PYTHON SKILL BUILDER SERIES 1

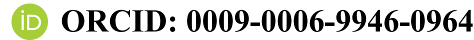
Debmalya Mukherjee, Ranjan Banerjee, Avijit Kumar Chaudhuri, Pranab Gharai, Neha Debnath

Python Skill Builder Series 1

Debmalya Mukherjee

Assistant Professor, Brainware University, Kolkata- 125, India

Email id: dbml.mukherjee@gmail.com


ORCID: 0009-0006-9946-0964

Ranjan Banerjee

Assistant Professor, Brainware University, Kolkata- 125, India

Email ID: rbkpccst@gmail.com

ORCID : 0009-0003-1950-7530

Avijit Kumar Chaudhuri

H.O.D. of Dept. of CSE, Brainware University, Kolkata- 125, India

Email ID: c.avijit@gmail.com

ORCID : 0000-0002-5310-3180

Pranab Gharai

Assistant Professor, Brainware University, Kolkata- 125, India

Email ID: pranab.g10@gmail.com

ORCID ID: 0009-0004-3602-5223

Neha Debnath

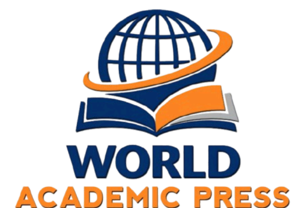
Brainware University, Kolkata- 125, India

ORCID- 0009-0004-6896-7290

Published By

World Academic Press, Kolkata-700126, India

www.worldacademic.press



© *2026 by* **Debmalya Mukherjee, Ranjan Banerjee, Avijit Kumar Chaudhuri, Pranab Gharai, Neha Debnath**

Published by:

World Academic Press , Kolkata, India

www.worldacademic.press

DOI: https://www.doi.org/10.66727/wap.9788199835306

License: *This work is licensed under the Creative Commons Attribution 4.0 International License (CC BY 4.0). To view a copy of this license, visit https://creativecommons.org/licenses/by/4.0/*

This book is the result of time, care, and thoughtful effort. It is meant to be read, reflected upon, and utilized to advance knowledge in the field. Under the CC BY 4.0 license, you are free to share and adapt this material for any purpose, provided appropriate credit is given to the authors.

Disclaimer: *Every effort has been made by the authors and publisher to present information that is accurate, reliable, and responsibly researched. This work is offered in good faith, with the hope that it informs, inspires, and invites thoughtful engagement.*

ISBN: 978-81-998353-1-3 (Paperback)

ISBN: 978-81-998353-0-6 (E-book)

First Edition: 2026

Table of Contents

Abstract 10

Chapter 1: Python Fundamentals 11

Introduction to Python 11

- Python and its popularity 11
- List of key features of Python 11
- Real-world applications of Python 11

Setup and Environment 12

- Steps to install Python on Windows 12
- Concept of an interpreter in Python 12
- Comments in python and the way they are written 12

Syntax and Execution 13

- Common syntax rules in Python 13
- Role of colons (:) in Python syntax 13
- Different ways to run Python code 13
- Common syntax rules in Python 14
- Method of checking the installed version of Python 14
- Dynamic typing in Python 14

MCQ’s 15

Chapter 2: Variables and Data Types 18

Variables and Typing Constraints 18

- Concept of a variable in python 18
- Rules for naming variables in Python 18
- Dynamic typing in Python and the difference of dynamic typing from static typing 18
- Type inference and its use in python 19

Core Built-in Data Types 19

- Python's basic built-in data types 19
- Difference between int, float, and complex in Python 19
- Bool type in Python and its behavior with other types 20
- Output of bool(0), bool(''), and bool([]) in Python 20
- None in Python, and its difference from 0, '', or False 20

Type Verification and Mutability 21

- Checking the data type of a variable in Python 21
- Checking if a variable is None and why we do not use == None 21
- The difference between mutable and immutable data types in Python (with examples) 22

Type Conversion Methods 22
Type casting in Python and its examples 22
Difference between implicit and explicit type conversion 22
String-to-number conversion in Python and the behavior when the string is invalid 23
MCQ's 23
Chapter 3: Working with Numbers and Strings 27
Mathematical Operations 27
Basic arithmetic operators in Python 27
Difference between / and // in Python 27
Purpose of the math module in Python 28
Method of rounding numbers in python 28
Main difference between int() and float() 29
String Fundamentals and Slicing 29
Creating strings in Python 29
Accessing individual characters or slices in a string 30
String slicing and its working process 31
Checking if strings are mutable or not in Python 31
String Operations and Formatting 31
Common string methods in Python 32
String concatenation in Python and method of joining strings 32
Formatting strings in Python 33
F-strings and their preference 33
Advanced String Concepts 34
String interning in Python 34
Differences between is and == for strings 34
MCQ's 35
Chapter 4: Control Flow and Decision Making 38
Conditional Branching 38
Purpose of if, elif, and else in Python 38
Multiple if vs elif 39
Simple program using nested if 39
Good practice for nested conditionals 40
One-line conditional (ternary operator) 40
Logic Evaluation 41
Logical operators in Python 41
Logical operators to simplify conditions 41
Chained comparisons (a < b < c) 42
Truthy and Falsy values 43

Checking if a variable exists 44
MCQ's 45
Chapter 5: Iteration in Python 48
Loop Foundations 48
Two main types of loops in Python 48
The for Loop Mechanics 48
For loop working process 48
Purpose of range() function 48
Reversing a loop using range() 49
Looping through a string 49
Difference between while and for loop 49
Advanced Iteration Strategies 49
Nested loops 50
Iterating index and value together 50
Infinite while loops 50
Else with loops 50
Loop Control and Generators 51
Loop control statements 51
Difference between break and continue 51
Iterators 51
Comprehensions 52
List comprehension 52
Reason that list comprehension is better 52
MCQ's 52
Chapter 6: Collections and Data Structures 56
Sequential Data: Lists and Tuples 56
Difference between list and tuple in Python 56
Creating a list and accessing elements 56
Common list methods 56
Advantages of tuples over lists 57
Unordered Data: Sets 57
Purpose of sets in Python 57
Set operations 58
Handling duplicates in sets 58
Key-Value Mapping: Dictionaries 58
Dictionaries in Python 58
Accessing and modifying dictionary values 59
Useful dictionary methods 59
Iterating through dictionary 59

Are dictionary keys mutable? 59
Checking key in dictionary 60
MCQ's 60
Chapter 7: Booleans and Operators 64
Boolean Logic 64
Boolean values in Python 64
Empty containers in Boolean context 64
Forcing Boolean evaluation 66
MCQ's 67
Chapter 8: Advanced Python Concepts 70
Object-Oriented Programming (OOP) 70
Inheritance in Python 70
Types of inheritance in Python 70
Use of super() function 71
Polymorphism in Python 71
Method overloading vs overriding 72
Encapsulation in Python 72
Advanced Iteration and Modularity 72
Iterators in Python 72
Creating a custom iterator 73
Variable scope in Python 73
Use of global keyword 74
Modules and Architecture 74
What is a Python module? 74
Built-in modules in Python 74
Use of if __name__ == "__main__": 75
Importing modules 75
MCQ's 75
Chapter 9: Working with Data and Dates 79
Date and Time Processing 79
Datetime module's uses 79
Getting current date and time 79
Formatting a date 79
Converting string into date 80
Advanced Mathematics 80
Math.floor() uses 80
Square root using Python 80
Difference between ceil() and floor() 81
Data Serialization and Parsing 81

Dictionary to JSON conversion 81
JSON string to Python object 81
Use of re module 82
Checking numbers in string using regex 82
Meaning of \d{3} in regex 82
Package Management 82
PIP in Python 82
Installing packages using PIP 83
Listing installed packages 83
MCQ's 83
Chapter 10: Error Handling and Input 86
Exception Handling Mechanisms 86
Importance of error handling in Python 86
Exception handling in Python 87
Basic try-except syntax 87
Multiple except blocks 88
Catching all exceptions 88
Use of else in exception handling 88
Specific Errors and Validation 89
Raise keyword 89
Division by zero error 89
ValueError in Python 89
Difference between TypeError and ValueError 90
User Interaction 90
Taking input from user 90
Converting input safely to number 90
String formatting methods 90
MCQ's 91
Chapter 11: Virtual Environments and File Handling 94
Environment Setup 94
Virtual environment in Python and its use 94
Creating and activating virtual environment 95
Use of requirements.txt 95
Importance of file handling 96
File opening modes in Python 96
Safe file handling using with 96
File Operations 97
Reading a file 97
Difference between read(), readline(), readlines() 97

Writing to a file in Python 97
Appending data to file 97
Difference between w and a 98
Reading and writing together (r+) 98
File System Management 98
Checking if file exists 98
Deleting a file 99
FileNotFoundError handling 99
MCQ’s 99
References 106

Abstract

This book presents a structured and simplified collection of frequently asked Python programming questions along with concise and easy-to-understand answers. The primary objective of this book is to strengthen fundamental programming knowledge, improve logical thinking, and assist learners in preparing effectively for technical interviews and academic examinations.

The content is systematically organized into eleven chapters covering essential Python topics, beginning with basic concepts such as Python fundamentals, variables, and data types, and gradually progressing toward intermediate and advanced topics. The book includes detailed explanations of control flow statements, iteration techniques, collections, operators, object-oriented programming concepts, modules, exception handling, and file operations. It also introduces practical topics such as virtual environments, JSON handling, regular expressions, and working with dates and external packages.

Each chapter is designed to promote conceptual clarity through short explanations, illustrative examples, and multiple-choice questions that reinforce learning and test understanding. The simplified language used throughout the book makes it suitable for beginners, students, and job aspirants who are building foundational skills in Python programming.

