

CONTENTS

Chapter 1: Introduction to python.....	10
1.1 History of Python /Introduction to Python	10
1.2 What is Python?	10
1.3 Python File Extension	11
1.4 Python is useful because:.....	11
1.5 Main Features of Python	12
1.6 Applications of python.....	12
1.7 Python Version	13
1.8 How Python Code Runs	13
1.9 Compiler vs Interpreter Basics	13
1.10 PYTHON INSTALLATION	14
1.11 Python Syntax Basics.....	16
Chapter2: Python Fundamentals	18
2.1 Comments	18
2.2 Variables	19
2.3 Keywords.....	21
2.4 Input & Output	22
2.5 Data Types.....	25
5.1 Numeric Data Types	27
5.2 String (str)	28
5.3 List (list)	28
5.4 Tuple (tuple)	28
5.5 Set (set)	29
5.6 Dictionary (dict)	29
5.7 Range (range)	30
5.8 Boolean (bool)	30
5.9 None (NoneType)	30
2.6 Type Conversion	31
2.7 Mutable and Immutable Data Types	31
2.8 Operators	33
8.1	33

Arithmetic Operators.....	33
8.2 Comparison Operators	36
8.3 Logical Operators	39
8.4 Assignment Operators	41
8.5 Membership OPERATORS' Membership	43
8.6 Identity Operators.....	44
8.7 Bitwise Operators	45
2.9 Conditional Statements.....	48
9.1 if Statement	49
9.2 if-else Statement	49
9.3 if-elif-else Statement	49
9.4 Nested if	50
2.10 Loops.....	50
10.1 for Loop.....	51
10.2 while Loop.....	52
2.11 Loop Control Statements	52
11.1 Break	52
11.2 continue.....	52
11.3 pass.....	53
2.12 Basic Python Programs	53
Chapter 3: String Manipulation.....	56
3.1 String Basics.....	56
3.2 Indexing & Slicing	56
3.3 String Methods	57
3.4 String Formatting.....	57
3.5 Escape Characters	58
3.6 String Validation	58
3.7 Regular Expressions (Basics)	59
Summary	59
Chapter 4: Data Structures in Python.....	61
Introduction	61
4.1 Lists	61
1.1 Introduction to Lists	61
1.2Creating Lists	62

1.3 Accessing List Elements	62
1.4 Indexing	62
1.5 Slicing.....	63
1.6 Updating Lists	63
1.7 Adding Elements.....	64
1.8 REMOVING Elements.....	64
1.9 List Methods.....	66
1.10 Nested Lists	66
1.11 List Comprehension.....	67
1.12 Copying Lists	67
1.13 Sorting Lists.....	67
1.14 Iterating Through Lists	68
4.2. Tuples	68
2.1 Introduction to Tuples	68
2.2 Creating Tuples	68
2.3 Tuple Indexing	69
2.4 Tuple Slicing.....	69
2.5 Tuple Methods.....	69
2.6 Packing and Unpacking.....	69
2.7 Nested Tuples	70
2.8 Advantages of Tuples.....	70
4.3. Sets	70
3.1 Introduction to Sets	70
3.2 Creating Sets.....	71
3.3 Set Methods	71
3.4 Adding & Removing Elements	71
3.5 Union	72
3.6 Intersection.....	72
3.7 Difference	72
3.8 Symmetric Difference	72
3.9 Frozen Set	73
3.10 Set Comprehension	73
4. 4 Dictionaries	73
4.1 Introduction to Dictionaries.....	73
4.2 Creating Dictionaries	73
4.3 Accessing Values.....	74

4.4 Adding & Updating Data	74
4.5 Removing Items	74
4.6 Dictionary Methods.....	75
4.7 Nested Dictionaries	75
4.8 Dictionary Comprehension	76
4.9 Iterating Through Dictionaries.....	76
Chapter 5: Functions	79
5.1 Introduction to Functions	79
1.1 Advantages of Functions	79
5.2 Defining and Calling Functions	79
2.1 Defining a Function	79
2.2 Calling a Function	80
2.3 return Statement	80
5.3 Types of Functions.....	80
3.1 Built-in Functions	80
3.2 User-Defined Functions	80
3.3 Recursive Functions	80
3.4 Lambda Functions	80
5.4 Return Values	80
4.1 Single Value	80
4.2 Multiple Values	81
5.5 Function Arguments	81
5.1 Positional Arguments	81
5.2 Keyword Arguments	81
5.3 Default Arguments.....	81
5.4 Variable -Length Arguments(*args)	81
5.5 Keyword Variable -Length Arguments (**kwargs)	81
5.6 Variable Scope	82
6.1 Local Variables.....	82
6.2 Global Variables	82
5.7 Anonymous (Lambda) Functions.....	83
7.1 What is a Lambda Function?	83
7.2 Syntax:	83
7.3 Lambda with Single Argument	84
7.4 Lambda with Multiple Arguments	84

