year
d.month
d.day
```

5.27.1 Getting Today's Date

```
today = datetime.date.today()
print(today)
```

5.27.2 Getting Current Local Date Components

```
d = datetime.date.today()

print(d.day)
print(d.month)
print(d.year)
```

5.27.3 Useful date Methods

Method	Purpose
today()	Current date
fromordinal()	Convert number to date
isoformat()	Convert to YYYY-MM-DD format
weekday()	0=Monday, 6=Sunday
isoweekday()	1=Monday, 7=Sunday

Example

```
date = datetime.date.today()
print(date.weekday())     # e.g. 6 = Sunday
print(date.isoformat())   # YYYY-MM-DD
```

5.28 time CLASS

Used to represent a time value (hour, minute, second, microsecond).

Creating time objects

```
t = datetime.time(10, 45, 30)
print(t)
```

Attributes

```
t.hour
t.minute
t.second
t.microsecond
```

5.29 datetime CLASS

Combines **date + time** into one object.

Creating datetime object

```
dt = datetime.datetime(2025, 1, 24, 14, 5, 30)
print(dt)
```

Current Date & Time

```
now = datetime.datetime.now()
print(now)
```

5.29.1 Extracting Components

now.year
now.month
now.day
now.hour
now.minute
now.second

5.29.2 Formatting datetime (strftime)

strftime() converts datetime to a formatted string.

Common Format Codes

Code	Meaning
%Y	Year (2025)
%y	Year (25)
%m	Month (01–12)
%d	Day (01–31)
%H	Hour (00–23)
%M	Minutes
%S	Seconds
%a	Weekday short
%A	Weekday full
%b	Month short
%B	Month full

Example

```
now = datetime.datetime.now()
print(now.strftime("%d-%m-%Y"))
print(now.strftime("%A"))
print(now.strftime("%H:%M:%S"))
```

5.29.3 Converting String to datetime (strptime)

```
s = "25-08-2024"
d = datetime.datetime.strptime(s, "%d-%m-%Y")
print(d)
```

5.30 timedelta CLASS (Date Arithmetic)

Used for adding or subtracting dates.

Example: Add 10 days

```
today = datetime.date.today()
future = today + datetime.timedelta(days=10)
print(future)
```

Subtract dates

```
d1 = datetime.date(2025, 5, 25)
d2 = datetime.date(2025, 5, 1)
delta = d1 - d2
print(delta.days)
```

5.30.1 Creating Your Own Time Differences

```
td = datetime.timedelta(days=2, hours=5)
```

5.31 date vs datetime

Feature	date	datetime
Stores	only date	date + time
Methods	fewer	many
Usage	birth date, deadlines	timestamps, logs

5.32 REAL-TIME EXAMPLES

1. Attendance system
2. Logging time stamps in servers
3. Expiry date calculation
4. Billing system & due dates
5. Reminders & schedulers
6. Banking interest calculation
7. File creation/modification tracking

5.33 INTRODUCTION TO DATABASE CONNECTIVITY

Database connectivity allows Python programs to:

- Connect to a database
- Send SQL queries
- Insert, update, delete records
- Retrieve data
- Close connections safely

Python supports many databases:

- SQLite
- MySQL
- PostgreSQL
- Oracle
- MongoDB

But mainly includes:

1. **SQLite** (default Python database)
2. **MySQL connectivity** (using connector module)

5.34 SQLITE DATABASE CONNECTIVITY

SQLite is a lightweight, file-based relational database built into Python.

You don't need to install anything — Python already includes **sqlite3 module**.

5.34.1 Import sqlite3 Module

```
import sqlite3
```

5.34.2 Connecting to Database

Syntax:

```
connection = sqlite3.connect("mydb.db")
```

- If file exists → opens it
- If not → creates new database file

5.34.3 Creating a Cursor

Cursor is used to execute SQL commands.

```
cursor = connection.cursor()
```

5.34.4 Creating a Table

```
cursor.execute("""
CREATE TABLE IF NOT EXISTS students(
   roll INTEGER PRIMARY KEY,
   name TEXT,
   marks REAL
)
""")
```

5.34.5 Inserting Records

```
cursor.execute("INSERT INTO students VALUES (101, 'Hari', 85.5)")
connection.commit()
```

Use commit() to save changes.

5.34.6 Multiple Inserts

```
data = [(102, 'Kiran', 90.2),
       (103, 'Ravi', 78.5)]

cursor.executemany("INSERT INTO students VALUES (?, ?, ?)", data)
connection.commit()
```

5.34.7 Retrieving Records (SELECT)

```
cursor.execute("SELECT * FROM students")

rows = cursor.fetchall()

for row in rows:
   print(row)
```

5.34.8 Updating Records

```
cursor.execute("UPDATE students SET marks = 95 WHERE roll = 101")
connection.commit()
```

5.34.9 Deleting Records

```
cursor.execute("DELETE FROM students WHERE roll = 103")
connection.commit()
```

5.34.10 Closing Connection

```
connection.close()
```

5.35 MYSQL DATABASE CONNECTIVITY

MySQL is a popular relational database used for large applications.

Python connects using module:

```
pip install mysql-connector-python
```

5.35.1 Import MySQL Module

```
import mysql.connector
```

5.35.2 Connecting to MySQL Server

```
db = mysql.connector.connect(
    host="localhost",
    user="root",
    password="yourpassword",
    database="college"
)
```

5.35.3 Creating Cursor

```
cur = db.cursor()
```

5.35.4 Creating a Table

```
cur.execute("""
CREATE TABLE students(
    roll INT PRIMARY KEY,
    name VARCHAR(30),
    marks FLOAT
)
""")
```

5.35.5 Inserting Records

```
cur.execute("INSERT INTO students VALUES (1, 'Hari', 88)")
db.commit()
```

5.35.6 Selecting Records

```
cur.execute("SELECT * FROM students")

for row in cur:
    print(row)
```

5.35.7 Updating Records

```
cur.execute("UPDATE students SET marks = 90 WHERE roll = 1")
db.commit()
```

5.35.8 Deleting Records

```
cur.execute("DELETE FROM students WHERE roll = 1")
db.commit()
```

5.36 EXCEPTION HANDLING IN DATABASE

Database errors must be handled carefully.

Example:

```
try:
    cur.execute("INSERT INTO students VALUES (1, 'Hari', 85)")
    db.commit()
except Exception as e:
    print("Error:", e)
    db.rollback()
```

5.37 PARAMETERIZED QUERIES (Protection from SQL Injection)

```
query = "INSERT INTO students VALUES (%s, %s, %s)"
data = (2, "Kiran", 92)

cur.execute(query, data)
db.commit()
```

5.38 DATABASE OPERATIONS SUMMARY (CRUD)

CRUD stands for:

- **C – Create (Insert)**
- **R – Read (Select)**
- **U – Update**
- **D – Delete**

SQLite & MySQL both follow same CRUD pattern.

5.39 REAL-TIME USES OF DATABASE CONNECTIVITY

1. Student management system
2. Library management system
3. Banking transactions
4. Online shopping apps
5. Inventory management
6. Hospital record systems
7. College administrative applications