Overall, this book serves as a practical learning resource for understanding core Python concepts, improving coding confidence, and preparing for technical interviews in software development and related fields.

Keywords: Python for beginners; Python programming; Python fundamental; Technical interview preparation; Python Q&A

Chapter 1: Python Fundamentals

Introduction to Python

Python and its popularity

Python is a high-level programming language that is interpreted, meaning the code runs line by line instead of being compiled all at once. It is widely appreciated because its syntax is simple and easy to read, making it beginner-friendly. Python supports different programming styles such as procedural, object-oriented, and functional programming. Its widespread use is mainly due to its rich standard library, strong developer community, and its usefulness in many fields like web development, data science, artificial intelligence, automation, and several other areas.

List of key features of Python

Python offers several useful features that make it a popular programming language among developers and beginners alike. Some of its important features include:

- Simple and easy-to-understand syntax, which makes learning faster
- Works as an interpreted language, so code runs line by line
- Supports dynamic typing, meaning variable types do not need to be declared manually
- Includes a rich standard library with many built-in tools
- Can run on different operating systems like Windows, Linux, and macOS
- Provides access to a large number of third-party libraries and packages
- Supports both object-oriented and functional programming approaches

Real-world applications of Python

Python is used in many practical fields because it is flexible and easy to work with. Some common real-world uses of Python include:

- Building websites and web applications using frameworks such as Django and Flask
- Performing data analysis and handling large datasets with tools like Pandas and NumPy
- Developing machine learning and artificial intelligence systems using libraries such as TensorFlow and scikit-learn
- Automating routine tasks and writing scripts to save time
- Creating games and entertainment software
- Designing desktop-based applications
- Developing networking-related programs and tools

Setup and Environment

Steps to install Python on Windows

The process of installing Python on a Windows system can be completed by following the steps given below:

1. Visit the official Python website (**python.org**) and download the latest version of Python.
2. After downloading, open the installer file and make sure to select the option **"Add Python to PATH."**
3. Click on the **Install Now** button to begin the installation.
4. Once the installation is finished, open Command Prompt (CMD) and type **python --version** to confirm that Python has been installed correctly.

Concept of an interpreter in Python

In Python, an interpreter is a program that reads the code step by step and executes each line one at a time. Unlike a compiler, which translates the entire program before running it, the interpreter processes instructions individually as they are encountered. The standard interpreter commonly used with Python is known as **CPython** (Zelle, 2016).

The following Python program can be written and run by following these steps.

You can write and run a basic Python program by following the steps given below:

Code:

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

To run:
First, save the program in a file named hello.py. After saving the file, open the command prompt and run it using the command:

```
python hello.py
```

Comments in python and the way they are written

Comments in Python are used to explain what the code does, making it easier for others (and yourself) to understand the program. These comments are ignored by the interpreter and do not affect how the program runs.

There are two common ways to write comments in Python:

• **Single-line comments:** These are written using the # symbol at the beginning of the line.

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

• **Multi-line comments:** These are usually written using triple quotes, which are also known as docstrings.

```
"""
This is an example
of a multi-line comment
"""
```

Syntax and Execution

Common syntax rules in Python

Docstrings are special strings in Python that are used to provide explanations or documentation for functions, classes, or modules. They help describe what a particular part of the code does and can be accessed later using the **__doc__** attribute.

The main difference between docstrings and normal comments is that regular comments are only meant for reading and are ignored completely by the program, while docstrings are stored as part of the program and can be retrieved whenever needed.

Role of colons (:) in Python syntax

In Python, the colon (:) symbol is used to indicate the beginning of a block of code. It is placed at the end of certain statements to show that the following lines belong to that specific block.

Colons are commonly used with statements such as **if**, **for**, **while**, **def**, and **class**, where a group of related instructions needs to be written under one structure.

Different ways to run Python code

Python programs can be executed in several different ways depending on the situation and tools available. Some commonly used methods to run Python code include:

• **Interactive mode (REPL):** In this method, commands are typed directly into the Python shell, and the results appear immediately after execution.

• **Script mode (.py files):** In this approach, Python code is written in a file with a **.py** extension and then executed using the command line.

• **Jupyter Notebooks:** These provide an interactive environment where code can be written, tested, and displayed along with output in separate sections.

• **Integrated Development Environments (IDEs):** Python code can also be run inside software tools such as **VS Code** or **PyCharm**, which offer features like debugging and code suggestions.

Common syntax rules in Python

Python follows certain basic syntax rules that programmers need to understand in order to write correct code. Some of the most important rules are listed below:

• Proper indentation (spacing) is required because it defines the structure of code blocks
• Semicolons are usually not needed at the end of statements
• A colon (:) is used to indicate the start of a block of code
• Python is case-sensitive, so uppercase and lowercase letters are treated differently
• Comments are written using the # symbol to explain parts of the code

Method of checking the installed version of Python

To find out which version of Python is installed on a computer, you can use the command line. The usual way is to type the following command:

```
python --version
```

or

```
python3 --version
```

After running one of these commands, the system will display the installed Python version, such as **Python 3.11.5**. The exact command used may vary depending on the operating system and how Python was set up on the system.

Dynamic typing in Python

Dynamic typing in Python means that you do not have to declare the data type of a variable before using it. Instead, Python automatically decides the type of the variable based on the value assigned to it while the program is running.

This also means that the same variable can store values of different types at different times during execution.

```
x = 10     # int
x = "Hi"   # now x is str
```

MCQ's

Which type of language is Python mainly considered?

A. Low-level language
B. Machine language
C. High-level language ✅
D. Assembly language

What is the default extension of a Python compiled file?

A. .class
B. .exe
C. .pyc ✅
D. .bin

Which of the following is NOT a feature of Python?

A. Easy syntax
B. Platform independent
C. Requires compilation before execution only
D. Open-source language

Which operator is used for exponentiation in Python?

A. ^
B. ** ✅
C. %%
D. //

What will be the output of:print(10 // 3)?

A. 3.33
B. 3.0
C. 3 ✅
D. 4

Which keyword is used to take input from the user?

A. scan
B. input() ✅
C. get()
D. read()

What will print(2 * "Hi") output?

A. HiHi ✅
B. 2Hi
C. Hi Hi
D. HiHiHi

Which of the following is a mutable data type?

A. tuple
B. string
C. list ✅
D. int

Which of the following is used to display output in Python?

A. echo
B. printf
C. print() ✅
D. show()

What is the result of bool(1)?

A. False
B. True ✅
C. Error
D. None

Which symbol is used for floor division?

A. /
B. %
C. // ✅
D. **

Which of the following is a keyword in Python?

A. value
B. var
C. if ✅
D. number

What will be the output of type("123")?

A. int
B. float
C. str ✅
D. char

Which of the following is used to store multiple values in a single variable?

A. variable
B. container types (list, tuple) ✅
C. operator
D. function

What is Python case sensitivity behavior?

A. Not case sensitive
B. Case sensitive ✅
C. Depends on OS
D. Only for variables

Chapter 2: Variables and Data Types

Variables and Typing Constraints

Concept of a variable in python

In Python, a variable is used to store data so that it can be used later in a program. You create a variable by giving it a name and assigning a value using the **= (assignment) operator**. Python automatically determines the type of value stored, so there is no need to declare the data type separately.

```
name = "Alice"
age = 25
```

Rules for naming variables in Python

When naming variables in Python, certain rules must be followed to ensure the names are valid and accepted by the language. These rules help avoid errors and make the code easier to understand.

Some important rules for naming variables in Python are:

• A variable name must begin with a letter (a–z or A–Z) or an underscore (_)
• A variable name cannot start with a number
• It can include letters, digits, and underscores within the name
• Reserved keywords such as **for**, **if**, **class**, and **while** cannot be used as variable names
• Python treats uppercase and lowercase letters differently, so names like **Age** and **age** are considered separate variables

Dynamic typing in Python and the difference of dynamic typing from static typing

Python follows the concept of **dynamic typing**, which means you do not need to declare the data type of a variable before using it. The type of the variable is automatically decided while the program is running, based on the value assigned to it. Another useful feature of dynamic typing is that the same variable can store different types of values at different times during program execution.

In contrast, **static typing** is used in languages such as Java or C++. In these languages, the data type of a variable must be declared before assigning a value to it, and once declared, the type usually cannot be changed later in the program.

```
x = 10      # int
x = "text"  # now it becomes a string
```

Type inference and its use in python

Type inference in Python refers to the ability of the language to automatically identify the data type of a variable based on the value assigned to it. This means that programmers do not need to manually specify whether a variable is an **int**, **str**, or any other type—Python determines it on its own during execution.

This feature makes coding faster and simpler because it reduces the need to write extra type declarations.

```
x = 5       # inferred as int
name = "Bob" # inferred as str
```

Core Built-in Data Types

Python's basic built-in data types

Python provides several built-in data types that are used to store different kinds of information. These data types help programmers handle numbers, text, logical values, and collections of data in an organized way.

Some of the main built-in data types in Python include:

• **Numeric types:** These are used to store numbers, such as **int**, **float**, and **complex**
• **Text type:** The **str** type is used to store text or sequences of characters
• **Boolean type:** The **bool** type stores logical values like **True** or **False**
• **None type:** The **NoneType** represents the absence of a value
• **Collection types:** These include **list**, **tuple**, **dict**, and **set**, which are used to store groups of multiple values

Difference between int, float, and complex in Python

In Python, **int**, **float**, and **complex** are numeric data types, but each one is used to represent different kinds of numbers.

• **int:** This type is used to store whole numbers that do not have any decimal part, such as **5**, **10**, or **-3**.