7.5 Lambda in Conditional Expressions	84
7.6 Lambda with map()	85
7.7 Lambda with filter()	85
7.8 Lambda with REDUCE ()	85
7.9 Sorting Using Lambda	86
7.10 Practical Use Cases	86
5.8 Recursive Functions	87
8.1 What is Recursion?	87
8.2 Recursive Function Syntax	87
8.3 Base Case	88
8.4 Recursive Case.....	88
8.5 Advantages and Limitations	89
8.6 Practical Examples	89
Chapter 6: Exception Handling & File Handling	92
6.1 Exception Handling	92
1.1 Introduction to Exception Handling	92
1.2 Handling Exceptions	94
1.3 Exception Clauses	96
1.4 Custom Exception Handling	98
6.2 File Handling	100
2.1 Introduction to File Handling	100
2.2 File Operations	103
2.3 Closing Files.....	109
Chapter7: Object-Oriented Programming (OOP)	111
7.1 Introduction to Object-Oriented Programming	111
1.1 What is Object-Oriented Programming (OOP)?	111
1.2 Need for OOP	115
Key Components of OOP'S	115
1.3 Features of OOP	115
1.4 Advantages of OOP	116
1.5 Applications of OOP.....	116
7.2 Procedural Programming vs Object-Oriented Programming.....	116
2.1 Procedural Programming	116
2.2 Object-Oriented Programming.....	117
2.3 Difference Between Procedural and OOP.....	117

2.4 Real-World Examples.....	117
7.3 Classes and Objects.....	118
3.1 What is a Class?	118
3.2 What is an Object?.....	118
3.3 Creating a Class	120
3.4 Creating Objects.....	120
3.5 Accessing Class Members	120
3.6 Creating Multiple Objects.....	121
3.7 Object Lifecycle.....	121
7.4 Attributes and Methods	121
7.5 Methods	122
5.1 Instance Methods	123
5.2 Class Methods — @classmethod	123
5.3 Static Methods — @staticmethod	123
7.6 Special Keywords	123
6.1 self Keyword.....	123
6.2 cls Keyword.....	124
7.7 Constructors	124
7.1 Introduction to Constructors.....	124
7.2 Purpose of Constructors	124
7.3 __init__() Method	125
7.4 Default Constructor	125
7.5 Parameterized Constructor	125
7.6 Constructor with Default Arguments.....	126
7.7 Constructor Chaining Using super()	126
7.8 Inheritance	126
8.1 Introduction to Inheritance.....	126
8.2 Benefits of Inheritance	127
7.9 Types of Inheritance	127
9.1 Single Inheritance	128
9.2 Multiple Inheritance.....	128
9.3 Multilevel Inheritance	129
9.4 Hierarchical Inheritance.....	130
9.5 Hybrid Inheritance	131
7.10 Related Concepts	132

10.1 Method Overriding	132
10.2 super() Method	132
10.3 Method Resolution Order (MRO)	133
7.11 Polymorphism	133
11.1 Introduction to Polymorphism	133
11.2 Method Overriding — Run-Time Polymorphism	134
11.3 Method Overloading Using Default Arguments.....	134
11.4 Duck Typing.....	135
11.5 Operator Overloading.....	136
7.12 Encapsulation	137
12.1 Introduction to Encapsulation	137
12.2 Public Members.....	138
12.3 Protected Members (..)	138
12.4 Private Members (..)	139
12.5 Access Specifiers in Python	139
12.6 Getter Methods.....	140
12.7 Setter Methods	140
12.8 Property Decorator — @property	141
7.13 Abstraction	142
13.1 What is Abstraction?	142
13.2 Why Abstraction?	142
13.3 Abstract Classes	142
13.4 Abstract Methods	143
13.5 abc Module	143
13.6 ABC Class	144
13.7 @abstractmethod Decorator.....	144
Chapter 8: Modules & Packages	147
8.1 Introduction to Modules	147
1.1 What is a Module?	147
1.2 Why Use Modules?	147
1.3 Types of Modules.....	148
8.2 Importing Modules	149
2.1 import Statement	149
2.2 from ... import	149
2.3 import ... as	150
2.4 Importing Multiple Modules.....	150