• **float:** This type is used for numbers that include decimal values, such as **5.5**, **3.14**, or **-0.25**.

• **complex:** This type is used to represent numbers that contain both a real part and an imaginary part. These numbers are written using **j** to represent the imaginary value, for example **3 + 4j**.

Bool type in Python and its behavior with other types

In Python, the **bool** data type is used to represent logical values, which can be either **True** or **False**. It is commonly used in conditions and decision-making statements.

When Python checks conditions, it automatically treats certain values as **False**. These include values that are empty or represent zero, such as:

- **0**
- **0.0**
- **''** (empty string)
- **[]** (empty list)
- {} (empty dictionary)
- **None**

All other values that are not empty or zero are usually considered **True** when used in conditions.

Output of bool(0), bool(''), and bool([]) in Python

The results of **bool(0)**, **bool('')**, and **bool([])** in Python are all **False**. This happens because these values represent either zero or empty objects, and Python treats such values as False when converting them to Boolean type.

```
bool (0)    # False
bool ('')   # False
bool ([])   # False
```

None in Python, and its difference from 0, '', or False

In Python, None is a special constant that represents the absence of any value. It is often used when a variable has no value assigned yet or when a function does not return anything.

None is not the same as 0, '' (empty string), or False. Each of these represents a different kind of value:

- **0 represents the number zero.**
- **'' represents an empty text string.**
- **False represents a Boolean value that means "not true."**

Even though all of these may behave similarly in some conditions (for example, they can be treated as False in checks), they are still completely different data types and are not equal to each other.

Type Verification and Mutability

Checking the data type of a variable in Python

In Python, you can find out the type of a variable by using the **type()** function. This function displays the data type of the value stored in the variable, which helps in understanding how the data is being handled in the program.

```
x = 10
print(type(x)) # Output: <class 'int'>
```

Checking if a variable is None and why we do not use == None

The main reason is that **None is a special object in Python**, and we are not comparing values, but checking identity.

- is checks whether both sides refer to the **same object in memory**
- == only checks whether values are **equal**

So, is None is the correct and safe way.

Simple explanation:

We use is None because None is unique, and we just want to confirm “Is this exactly None?” not “Is this equal to something like None?”

Example:

```
x = None

if x is None:
    print("x has no value")
```

The difference between mutable and immutable data types in Python (with examples)

In Python, data types are divided into mutable and immutable based on whether their values can be changed after they are created. This helps us understand how data behaves in memory.

Some important points are:

- Mutable data types are those whose values can be changed after creation
- Immutable data types are those whose values cannot be changed after creation
- When a mutable object is modified, the same memory object is updated
- When an immutable object is changed, a new object is created in memory

Examples:

- Mutable data types include list, dictionary, and set
- Immutable data types include string, integer, float, and tuple

Type Conversion Methods

Type casting in Python and its examples

Type casting in Python refers to the process of changing a value from one data type into another. This is useful when you need a value to be in a specific format so that certain operations can be performed correctly.

Python provides built-in functions that allow you to convert data types easily, such as **int()**, **float()**, and **str()**.

```
x = int (3.7)       # 3
y = float ("4.5")   # 4.5
z = str (100)       # "100"
```

Difference between implicit and explicit type conversion

In Python, data types can be converted in two main ways: **implicit conversion** and **explicit conversion**.

- **Implicit type conversion:** This happens automatically when Python changes one data type into another without the programmer doing anything. It usually occurs when different data types are used together in an expression, and Python converts them to make the calculation possible.

- **Explicit type conversion:** This type of conversion is done manually by the programmer using built-in functions such as **int()**, **float()**, or **str()** to change a value into the required data type.

```
# Implicit
x = 10 + 3.5      # x becomes float

# Explicit
y = int (3.9)     # y is 3
```

String-to-number conversion in Python and the behavior when the string is invalid

In Python, strings can be changed into numbers by using built-in functions like **int()** and **float()**. These functions read the numeric value written inside the string and convert it into the correct number format.

If the string contains only digits (or valid decimal values), the conversion works without any problem. However, if the string includes letters or any non-numeric characters, Python cannot convert it and gives an error called **ValueError**.

```
int ("123")   # OK
int ("abc")   # ValueError
```

MCQ's

Which rule is correct for naming variables in Python?

A. Must start with a number
B. Can use spaces
C. Cannot use keywords like if, for ✅
D. Both B and C are correct

What will be the output of type(10)?

A. float
B. str
C. int ✅
D. bool

Which of the following is an immutable data type?

A. list
B. dictionary
C. tuple ✅
D. set

What is the result of type(3.0)?

A. int
B. float ✅
C. str
D. complex

Which of the following creates a list?

A. (1,2,3)
B. {1,2,3}
C. [1,2,3] ✅
D. "1,2,3"

What will be the output of bool("Python")?

A. False
B. True ✅
C. Error
D. None

Which data type is used to store decimal numbers?

A. int
B. float ✅
C. bool
D. str

What will happen if we use int("10a")?

A. 10
B. Error ✅
C. 0
D. 1

Which function converts a value into string?

A. int()
B. float()
C. str() ✅
D. char()

What is the output of type(None)?

A. int
B. NoneType ✅
C. bool
D. str

Which of the following is used for complex numbers?

A. 5 + 2j ✅
B. 5 + 2i
C. 5 + j2
D. 5 + 2x

What is the output of float(10)?

A. 10
B. 10.0 ✅
C. "10"
D. Error

Which of the following is NOT a Python data type?

A. list
B. tuple
C. array ✅
D. dict

Correct Answer: **C. array**

What does type(True + False) return?

A. str
B. bool

C. int ✅
D. float

What will be the output of int(True)?

A. 0
B. 1 ✅
C. True
D. Error

Chapter 3: Working with Numbers and Strings

Mathematical Operations

Basic arithmetic operators in Python

In Python, arithmetic operators are used to perform basic mathematical calculations on numbers. These operators help us do operations like addition, subtraction, multiplication, and more.

Some important arithmetic operators in Python are:

- The **addition (+)** operator is used to add two numbers
- The **subtraction (-)** operator is used to subtract one number from another
- The **multiplication (*)** operator is used to multiply two numbers
- The **division (/)** operator is used to divide one number by another and gives a float result
- The **floor division (//)** operator divides numbers and gives the integer part only
- The **modulus (%)** operator gives the remainder after division
- The **exponentiation ()**** operator is used to find the power of a number

Examples:

```
a = 10
b = 3

print(a + b)   # 13
print(a - b)   # 7
print(a * b)   # 30
print(a / b)   # 3.333...
print(a // b)  # 3
print(a % b)   # 1
print(a ** b)  # 1000
```

Difference between / and // in Python

In Python, both / and // are division operators, but they work in different ways and give different types of results.

- The **/ (division) operator** is used for normal division and always gives the result in decimal (float form), even if the answer is a whole number
- The **// (floor division) operator** is used to divide numbers and gives only the integer part by removing the decimal part

Examples:

```
a = 10
b = 3

print(a / b)   # 3.333...
print(a // b)  # 3
```

Purpose of the math module in Python

The math module in Python is a built-in module that provides a collection of mathematical functions and constants. It is used to perform advanced mathematical operations that are not directly available with basic operators.

Some important purposes of the math module are:

- It provides functions for calculations like square root, power, logarithm, and trigonometry
- It helps in performing complex mathematical operations easily without writing extra code
- It includes useful constants like π (pi) and e (Euler's number)
- It makes mathematical programming faster and more accurate

Examples of math module functions:

```
import math

print(math.sqrt(16))   # 4.0
print(math.pow(2, 3))  # 8.0
print(math.pi)         # 3.14159...
print(math.log(10))    # natural logarithm
```

Method of rounding numbers in python

In Python, rounding means converting a decimal number into a simpler form by adjusting it to the nearest whole number or a required number of decimal places. Python provides built-in methods to do this easily.

Some important methods for rounding numbers are:

- The **round() function** is used to round a number to the nearest integer or to a specific number of decimal places
- If the decimal part is 0.5 or more, the number is rounded up
- If the decimal part is less than 0.5, the number is rounded down
- You can also control how many digits after the decimal point you want

Examples:

```
print(round(4.3))      # 4
print(round(4.7))      # 5
print(round(5.678, 2)) # 5.68
```

Main difference between int() and float()

In Python, int() and float() are type conversion functions used to change values into different numeric types. They both convert numbers, but they work in different ways.

• The **int() function** is used to convert a value into an integer (whole number without decimal part)
• The **float() function** is used to convert a value into a floating-point number (number with decimal part)

Important differences:

• int() removes the decimal part and keeps only the whole number
• float() adds a decimal part to the number (even if it is .0)
• int() is used when we need whole numbers
• float() is used when we need more precise values with decimals

Examples:

```
print(int(5.9))    # 5
print(int(3.2))    # 3

print(float(5))    # 5.0
print(float(7.8))  # 7.8
```

String Fundamentals and Slicing

Creating strings in Python

In Python, strings are created by writing text inside quotes. A string can include letters, numbers, symbols, or even spaces. Python allows different types of quotes to define a string.

Some important ways to create strings in Python are:

• A string can be written using single quotes ' '
• A string can also be written using double quotes " "
• For multi-line strings, triple quotes ''' ''' or """ """ are used
• Strings can contain any text, including words, sentences, or characters

Examples:

```
name = 'Python'
city = "Siliguri"

print(name)
print(city)
```

Multi-line string example:

```
text = """This is
a multi-line
string."""
print(text)
```

Simple understanding:

Accessing individual characters or slices in a string

In Python, a string is a sequence of characters, so we can access individual characters or a part of the string using indexing and slicing.

1. Accessing individual characters (Indexing)

- Each character in a string has an index number
- Indexing starts from **0** (left to right)
- We can also use negative indexing to access characters from the end

Examples:

```
text = "Python"

print(text[0])   # P
print(text[2])   # t
print(text[-1])  # n
```

2. Accessing a part of string (Slicing)

- Slicing is used to get a range of characters from a string
- It is written as string[start:end]
- The end index is not included in the result

Example:

```
text = "Python"
print(text[0:4])   # Pyth
print(text[2:6])   # thon
```

3. Slicing with step value

• We can also define a step value using string[start:end:step]
• It skips characters based on the step value

Example:

```
text = "Python"

print(text[0:6:2])  # Pto
```

String slicing and its working process

String slicing in Python is a method used to extract a part (substring) from a string. It allows us to select a range of characters from a string using index positions.

How slicing works:

• Slicing is written as string[start : end : step]
• The **start index** is where slicing begins (included)
• The **end index** is where slicing stops (not included)
• The **step** decides how many characters to skip (optional)
• Indexing starts from 0 in Python

Example:

```
text = "Python"
print(text[0:4])   # Pyth
```

Checking if strings are mutable or not in Python

In Python, we can check whether strings are mutable or immutable by trying to change a character in a string and observing the result.

• If a data type is mutable, we can change its value after creation
• If it is immutable, we cannot change its value directly

Checking with an example:

```
text = "Python"
text[0] = "J"
print(text)
```

String Operations and Formatting

Common string methods in Python

In Python, string methods are built-in functions used to perform different operations on strings. They help in modifying, checking, and processing text easily.

Some common string methods in Python are:

- The **lower() method** is used to convert all characters in a string to lowercase
- The **upper() method** is used to convert all characters in a string to uppercase
- The **strip() method** is used to remove extra spaces from the beginning and end of a string
- The **replace() method** is used to replace a part of a string with another value
- The **split() method** is used to divide a string into a list based on a separator
- The **find() method** is used to find the position of a substring in a string
- The **len() function** is used to find the length of a string

Examples:

```
text = "  Hello Python  "

print(text.lower())    # hello python
print(text.upper())    # HELLO PYTHON
print(text.strip())    # Hello Python
print(text.replace("Python", "World"))
```

String concatenation in Python and method of joining strings

String concatenation means joining two or more strings together to form a single string. Python provides different ways to combine strings easily.

Some important methods of joining strings in Python are:

- The **+ operator** is used to directly join two strings
- The **+= operator** is used to add one string to another and update the original string
- The **join() method** is used to join multiple strings from a list or sequence with a separator
- Strings can also be joined using formatting methods like f-strings

Examples:

```
a = "Hello"
b = "World"

print(a + b)          # HelloWorld
print(a + " " + b)    # Hello World
```

Using += operator:

```
text = "Hello"
text += " Python"
print(text)
```

Using join() method:

```
words = ["I", "love", "Python"]
print(" ".join(words))
```

Formatting strings in Python

String formatting in Python means creating a well-structured string by combining text with variables in a readable way. It helps to insert values inside a string without manually using many + operators.

Some important methods of formatting strings in Python are:

- The **f-string method** is the most modern and easy way to format strings
- The **format() method** is used to insert values into placeholders {}
- The **% operator method** is an older way of formatting strings

Examples:

Using f-string:

```
name = "Rahul"
age = 20

print(f"My name is {name} and my age is {age}")
```

Using format() method:

```
print("My name is {} and age is {}".format("Rahul", 20))
```

Using % operator:

```
print("My name is %s and age is %d" % ("Rahul", 20))
```

F-strings and their preference

F-strings (formatted string literals) are a way to create strings in Python where we can directly insert variables or expressions inside a string using curly braces {}. They are written by adding an **f or F before the string**.

Example of f-string:

```
name = "Amit"
age = 21
```

```
print(f"My name is {name} and my age is {age}")
```

We can also use expressions inside f-strings:

```
a = 5
b = 3

print(f"Sum is {a + b}")
```

f-strings preference

- They are **easy to write and read** compared to other methods
- They are **faster in execution** than format() and % methods
- They allow **direct use of variables and expressions inside the string**
- They make code **clean, short, and more understandable**
- They reduce the need for extra concatenation using +

Advanced String Concepts

String interning in Python

String interning in Python is a memory optimization technique where Python stores only one copy of certain strings in memory and reuses them instead of creating multiple copies of the same string.

How it works:

- When two or more strings have the same value, Python may store them at the same memory location
- This helps to save memory and improve performance
- Python automatically interns some strings, especially small or frequently used strings

Example:

```
a = "hello"
b = "hello"
print(a is b)
```

Differences between is and == for strings

In Python, both is and == are used for comparison, but they work in different ways when checking strings.

- **The == operator (value comparison)**

- It checks whether two strings have the **same value/content**
- It does not care about memory location
- It is used for normal comparison of strings

Example:

```
a = "hello"
b = "hello"
print(a == b)
```

MCQ's

What will be the result of 10 % 3 in Python?

A. 3
B. 1 ✅
C. 0
D. 2

Which function is used to find the minimum value in numbers?

A. min() ✅
B. lowest()
C. small()
D. lesser()

What does math.pow(2, 3) return?

A. 6
B. 8.0 ✅
C. 9
D. 4

What will be the output of round(5.678, 1)?

A. 5.6
B. 5.7 ✅

C. 5.68
D. 5.8

Which function is used to get the absolute value?

A. abs() ✅
B. absolute()
C. mod()
D. value()

What will be the output of len("Data")?

A. 3
B. 4 ✅
C. 5
D. Error

What does 'Python'.lower() return?

A. PYTHON
B. python ✅
C. Python
D. error

Which method checks if a string ends with a specific word?

A. endswith() ✅
B. finishwith()
C. end()
D. endstr()

What will be the output of "abc123".isalnum()?

A. True ✅
B. False
C. Error
D. None

What does string concatenation mean?

A. Adding numbers
B. Joining strings together ✅
C. Removing strings
D. Sorting strings

What is the output of "Hello"[0]?

A. e
B. H ✅
C. o
D. Error

What will be the result of 'Python'.replace('P','J')?

A. Jython ✅
B. Python
C. Jythonn
D. error

Which operator is used to repeat strings?

A. +
B. * ✅
C. %
D. //

What will be the output of "abc".find("b")?

A. 0
B. 1 ✅
C. -1
D. 2

Which method splits a string into a list?

A. split() ✅
B. divide()
C. break()
D. separate()

Chapter 4: Control Flow and Decision Making

Conditional Branching

Purpose of if, elif, and else in Python

These statements are used to make decisions in a program. They help Python choose which block of code should run based on conditions.

- if is used to check the first condition
- elif is used to check another condition if the first one is false
- else runs when none of the conditions are true

Example:

```
x = 10
if x > 0:
    print("Positive")
elif x == 0:
    print("Zero")
else:
    print("Negative")
```

It helps programs behave differently based on input or conditions.

Match statement in Python.

The match statement (introduced in Python 3.10) is used for pattern matching. It works like switch-case in other languages.

Example:

```
status = 200
match status:
    case 200:
        print("OK")
    case 404:
        print("Not Found")
    case _:
        print("Unknown status")
```

It makes multiple condition checking cleaner and easier.

Logical operators in Python.

Logical operators are used to combine multiple conditions.

- and → both conditions must be true
- or → at least one condition must be true
- not → reverses the condition

Example:

```
x = 5
if x > 0 and x < 10:
    print("Single digit positive number")
```

Multiple if vs elif

If we use multiple if statements, all conditions are checked separately.

If we use elif, only one block runs once a condition becomes true.

Example:

```
x = 10
if x > 5:
    print("Greater than 5")
if x > 8:
    print("Greater than 8")
```

Both will execute.

But:

```
x = 10
if x > 5:
    print("Greater than 5")
elif x > 8:
    print("Greater than 8")
```

Only first block runs.

Simple program using nested if

Nested if means an if statement inside another if statement. It is used when one condition depends on another.

Example:

```
age = 25
if age > 18:
    if age < 30:
        print("Young adult")
```

First it checks age > 18, then inside it checks age < 30.

Good practice for nested conditionals

To write clean code:

- Avoid too many nested if blocks
- Use elif instead of deep nesting
- Break logic into functions
- Keep conditions simple

Example:

```
age = 19

if age < 13:
    print("Child")
elif age < 20:
    print("Teenager")
else:
    print("Adult")
```

One-line conditional (ternary operator)

This is a shortcut for if-else. It helps write simple conditions in one line.

Example:

```
num = 15
result = "Positive" if num > 0 else "Negative"
print(result)
```

Use of _ wildcard in match

_ is used as a default case in match. It runs when no other case matches.

It works like “else”.

Example:

```
grade = "D"
match grade:
    case "A":
        print("Excellent")
    case "B":
        print("Good")
    case _:
        print("Needs Improvement")
```

Logic Evaluation

Logical operators in Python

Logical operators are used to combine multiple conditions.

- and → both conditions must be true
- or → at least one condition must be true
- not → reverses condition

Example:

```
age = 18
has_id = True
print(age >= 18 and has_id)
```

Logical operators to simplify conditions

Logical operators help combine multiple conditions into a single statement.

Example:

```
score = 85
if score >= 80 and score <= 90:
    print("Good performance")
```

Short form:

```
if 80 <= score <= 90:
    print("Good performance")
```

Truthy and falsy values.

Python treats some values as False in conditions:

- Falsy → 0, None, False, "", [], {}
- Truthy → everything else

Example:

```
text = ""

if text:
    print("Has value")
else:
    print("Empty string")
```

Match-case for menu system.

match-case is useful for menu-driven programs because it makes code clean and organized.