2.5 Reloading Modules	150
2.6 Module Search Path (sys.path)	150
8.3 Creating User-Defined Modules	151
3.1 Creating a Module	151
3.2 Using Custom Modules	151
3.3 Organizing Code into Modules	152
3.4 Best Practices for Modules	152
8.4 Packages	153
4.1 What is a Package?	153
4.2 Creating Packages	153
9.3 Package Structure	154
4.4 __init__.py	154
4.5 Importing from Packages	155
4.6 Advantages of Packages	155
8.5 Built-in Python Libraries	155
math Module	156
5.2 random Module	158
5.3 datetime Module	159
5.4 sys Module	161
5.5 statistics Module	162
Chapter 9: External Python Libraries	166
9.1 NumPy – Numerical Computing	166
1.1 Arrays	167
1.2 Array Operations	167
1.3 Mathematical Functions	168
1.4 Statistical Functions	168
1.5 Broadcasting	169
9.2 Pandas – Data Analysis	170
2.1 Series	170
2.2 Data Frame	171
2.3 Data Import & Export	171
2.4 Data Cleaning	172
2.5 Filtering	172
2.6 Group By	173
2.7 Merge & Join	173

9.3 Matplotlib – Data Visualization	174
3.1 Line Chart	175
3.2 Bar Chart.....	175
Scatter Plot.....	176
3.4 Histogram	176
3.5 Pie Chart	176
3.6 Box Plot.....	177
9.4 Seaborn – Statistical Visualization.....	177
4.1 Count Plot.....	177
4.2 Heatmap	178
4.3 Pair Plot	178
4.4 Distribution Plot	179
4.5 Violin Plot.....	179
9.5 Scikit-learn – Machine Learning.....	179
5.1 Data Preprocessing	180
5.2 Train-Test Split	180
5.3 Feature Scaling.....	181
5.4 Building ML Models.....	181
5.5 Model Evaluation	182



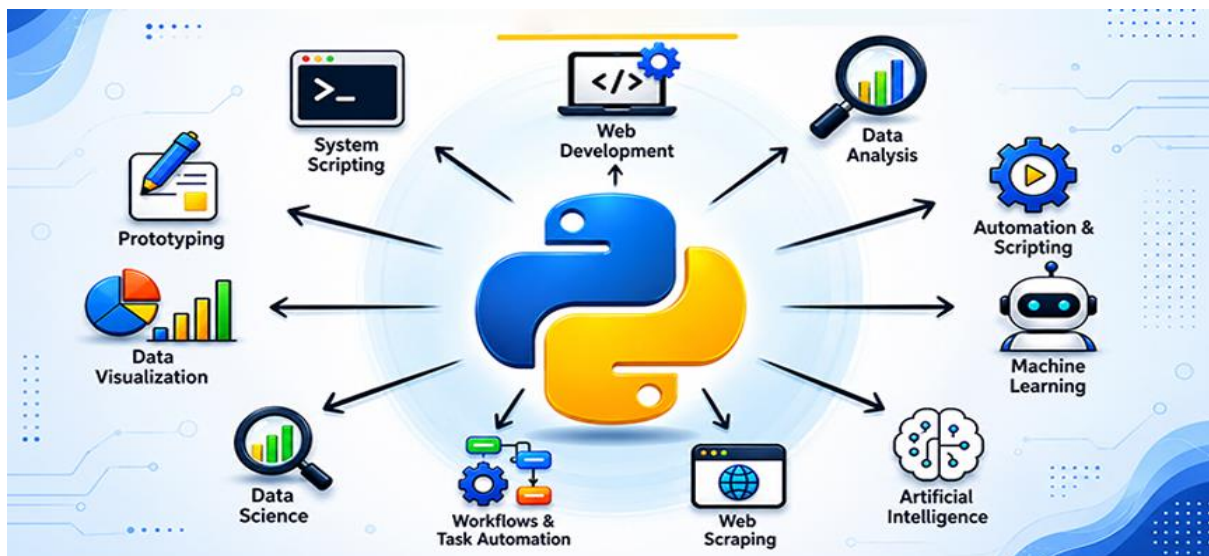
1.1 HISTORY OF PYTHON /INTRODUCTION TO PYTHON

Python was created by **Guido van Rossum**. He started working on Python in the late 1980s, and **the first version was released in 1991**.