Example:

```
option = "2"

match option:
    case "1":
        print("Play Game")
    case "2":
        print("Settings")
    case "3":
        print("Exit")
    case _:
        print("Invalid choice")
```

Chained comparisons (a < b < c)

Python allows multiple comparisons in one line. It improves readability.

It means both conditions must be true.

Example:

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

Equivalent to:
a < b and b < c

Truthy and Falsy values

In Python, every value is treated as either True or False in conditions.

- Truthy values → treated as True (non-empty or non-zero values)
- Falsy values → treated as False

Python decides automatically if something is “true-like” or “false-like”.

Example:

```
print(bool(100))     # True
print(bool(""))      # False
print(bool([]))      # False
```

What does not operator do?

The not operator reverses the Boolean value.

- If value is True → becomes False
- If value is False → becomes True

It flips the result.

Example:

```
x = True
print(not x)
y = 0
print(not y)
```

Relational operators

Relational operators compare two values and return True or False.

- == → equal to
- != → not equal to
- > → greater than

- < → less than
- >= → greater or equal
- <= → less or equal

Example:

```
a = 15
b = 20
print(a < b)
print(a != b)
```

Checking if a variable exists

We can check if a variable exists using locals() or globals().

Example:

```
if "x" in locals():
    print("x exists")
else:
    print("x does not exist")
```

Is Match-case a replacement for if-else.

Not completely.

- match-case is best for fixed values or patterns
- if-else is better for complex conditions and ranges

Example:

```
value = 2

match value:
    case 1:
        print("One")
    case 2:
        print("Two")
    case _:
        print("Other")
```

MCQ’s

What is the main purpose of control flow in Python?

A. To store data
B. To execute code in order only
C. To make decisions in program execution ✅
D. To define variables

Which keyword is used for multiple conditions check?

A. loop
B. elif
C. nested
D. both if and elif can be used together ✅

What will be the output of if []: ?

A. True
B. False
C. No output (False condition) ✅
D. Error

Which of the following is used when no condition matches in match-case?

A. case end
B. case default
C. case _ ✅
D. case none

What will be the result of True and False?

A. True
B. False ✅
C. Error
D. None

Which block runs when if condition is False?

A. if
B. elif
C. else ✅
D. try

What is the output of 10 > 5 and 3 > 4?

A. True
B. False ✅
C. Error
D. None

Which symbol is used for logical NOT?

A. !
B. not ✅
C. ~
D. ^

What will be the output of bool(100)?

A. False
B. True ✅
C. Error
D. 0

Which statement allows checking multiple values of a single variable?

A. if-else
B. loop
C. match-case ✅
D. function

What is the output of 0 or 5?

A. 0
B. 5 ✅
C. True
D. False

Which keyword stops further conditions in a loop?

A. stop
B. break ✅
C. end
D. exit only

What happens if no condition in if-elif is True and no else is present?

A. Error
B. Nothing executes ✅
C. Program crashes
D. Restart

What is the result of 7 >= 7?

A. False
B. True ✅
C. Error
D. None

Which operator checks both conditions must be True?

A. or
B. and ✅
C. not
D. xor

Chapter 5: Iteration in Python

Loop Foundations

Two main types of loops in Python

Python has two main loops used for repeating tasks.

• for loop is used when we know the sequence (like list, string, range)
• while loop is used when repetition depends on a condition

In simple words:

- for loop = fixed iteration over items
- while loop = runs until condition becomes false

Example:

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

count = 1
while count <= 3:
    print(count)
    count += 1
```

The `for` Loop Mechanics

For loop working process

A for loop goes through each item in a sequence one by one and executes the block of code.

It automatically moves from one element to the next.

Example:

```
for word in "Code":
    print(word)
```

Purpose of range() function

range() generates a sequence of numbers. It is mostly used in loops when we need repetition.

• range(n) → 0 to n-1
• range(start, end) → custom range
• range(start, end, step) → step control

Example:

```
for i in range(2, 7):
    print(i)
```

Reversing a loop using range()

We can reverse a loop by using a negative step value.

Example:

```
for i in range(5, 0, -1):
    print(i)
```

Looping through a string

When we loop through a string, Python treats it like a sequence of characters.

Example:

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

Difference between while and for loop

• for loop → used when number of steps is known
• while loop → used when condition controls repetition

Example:

```
i = 0
while i < 4:
    print(i)
    i += 1
```

Advanced Iteration Strategies

Nested loops

A nested loop means a loop inside another loop. It is used for working with rows and columns or combinations.

Example:

```
for i in range(2):
    for j in range(3):
        print("i=", i, "j=", j)
```

Iterating index and value together

We use enumerate() to get both index and value at the same time.

Example:

```
colors = ["red", "blue", "green"]
for i, color in enumerate(colors):
    print(i, color)
```

Infinite while loops

Python does not stop infinite loops automatically. It depends on the programmer to write correct conditions.

If condition never becomes false, loop runs forever.

Example:

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

Else with loops

The else block runs only when the loop finishes normally (without break).

Example:

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

Loop Control and Generators

Loop control statements

Loop control statements change how loops behave.

• break → stops the loop completely
• continue → skips current iteration
• pass → does nothing, used as placeholder

Example:

```
for i in range(5):
    if i == 3:
        break
    print(i)
```

Difference between break and continue

• break ends the loop immediately
• continue skips only one iteration and continues next

Example:

```
for i in range(5):
    if i == 2:
        continue
    print(i)
```

Iterators

Iterators are objects that return items one by one instead of all at once.

They use:

- __iter__() → to initialize
- __next__() → to get next value

Example:

```
my_list = [5, 10, 15]
it = iter(my_list)
print(next(it))
print(next(it))
```

Comprehensions

List comprehension

List comprehension is a short way to create lists using loops in a single line.

It makes code shorter and cleaner.

Example:

```
nums = [1, 2, 3, 4]
even = [n for n in nums if n % 2 == 0]
print(even)
```

Reason that list comprehension is better

List comprehension is preferred because:

• It reduces code length
• It is faster than normal loops
• It improves readability
• It avoids extra lines of code

Example:

```
squares = [x*x for x in range(6)]
print(squares)
```

MCQ's

Which keyword is used to create a loop that runs based on a condition?

A. for
B. while ✅
C. loop
D. repeat

What will be the output of range(1, 5)?

A. 1 2 3 4 5
B. 1 2 3 4 ✅

C. 0 1 2 3 4
D. 2 3 4 5

Which loop is mainly used when number of iterations is known?

A. while
B. for ✅
C. do
D. repeat

What will happen if a while condition is always True?

A. Loop runs once
B. Loop stops automatically
C. Infinite loop occurs ✅
D. Error occurs

Which statement is used as a placeholder inside loops?

A. continue
B. pass ✅
C. break
D. skip

What is the output of range(5, 1, -1)?

A. 5 4 3 2 1 ✅
B. 5 4 3 2
C. 5 4 3 2 1 0
D. 1 2 3 4 5

What does nested loop mean?

A. Loop inside function
B. Loop inside another loop ✅
C. Two separate loops
D. Infinite loop

Which keyword is used to stop loop completely when condition is met?

A. stop
B. exit
C. break ✅
D. return

What will be the output of:

```
for i in range(1):
    print("Hello")
```

A. Hello once ✅
B. No output
C. Error
D. Infinite Hello

Which function is used to convert list into iterator?

A. iter() ✅
B. range()
C. next()
D. loop()

What will be the output of next(iter([10, 20]))?

A. 20
B. 10 ✅
C. Error
D. None

Which statement skips current loop iteration?

A. break
B. continue ✅
C. pass
D. skip

What will print(range(3)) directly output?

A. 0 1 2
B. range(0, 3) object representation ✅

C. [0, 1, 2]
D. Error

What is list comprehension mainly used for?

A. Writing functions
B. Creating lists in a single line ✅
C. Loop termination
D. Input handling

Which loop executes at least zero or more times depending on condition?

A. for loop
B. while loop ✅
C. do-while
D. nested loop

Chapter 6: Collections and Data Structures

Sequential Data: Lists and Tuples

Difference between list and tuple in Python

List and tuple are both used to store multiple values, but they are different in how they behave.

• A **list is mutable**, meaning we can change its values after creation
• A **tuple is immutable**, meaning once created, it cannot be changed

Simple idea:

- List = changeable collection
- Tuple = fixed collection

Example:

```
my_list = [5, 10, 15]
my_list[1] = 99
my_tuple = (5, 10, 15)
```

Creating a list and accessing elements

A list is created using square brackets []. We can access elements using index numbers starting from 0.

Index helps us get specific items (Matthes, 2023).

Example:

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

print(fruits[0])  # mango
print(fruits[2])  # banana
```

Common list methods

List methods are built-in functions used to modify or work with lists.

- append() → adds item at end
- insert() → adds item at specific position
- extend() → adds multiple items
- remove() → removes specific item
- pop() → removes last or indexed item
- sort() → arranges in order
- reverse() → reverses list

Example:

```
nums = [3, 1, 4]
nums.append(10)
nums.sort()
print(nums)
```

Advantages of tuples over lists

Tuples are better in some cases because:

- They are faster than lists
- They use less memory
- They protect data from accidental changes
- They can be used as dictionary keys

Simple idea: tuples are safe and stable data containers.

Example:

```
coordinates = (10, 20)
```

Unordered Data: Sets

Purpose of sets in Python

A set is used to store only unique values.

- It automatically removes duplicates
- It does not maintain order
- It is useful for mathematical operations

Example:

```
numbers = {2, 2, 3, 4, 4}
print(numbers)
```

Set operations

Sets support mathematical operations like union, intersection, and difference.