Python was designed to be simple, readable, and easy to use. Guido van Rossum wanted programmers to focus more on solving problems rather than dealing with complicated syntax.

Interestingly, **the name Python does not come from the snake**. It was inspired by the **British comedy show “Monty Python’s Flying Circus.”**

Today, Python is widely used in:



Python is popular because **it is easy for beginners to learn** and powerful enough for professional applications.

1.2 WHAT IS PYTHON?

Python is a high-level, general-purpose programming language. It is used to develop programs for many different purposes, including:

- Automation
- Web Development
- Data Handling and Data Analysis

- Artificial Intelligence
- Machine Learning
- Scripting
- Testing
- Backend Development

Python is popular because its syntax is simple, easy to read, and easy to understand.

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

Output: Hello, World!

1.3 PYTHON FILE EXTENSION

Python programs are saved with the **.py file extension**.

Examples:

- **main.py**
- **app.py**
- **hello.py**

In simple words, Python is an easy-to-learn and powerful programming language used for building a wide variety of applications.

Why Learn Python?

Python is easy to start with and powerful enough to build real-world applications and projects. It is one of the most popular programming languages because it is simple, versatile, and widely used.

1.4 PYTHON IS USEFUL BECAUSE:

- **Beginner-friendly:** Its simple syntax makes it easy to learn and understand.
- **Simple and readable:** Python code is clean and easy to read.
- **Used in many industries:** It is used in technology, finance, healthcare, education, and many other fields.
- **Many libraries:** Python provides a large number of libraries and tools that make development easier.
- **Good for automation:** It can automate repetitive tasks and save time.

- **Useful for Data Science:** Libraries such as Pandas and NumPy are widely used for data analysis.
- **Used in AI and Machine Learning:** Python supports AI and ML development with libraries such as scikit-learn.
- **Web and Backend Development:** Python can be used to build websites, web applications, and backend systems.

Python is a great language for beginners and also powerful enough for professionals to build real-world applications.

1.5 MAIN FEATURES OF PYTHON

Feature	Meaning
Easy syntax	Python code is simple and readable
High-level	We do not need to deal directly with machine-level details
Interpreted	Python code runs using the Python interpreter
Dynamically typed	No need to declare variable type manually
Object-oriented	Supports classes and objects
Cross-platform	Runs on Windows, macOS, and Linux
Large library support	Many built-in and external libraries are available
Open-source	Free to use

1.6 APPLICATIONS OF PYTHON

Application	Use of Python
Web Development	Building websites and web applications using frameworks like Django and Flask
Data Analysis	Analysing, cleaning, and processing data using Pandas and NumPy.
Data Visualization	Creating charts and graphs using Matplotlib.
Artificial Intelligence	Developing intelligent systems and AI applications.
Machine Learning	Building prediction and classification models using libraries such as scikit-learn.

Automation	Automating repetitive tasks such as file handling, emails, and data processing.
Backend Development	Creating server-side applications and APIs.
Software Development	Developing desktop applications and software.
Testing	Writing automated tests to test software applications.
Web Scraping	Extracting data from websites.
Scripting	Writing small programs to perform specific tasks.
Game Development	Creating simple games and interactive applications.
Cybersecurity	Building security tools and analysing systems.
Internet of Things (IoT)	Programming devices, sensors, and smart systems.

1.7 PYTHON VERSION

For modern learning, use Python 3. Python 2 is old and should not be used for new learning.

To check Python version,
use: `python --version` or: `python3 --version`
output: Python 3.12.5

1.8 HOW PYTHON CODE RUNS

Basic flow: Python Code → Python Interpreter → Output

More accurate flow: Source Code → Bytecode → Python Virtual Machine → Output

For now, remember:

- Python code is written in .py files.
- Python interpreter reads and runs the code.
- Python runs code from top to bottom.
- Output is shown on the screen

1.9 COMPILER VS INTERPRETER BASICS

Before code runs, it must be translated into a form the computer can understand.