• Union → combines all elements
• Intersection → common elements
• Difference → elements in one set only

Example:

```
a = {1, 3, 5}
b = {3, 4, 6}
print(a | b)
print(a & b)
print(a - b)
```

Handling duplicates in sets

Sets automatically remove duplicate values because they only allow unique elements.

Example:

```
s = {10, 10, 20, 30, 30}
print(s)
```

Output will contain only unique values.

Key-Value Mapping: Dictionaries

Dictionaries in Python

A dictionary stores data in key-value pairs.

• Key is like a name
• Value is the data

It is useful for structured information.

Example:

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

Accessing and modifying dictionary values

We access values using keys and can also update them.

Example:

```
student = {"name": "Ravi", "marks": 80}

student["marks"] = 90
print(student["marks"])
```

Useful dictionary methods

Common dictionary methods help in working with data easily.

• keys() → returns all keys
• values() → returns all values
• items() → returns key-value pairs
• get() → safely access value
• update() → modify dictionary
• pop() → remove item

Example:

```
data = {"a": 1, "b": 2}
print(data.keys())
```

Iterating through dictionary

We can loop through a dictionary to get keys and values together.

Example:

```
student = {"name": "Asha", "age": 19}
for k, v in student.items():
    print(k, v)
```

Are dictionary keys mutable?

No, dictionary keys must be immutable.

Allowed keys:

- strings
- numbers

- tuples

Not allowed:

- lists
- sets

Example:

```
valid = {("x", 1): "value"}
```

When to use list, tuple, set, dictionary

Each structure has a different purpose:

• List → when data changes frequently
• Tuple → when data should not change
• Set → when uniqueness is needed
• Dictionary → when data is stored as key-value pairs

Example:

```
items = [1, 2, 3]
fixed = (1, 2, 3)
unique = {1, 2, 3}
info = {"id": 101}
```

Checking key in dictionary

We can check if a key exists using in.

Example:

```
student = {"name": "John"}
if "name" in student:
    print("Key exists")
```

MCQ's

Which data structure stores key-value pairs?

A. list
B. tuple
C. dictionary ✅
D. set

What will be the output of:

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

A. [10, 20, 30]
B. [40, 10, 20, 30]
C. [10, 20, 30, 40] ✅
D. Error

Which of the following allows duplicate values?

A. set
B. dictionary keys
C. list ✅
D. tuple keys

What will be the output of:

```
s = {5, 6, 7}
print(type(s))
```

A. list
B. tuple
C. set class type ✅
D. dict

Which symbol is used for empty set creation?

A. {}
B. []
C. () ✅
D. <>

hat will be the output of:

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

A. 2
B. 3 ✅
C. 4
D. Error

Which method removes last element from a list?

A. remove()
B. delete()
C. pop() ✅
D. discard()

What happens if we try to modify a tuple?

A. Allowed
B. Error occurs ✅
C. Changes partially
D. Converts to list

Which method returns all keys in a dictionary?

A. values()
B. items()
C. keys() ✅
D. get()

What will be the output of:

```
d = {"x": 10, "y": 20}
print(d.get("z"))
```

A. 0
B. Error
C. None ✅
D. False

Which operation is used to combine two sets?

A. +
B. merge()
C. union() ✅
D. join()

What will be the output of:

```
lst = [1, 2, 3]
lst.remove(2)
print(lst)
```

A. [1, 3] ✅
B. [2, 3]
C. [1, 2]
D. Error

Which data structure is best for fast lookup?

A. list
B. tuple
C. set or dictionary ✅
D. string

What will be the output of:

```
d = {"a": 5, "b": 10}
print("a" in d)
```

A. False
B. True ✅
C. Error
D. None

Which collection is indexed and ordered?

A. set
B. dictionary (keys only)
C. list and tuple ✅
D. None

Chapter 7: Booleans and Operators

Boolean Logic

Boolean values in Python

Boolean values are used to represent truth in Python. They help in decision making and conditions.

• There are only two Boolean values: True and False
• They are used in comparisons and logical operations

In simple words: Boolean means yes (True) or no (False).

Example:

```
is_raining = False
is_sunny = True
print(is_raining)
```

Empty containers in Boolean context

Empty containers are always considered False in Python.

• Empty list []
• Empty tuple ()
• Empty string ""
• Empty dictionary {}

Example:

```
data = ""
if data:
    print("Not empty")
else:
    print("Empty value")
```

Operator precedence

Operator precedence decides the order of execution in expressions.

Some operators are evaluated first, like and before or.

Example:

```
result = True or False and False
print(result)
```

First and is evaluated, then or.

Bitwise operators

Bitwise operators work on binary (0 and 1) values.

- & → AND
- | → OR
- ^ → XOR
- ~ → NOT
- << → left shift
- >> → right shift

Example:

```
a = 6
b = 3
print(a & b)
print(a | b)
```

Behaviour of and operator

The and operator returns:

- First falsy value if found
- Otherwise returns the last value

It does not always return True/False directly.

Example:

```
print(0 and 8)
print(4 and 9)
```

Short-circuit evaluation

Python stops checking conditions as soon as the result is known.

This improves performance.

Example:

```
def test():
    print("Function called")
    return True
print(True or test())
```

Function will not run because result is already True.

Result of 5 & 3

Bitwise AND compares binary digits.

5 = 0101
3 = 0011

Result = 0001 → 1

Example:

```
print(5 & 3)
```

How XOR (^) works

XOR returns 1 when bits are different.

Same bits → 0
Different bits → 1

Example:

```
print(7 ^ 2)
```

Bitwise NOT (~) operator

~ flips all bits of a number and gives a negative result in Python.

It works using complement rule.

Example:

```
print(~8)
```

Forcing Boolean evaluation

We use bool() to convert any value into True or False.

Example:

```
print(bool(50))
print(bool("Python"))
print(bool(0))
```

MCQ's

What is the output of bool("") in Python?

A. True
B. False ✅
C. 0
D. Error

Which value is considered truthy in Python?

A. 0
B. None
C. "Python" ✅
D. False

What does the operator or return if first value is True?

A. Second value
B. False
C. First value ✅
D. Error

What is the output of:

False or True

A. False
B. True ✅
C. None
D. Error

Which operator is used for bitwise OR?

A. ||
B. or

C. | ✅
D. &

What will be the output of:

not 0

A. False
B. True ✅
C. 0
D. Error

Which of the following is NOT a logical operator?

A. and
B. or
C. not
D. xor (in Python logical context) ✅

What is the result of 4 | 1?

A. 5 ✅
B. 4
C. 1
D. 5

What does 6 ^ 3 return?

A. 5 ✅
B. 6
C. 5
D. 7

Which value will make bool(value) return True?

A. ""
B. 0
C. "0" ✅
D. []

What is the output of:

True or False and False

A. False
B. True ✅
C. Error
D. None

Which operator has lowest precedence?

A. not
B. and
C. or ✅
D. ==

What is the result of:

5 is 5

A. True (in many cases)
B. False
C. Depends on memory optimization ✅
D. Error

What does bitwise AND (&) do?

A. Adds numbers
B. Compares values
C. Compares binary bits ✅
D. Subtracts values

What is the output of:

bool([])

A. True
B. False ✅
C. None
D. Error

Chapter 8: Advanced Python Concepts

Object-Oriented Programming (OOP)

Inheritance in Python

Inheritance is an important feature of Object-Oriented Programming where one class can reuse the properties and methods of another class.

- The class that gives features is called **parent class**
- The class that receives features is called **child class**
- It helps in code reusability and reduces repetition

A child class inherits behavior from **a** parent class.

Example:

```
class Animal:
    def sound(self):
        print("Animals make sound")
class Dog(Animal):
    pass
d = Dog()
d.sound()
```

Types of inheritance in Python

Python supports different types of inheritance:

- Single inheritance → one parent, one child
- Multiple inheritance → multiple parents
- Multilevel inheritance → parent → child → grandchild
- Hierarchical inheritance → one parent, multiple children
- Hybrid inheritance → combination of multiple types

Example:

```
class A:
    pass
class B(A):
    pass
```

```
class C(B):
    pass
```

Use of super() function

super() is used to call methods of the parent class inside a child class.

- It helps reuse parent class code
- Commonly used in constructors and overridden methods

Example:

```
class Vehicle:
    def start(self):
        print("Vehicle starting")
class Car(Vehicle):
    def start(self):
        super().start()
        print("Car is ready")
c = Car()
c.start()
```

Polymorphism in Python

Polymorphism means “many forms”. It allows the same function or method to behave differently based on the object.

Same name, different behavior.

Example:

```
class Bird:
    def fly(self):
        print("Bird can fly")
class Airplane:
    def fly(self):
        print("Airplane flies in sky")
def test(obj):
    obj.fly()
test(Bird())
test(Airplane())
```

Method overloading vs overriding

• Overloading → same method name, different arguments (not fully supported in Python)
• Overriding → child class changes parent method

Python mainly supports overriding.

Example:

```
class A:
    def show(self):
        print("Class A")
class B(A):
    def show(self):
        print("Class B")
obj = B()
obj.show()
```

Encapsulation in Python

Encapsulation means hiding internal data and restricting direct access.

• _variable → protected (suggested internal use)
• __variable → private (strong restriction)

Example:

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

Advanced Iteration and Modularity

Iterators in Python

Iterators are objects that return elements one by one instead of returning everything at once (Lutz, 2013) .

• They use __iter__() and __next__() methods
• Useful for large data handling

Example:

```
nums = [100, 200, 300]
it = iter(nums)
print(next(it))
print(next(it))
```

Creating a custom iterator

We can create our own iterator by defining a class with required methods.

It helps control how values are generated step by step.

Example:

```
class Numbers:
    def __init__(self):
        self.i = 1
    def __iter__(self):
        return self
    def __next__(self):
        if self.i <= 3:
            val = self.i
            self.i += 1
            return val
        else:
            raise StopIteration
obj = Numbers()
for num in obj:
    print(num)
```

Variable scope in Python

Scope means where a variable can be used.

- Local → inside function
- Global → outside function
- Enclosing → nested function
- Built-in → predefined functions

This is called LEGB rule.

Example:

```
x = 50
def demo():
    y = 10
    print(y)
demo()
print(x)
```

Use of global keyword

Global is used when we want to change a global variable inside a function.

Example:

```
count = 5
def update():
    global count
    count = 10
update()
print(count)
```

Modules and Architecture

What is a Python module?

A module is a Python file containing code like functions, variables, or classes.

It helps in organizing code into reusable parts.

Example:

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

Built-in modules in Python

Built-in modules are pre-installed and ready to use in Python.

They help perform common tasks easily.

Examples:

- math → calculations
- os → system operations
- random → random values
- datetime → date/time handling

Example:

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

Use of if __name__ == "__main__":

This condition checks whether a file is run directly or imported.

If run directly → code executes
If imported → code does not run

Example:

```
if __name__ == "__main__":
    print("Running directly")
```

Importing modules

We use import to use external modules in Python.

Example:

```
import random
print(random.randint(1, 10))
```

MCQ's

What is method overriding in Python?

A. Defining multiple methods with same name in one class
B. Child class redefining a method of parent class ✅
C. Deleting a method
D. Calling a function twice

Which of the following best describes encapsulation?

A. Hiding data using classes and restricting access ✅
B. Writing multiple functions
C. Using loops inside functions
D. Creating variables globally

What is the purpose of the self keyword?

A. Refers to class name
B. Refers to current object instance ✅
C. Refers to parent class
D. Refers to module

Which operator is used in multiple inheritance?

A. &
B. +
C. comma inside class definition ✅
D. ->

What will be the output of:

```
class A:
    def show(self):
    return "Hello"

obj = A()
print(obj.show())
```

A. Hello ✅
B. Error
C. None
D. show

What is abstraction in Python?

A. Showing all internal details
B. Hiding implementation and showing only functionality ✅
C. Copying code
D. Removing classes

Which keyword is used to create an abstract class?

A. abstract
B. @abstractmethod + ABC module ✅
C. virtual
D. interface

What does a module in Python contain?

A. Only variables
B. Only classes
C. Functions, classes, and variables ✅
D. Only loops

What is the purpose of import statement?

A. Delete a module
B. Load external code into program ✅
C. Compile program
D. Run program

What will happen if a module is imported twice?

A. Error occurs
B. It runs twice
C. It is loaded only once internally ✅
D. Program crashes

What is function overloading in Python?

A. Same function name with different parameters (simulated) ✅
B. Multiple functions with different names
C. Removing functions
D. Using loops in functions

What is the output of:

```
class A:
    def __init__(self):
    print("Object created")

A()
```

A. Object created ✅
B. Error
C. None
D. init

Which keyword is used for exception handling in Python?

A. error
B. try-except ✅
C. catch-only
D. handle

What is the use of name variable?

A. Stores class name
B. Stores module name
C. Checks if file is run directly or imported ✅
D. Stores function name

What is a package in Python?

A. A single function
B. A folder containing modules with **init**.py ✅
C. A variable type
D. A loop structure

Chapter 9: Working with Data and Dates

Date and Time Processing

Datetime module's uses

The datetime module in Python is used to work with dates and time in a program.

• It helps us get current date and time
• It allows calculations like adding or subtracting days
• It is used for scheduling, logging, and time-based applications

it helps Python handle real-world time information.

Example:

```
from datetime import datetime
today = datetime.now()
print(today)
```

Getting current date and time

We use datetime.now() to get the current system date and time.

Example:

```
from datetime import datetime
current = datetime.now()
print("Now:", current)
```

It shows year, month, day, time, and microseconds.

Formatting a date

We use strftime() to convert date into a readable format.

It helps display date in different styles.

Example:

```
from datetime import datetime
now = datetime.now()
```

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

Output format: 26-07-2025 18:40:10

Converting string into date

We use strptime() to convert a string into a datetime object.

Useful when reading data from users or files.

Example:

```
from datetime import datetime
date_str = "2025/07/26"
date_obj = datetime.strptime(date_str, "%Y/%m/%d")
print(date_obj)
```

Advanced Mathematics

Math.floor() uses

math.floor() rounds a number down to the nearest whole integer.

It always moves towards the smaller number.

Example:

```
import math
print(math.floor(7.9))
```

Square root using Python

We use math.sqrt() to find square root of a number.

It only works for zero or positive numbers.

Example:

```
import math
print(math.sqrt(25))
```

Difference between ceil() and floor()

• ceil() → rounds number UP
• floor() → rounds number DOWN

Simple idea:

- ceil = higher value
- floor = lower value

Example:

```
import math
print(math.ceil(6.2))
print(math.floor(6.8))
```

Data Serialization and Parsing

Dictionary to JSON conversion

We use json.dumps() to convert Python dictionary into JSON format.

Useful for APIs and data sharing (Sweigart, 2019).

Example:

```
import json
data = {"city": "Kolkata", "temp": 30}
json_data = json.dumps(data)
print(json_data)
```

JSON string to Python object

We use json.loads() to convert JSON into Python dictionary.

Used when reading API or file data.

Example:

```
import json
data = '{"city": "Delhi", "temp": 35}'
obj = json.loads(data)
print(obj["city"])
```

Use of re module

The re module is used for pattern matching using regular expressions.

It helps in searching and validating text.

Example:

```
import re
text = "My number is 98765"
match = re.search(r"\d+", text)
print(match.group())
```

Checking numbers in string using regex

We use \d to find digits inside a string.

Example:

```
import re
text = "ID: A45B67"
result = re.findall(r"\d", text)
print(result)
```

Meaning of \d{3} in regex

\d{3} means exactly 3 digits together.

Useful for patterns like PIN codes.

Example:

```
import re
text = "Code: 4567"
match = re.search(r"\d{3}", text)
print(match.group())
```

Package Management

PIP in Python

PIP is Python’s package manager used to install external libraries.

It helps download ready-made code packages.

Examples:

- numpy
- pandas
- requests

Simple idea: PIP is like an app store for Python libraries.

Installing packages using PIP

We install packages using terminal or command prompt.

Command:

pip install package_name

Example:

pip install pandas

Listing installed packages

We use pip list to see all installed Python libraries.

Command:

pip list

It shows package names and versions.

MCQ's

Which function gives the current time in Python?

A. time.now()
B. datetime.time()
C. datetime.now().time() ✅
D. clock.time()

What does strftime() do in Python?

A. Converts string to integer
B. Formats date/time into readable string ✅

C. Reads JSON file
D. Converts JSON to dict

What will be the output format of datetime.now()?

A. Only date
B. Only time
C. Date and time together ✅
D. Error

Which function is used to convert a string into a date?

A. strptime() ✅
B. parse()
C. convert()
D. dateformat()

What does math.ceil(4.1) return?

A. 4
B. 5 ✅
C. 4.1
D. Error

Which module is used for random number generation?

A. random ✅
B. math
C. datetime
D. os

What does random.randint(1, 5) return?

A. Float between 1 and 5
B. Integer between 1 and 5 inclusive ✅
C. Only 5
D. Error

Which function is used to convert JSON string into dictionary?

A. json.parse()
B. json.loads() ✅
C. json.convert()
D. json.read()

What does json.dumps() return?

A. Python list
B. JSON string representation of object ✅
C. File object
D. Dictionary

Which module is used for regular expression search?

A. regex
B. re ✅
C. search
D. pattern

What does re.match() do?

A. Searches anywhere in string
B. Matches only at beginning of string ✅
C. Replaces text
D. Splits string

What is the output of math.abs(-10)?

A. -10
B. 10 ✅

Which function is used to find all digits in a string using regex?

A. re.search()
B. re.findall() with \d+ ✅
C. re.match()
D. re.split()

What does pip freeze do?

A. Deletes packages
B. Lists installed packages with versions ✅
C. Installs packages
D. Updates Python

Which format is used in strftime("%Y")?

A. Month
B. Day
C. Year ✅
D. Hour

Chapter 10: Error Handling and Input

Exception Handling Mechanisms

Importance of error handling in Python

Handling errors makes programs:

• More stable
• User-friendly
• Prevents crashes
• Improves real-world usability

it keeps the program running smoothly even when users make mistakes.

Example:

```
try:
    num = int(input("Enter number: "))
except:
    print("Invalid input, try again")
```

Exception handling in Python

Exception handling is a way to handle errors in Python so that the program does not stop suddenly.

Instead of crashing, Python gives a chance to handle the error smoothly.

It uses:
• try → code that may cause error
• except → handles the error
• else → runs if no error occurs
• finally → always runs

Example:

```
try:
    result = 20 / 2
except:
```

```
    print("Error occurred")
else:
    print("Result is:", result)
finally:
    print("Program finished")
```

Basic try-except syntax

We use try-except to catch errors and prevent program crashes.

Simple idea: try first, handle if something goes wrong.

Example:

```
try:
    num = 5 / 0
except ZeroDivisionError:
    print("You cannot divide by zero")
```

Multiple except blocks

We can handle different errors separately using multiple except blocks (Barry, 2016).

Each error type can be handled differently.

Example:.

```
try:
    x = int("10a")
except ValueError:
    print("Invalid number format")
except ZeroDivisionError:
    print("Division issue")
```

Catching all exceptions

A generic except catches all errors, but it is not recommended for detailed debugging.

Example:

```
try:
    print(10 / 0)
except:
    print("Something went wrong")
```

Use of else in exception handling

The else block runs only when no error happens in the try block.