There are two common translators:

- **Compiler**
- **Interpreter**

Compiler: A compiler translates the whole program before running it.

Flow: Source Code → Compiler → Executable File → Run Program

Examples of compiled languages:

- **C**
- **C++**

Interpreter: An interpreter runs code more directly.

Flow: Source Code → Interpreter → Output

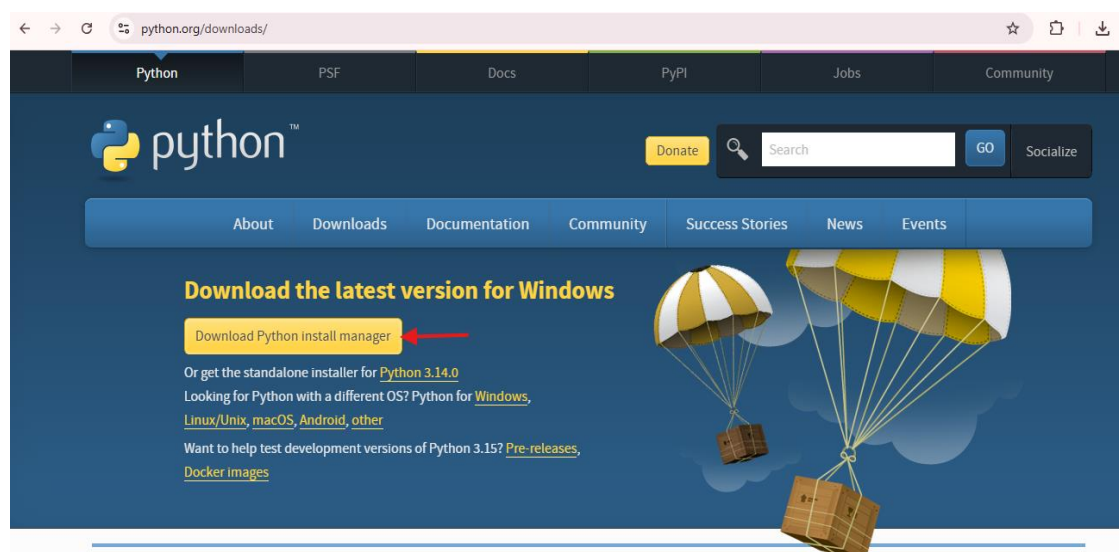
Python is commonly called an interpreted language

1.10 PYTHON INSTALLATION

The first two phases partially use ETL (Extract, Transform and Load) to handle the data. Let us take a look at these phases one by one:

Step 1: Download Python

1. Go to the official website → <https://www.python.org/downloads/>



2. Click on “**Download Python 3.x.x**” (latest stable version)

Looking for a specific release?

Python releases by version number:

Release version	Release date	Click for more	
Python 3.9.25	Oct. 31, 2025	Download	Release Notes
Python 3.13.9	Oct. 14, 2025	Download	Release Notes
Python 3.11.14	Oct. 9, 2025	Download	Release Notes
Python 3.10.19	Oct. 9, 2025	Download	Release Notes
Python 3.9.24	Oct. 9, 2025	Download	Release Notes
Python 3.12.12	Oct. 9, 2025	Download	Release Notes
Python 3.14.0	Oct. 7, 2025	Download	Release Notes
Python 3.13.8	Oct. 7, 2025	Download	Release Notes

[View older releases](#)

Step 2: Run the Installer

1. Locate the downloaded file (e.g., python-3.12.0.exe).
2. IMPORTANT → Before clicking Install Now, check the box: “Add Python to PATH”
3. Click Install Now.