It is used for successful execution code.

Example:

```
try:
    num = 10 + 5
except:
    print("Error occurred")
else:
    print("Sum is:", num)
```

Specific Errors and Validation

Raise keyword

Raise is used to manually create an error in Python.

It is useful for validation.

Example:

```
score = -10
if score < 0:
    raise ValueError("Score cannot be negative")
```

Division by zero error

Dividing any number by zero is not allowed in Python.

It raises ZeroDivisionError.

Example:

```
try:
    print(15 / 0)
except ZeroDivisionError:
    print("Cannot divide by zero")
```

ValueError in Python

ValueError happens when a correct type is given but the value is wrong.

Example: converting invalid text to number.

Example:

```
try:
    num = int("hello")
except ValueError:
    print("Invalid conversion")
```

Difference between TypeError and ValueError

• TypeError → wrong data type used
• ValueError → correct type but invalid value

Example:

```
# TypeError
print("5" + 10)

# ValueError
int("abc")
```

User Interaction

Taking input from user

We use input() to get data from users.

It always returns a string by default.

Example:

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

Converting input safely to number

User input may be wrong, so we use try-except to handle errors.

Example:

```
try:
    age = int(input("Enter age: "))
    print("Age is:", age)
except ValueError:
    print("Please enter a valid number")
```

String formatting methods

Python provides different ways to format strings:

- % formatting (old style)
- .format() method
- f-strings (modern and best method)

Example:

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

MCQ's

Which block is executed if no exception occurs in try?

A. except
B. finally
C. else ✅
D. error

What is the purpose of the else block in exception handling?

A. Runs when error occurs
B. Runs when no error occurs in try ✅
C. Always runs
D. Stops program

What will be the output of:

```
try:
    x = 10
except:
    print("Error")
```

```
else:
    print("Success")
```

A. Error
B. Success ✅
C. Nothing
D. Exception

Which error occurs when dividing by zero?

A. ValueError
B. ZeroDivisionError ✅
C. TypeError
D. NameError

What does input() function return?

A. Integer
B. Float
C. String ✅
D. Boolean

Which keyword is used to manually trigger an error?

A. throw
B. raise ✅
C. error
D. except

What will happen if no exception is handled?

A. Program ignores error
B. Program continues
C. Program crashes with error message ✅
D. Program restarts

Which error occurs when converting invalid string to int?

A. TypeError
B. ValueError ✅

C. NameError
D. SyntaxError

What is the output of:

```
print(int("10"))
```

A. "10"
B. 10 ✅
C. Error
D. None

Which block always executes in exception handling?

A. try
B. except
C. finally ✅
D. else

What happens if multiple except blocks are present?

A. All run
B. Only matching one runs ✅
C. None run
D. Program stops

What is the result of:

```
try:
    print(5/0)
except:
    print("Handled")
```

A. 0
B. Handled ✅
C. Error
D. None

Which exception is raised when file is not found?

A. NameError
B. FileNotFoundError ✅

C. ValueError
D. IndexError

What is the purpose of exception handling?

A. Make program slower
B. Stop program always
C. Prevent program crash and handle errors gracefully ✅
D. Remove errors permanently

Which is the safest way to handle unknown errors?

A. except ValueError
B. except TypeError
C. except Exception ✅
D. except NameError

Chapter 11: Virtual Environments and File Handling

Environment Setup

Virtual environment in Python and its use

A virtual environment in Python is a separate workspace where we can install libraries and packages without affecting the main Python installation.

It is used to:
• Keep project dependencies separate
• Avoid version conflicts between projects
• Maintain a clean development setup

Each project gets its own "mini Python system".

Example:

```
# Creating environment
python -m venv project_env
```

Creating and activating virtual environment

We first create a virtual environment and then activate it depending on the system.

Create:

```
python -m venv myenv
```

Activate (Windows):

```
myenv\Scripts\activate
```

Activate (Mac/Linux):

```
source myenv/bin/activate
```

After activation, installed packages stay inside this environment only.

Use of requirements.txt

requirements.txt stores all required libraries of a project with versions.

It helps others run the same project without missing packages.

Install from file:

```
pip install -r requirements.txt
```

Example content:

```
flask==2.0.1
requests==2.28.1
```

Importance of file handling

File handling is important because programs need to store and retrieve data.

It is used in:
• Saving user data
• Logging errors
• Reading configuration files
• Storing results

Simple idea: file handling helps programs remember information permanently.

Example:

```
with open("report.txt", "w") as f:
    f.write("Exam results stored successfully")
```

File opening modes in Python

Python provides different modes to open files:

• r → read file
• w → write (overwrite)
• a → add content
• x → create new file
• b → binary mode
• t → text mode

Simple idea: mode decides how file is used.

Example:

```
file = open("notes.txt", "w")
file.close()
```

Safe file handling using with

Safe file handling using with automatically closes the file after use.

It prevents file corruption and errors.

Example:

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

File Operations

Reading a file

We use read() inside a file to get its content.

It reads everything at once.

Example:

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

Difference between read(), readline(), readlines()

- read() → reads full file
- readline() → reads one line
- readlines() → reads all lines as list

Example:

```
with open("data.txt", "r") as f:
    print(f.readline())
```

Writing to a file in Python

We use write() to store data in a file.

If file does not exist, Python creates it.

Example:

```
with open("output.txt", "w") as f:
    f.write("Python file handling example")
```

Appending data to file

Appending means adding new content without removing old data.

Example:

```
with open("log.txt", "a") as f:
    f.write("New entry added\n")
```

Difference between w and a

• w → deletes old content and writes new data
• a → keeps old data and adds new data

Simple idea:

- w = replace
- a = add

Example:

```
with open("file.txt", "w") as f:
    f.write("Fresh data")
```

Reading and writing together (r+)

r+ mode allows both reading and writing in the same file.

Example:

```
with open("notes.txt", "r+") as f:
    content = f.read()
    f.write("\nAdded new content")
```

File System Management

Checking if file exists

We use os.path.exists() to check file availability.

Example:

```
import os
if os.path.exists("sample.txt"):
    print("File is present")
else:
    print("File missing")
```

Deleting a file

We use os.remove() to delete files.

Always check before deleting.

Example:

```
import os
os.remove("old_file.txt")
```

FileNotFoundError handling

If file does not exist, Python raises an error.

We handle it using try-except.

Example:

```
try:
    open("missing.txt", "r")
except FileNotFoundError:
    print("File does not exist")
```

MCQ's

What is the main purpose of a virtual environment in Python?

A. To speed up execution
B. To isolate project dependencies ✅
C. To delete unused packages
D. To compile Python code

Which folder is automatically created when a virtual environment is made?

A. lib
B. env/venv folder structure ✅
C. python_packages
D. scripts only

Which command is used to check installed packages in a virtual environment?

A. pip show
B. pip list ✅
C. pip check
D. pip showall

What does pip freeze do?

A. Deletes all packages
B. Shows installed packages with versions ✅
C. Installs new packages
D. Activates environment

Which file is used to recreate the same environment on another system?

A. env.py
B. setup.txt
C. requirements.txt ✅
D. packages.json

What does read() function do in file handling?

A. Reads only first line
B. Reads entire file content ✅
C. Writes file
D. Deletes file

Which method returns file content as a list of lines?

A. read()
B. readline()
C. readlines() ✅
D. splitlines() only

What happens when a file is opened in 'a' mode?

A. File is deleted first
B. Data is overwritten
C. Data is added at end of file ✅
D. File becomes read-only

Which mode is used for both reading and writing?

A. r
B. w
C. r+ ✅
D. a

What is the benefit of using with open()?

A. Faster execution
B. Automatic file closing ✅
C. No errors occur
D. File encryption

What does file.close() do?

A. Deletes file
B. Saves file permanently
C. Releases system resources ✅
D. Opens file again

What will happen if you write to a file in 'r' mode?

A. File is created
B. Error occurs ✅
C. File is overwritten
D. File is appended

Which function is used to check if a file exists?

A. os.check()
B. os.exists()

C. file.exists()
D. os.path.exists() ✅

Which module is used for file operations in Python?

A. math
B. os and built-in open() system ✅
C. random
D. json

What happens if you open a file in 'w' mode and it already exists?

A. Nothing changes
B. Error occurs
C. File content is erased and overwritten ✅
D. File becomes read-only

References

Barry, P. (2016). Head first Python: A brain-friendly guide (2nd ed.). O'Reilly Media.

Lutz, M. (2013). Learning Python (5th ed.). O'Reilly Media.

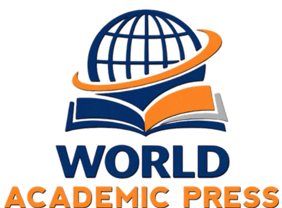
Matthes, E. (2023). Python crash course: A hands-on, project-based introduction to programming (3rd ed.). No Starch Press.

Sweigart, A. (2019). Automate the boring stuff with Python: Practical programming for total beginners (2nd ed.). No Starch Press.

Zelle, J. M. (2016). Python programming: An introduction to computer science (3rd ed.). Franklin, Beedle & Associates.

PYTHON
SKILL BUILDER
SERIES 1

This book offers a clear and simple collection of frequently asked Python programming questions with easy-to-understand answers. It is designed to help readers build strong programming basics, improve their logical thinking, and gain coding confidence. The content is carefully organized into eleven chapters that guide you step-by-step from basic Python concepts to more advanced topics. Each chapter includes short explanations, helpful examples, and multiple-choice questions to test test your learning. It is the perfect practical guide for beginners, students, and job seekers who want to prepare successfully for technical interviews and academic exams.



World Academic Pres, Kolkata, India
www.worldacademic.press