Step 3: Verify Installation

1. Open **Command Prompt** (cmd) and type:

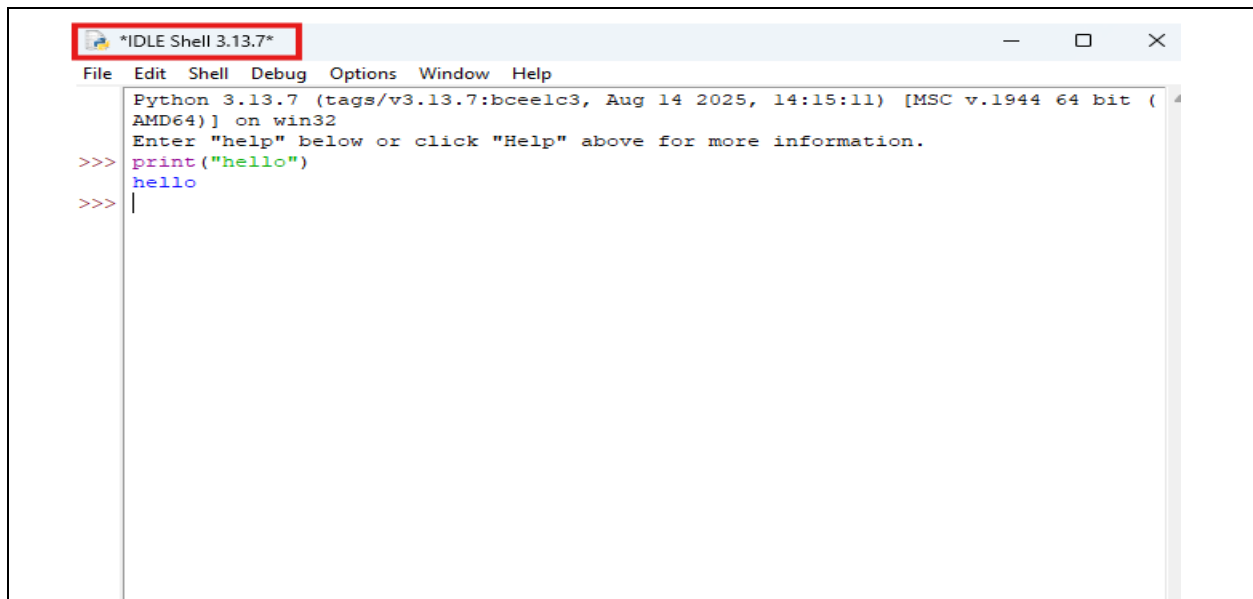
```
Administrator: C:\WINDOWS\
Microsoft Windows [Version 10.0.26200.6899]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Administrator>python --version
Python 3.13.7

C:\Users\Administrator>where python
C:\Users\Administrator\AppData\Local\Programs\Python\Python313\python.exe
C:\Users\Administrator\AppData\Local\Microsoft\WindowsApps\python.exe

C:\Users\Administrator>
```

2. After that we can code in IDLE Python



```
Python 3.13.7 (tags/v3.13.7:bceelc3, Aug 14 2025, 14:15:11) [MSC v.1944 64 bit (AMD64)] on win32
Enter "help" below or click "Help" above for more information.
>>> print('hello')
hello
>>> |
```

1.11 PYTHON SYNTAX BASICS

Python syntax means the set of rules used to write Python programs.

Just like English has grammar, Python also has grammar.

If we do not follow Python syntax rules, the program gives an error Python is beginner-friendly because its syntax is simple, readable, and close to normal English

Eg. `print("Hello, World!")`

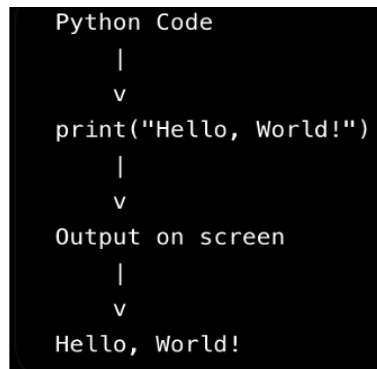
Output: Hello, World!

First Python Code

The first Python program usually starts with printing a message on the screen

Part	Meaning
<code>print</code>	Built-in Python function used to display output
<code>()</code>	Parentheses used to pass data to the function
<code>"Hello, World!"</code>	Text value, also called a string

Diagram:



- **Eg.**
- Print ("Welcome to Python")
Print ("Python is easy to learn")

Output:

Welcome to Python
Python is easy to learn

Each print () statement displays output on a new line by default



2.1 COMMENTS

Comments are notes written in Python code to explain what the code does. Comments are mainly used to make programs easier to read and understand.

Python ignores comments when running the program.

Single-Line Comment

A single-line comment starts with the # symbol

```
# This is a comment
```

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

You can also write a comment after a statement:

```
print ("Hello, World!") # Display a message
```

Multi-Line Comments

Python does not have a special multi-line comment symbol. However, multiple lines can be commented using #.

```
This is the first line
```

```
# This is the second line
```

```
# This is the third line
```

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

You may also see triple quotes used for multi-line notes:

```
"""
```

```
This is a multi-line
```

```
text block.
```

```
"""
```

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

Eg.

```
# Program to add two numbers
```

```
num1 = 10
```

```
num2 = 20
```

```
# Calculate the sum
```

```
sum = num1 + num2
```

```
print(sum)
```

Output: 30

Comments help programmers understand the purpose and logic of a program.

Practice Questions

1. What is a comment in Python?
2. Why do we use comments in Python programs?

3. How do you write a single-line comment in Python?
4. How do you write a multi-line comment in Python?
5. Write a Python program using comments to explain the code.

2.2 VARIABLES

A variable is a name used to store data in a program. You can think of a variable as a container that holds a value.

Creating a Variable

In Python, you create a variable using the assignment operator =.

```
name = "Rahul"
age = 20
Here:
name stores " Rahul "
age stores 20
```

Displaying Variables

Use the print () function to display a variable's value.

```
name = " Rahul "
age = 20
print(name)
print(age)
Output:
Rahul
20
```

Different Types of Variables

Python can store different types of data:

```
name = "Rahul"    # String
age = 20          # Integer
salary = 50000.50 # Float
is_student = True  # Boolean
```

Python is Dynamically Typed

You do not need to declare the data type separately.

```
x = 10
print(x)
x = "Python"
print(x)
```

Python automatically determines the data type.

Variable Naming Rules

A variable name must start with a letter or underscore _.

It cannot start with a number.

It can contain letters, numbers, and underscores.

Spaces are not allowed.

Python keywords cannot be used as variable names.

Variable names are case-sensitive.

Valid Variable Names

```
name = "Rahul"
```

```
_age = 20
```

```
student1 = "Rahul"
```

```
total_marks = 90
```

Invalid Variable Names

```
1name = "Amit"    # Cannot start with a number
```

```
student name = "Rahul" # Spaces are not allowed
```

```
class = "Python"   # Keyword cannot be used
```

Assigning Multiple Variables

```
name, age, city = "Amit", 25, "Pune"
```

```
Same value
```

```
x = y = z = 100
```

Changing a Variable Value

A variable's value can be changed:

```
age = 25
```

```
age = 26
```

```
print(age)
```

Output: 26

A variable is a named container used to store data that can be used and changed during program execution

Practice Questions

1. What is a variable in Python?
2. Why do we use variables?
3. Write a program to create a variable named name and store your name.
4. Create variables to store your name, age, and city.
5. Write a program to print the value of a variable.
6. Create two variables and add their values.

7. Write a program to change the value of an existing variable.
 8. Create multiple variables in a single line.
 9. Assign the same value to multiple variables.
 10. Use the type () function to check the data type of a variable.
 11. Write a program to swap the values of two variables.
 12. Create variables using valid variable names.
 13. Identify whether the following are valid or invalid variable names:
 - age
 - 2name
 - student_name
 - class
 - _salaryCreate a variable and update its value using +=.
 14. Write a program to store and print your name, age, and salary using variables.
-

2.3 KEYWORDS

Keywords are reserved words in Python. They already have a special meaning and purpose, so they cannot be used as variable names, function names, or class names.

Some Common Python Keywords

Keyword	When we use it
if	To check a condition
else	To execute code when the if condition is false
for	To repeat code for a sequence
while	To repeat code while a condition is true
class	To create a class
def	To create a function
return	To return a value from a function
True	Represents a true Boolean value
False	Represents a false Boolean value
None	Represents no value or an empty value
import	To import a module or library
try	To handle possible errors
except	To handle an error

with	To work with resources such as files
as	To create an alias or reference
continue	To skip the current iteration and move to the next

Example:

```
age = 20
if age >= 18:
    print("Eligible")
else:
    print ("Not Eligible")
Output: Eligible
```

Practice Questions

1. What are keywords in Python?
 2. Why cannot keywords be used as variable names?
 3. What is the purpose of the if keyword?
 4. What is the difference between if and else?
 5. When do we use the for keyword?
 6. When do we use the while keyword?
 7. What is the purpose of the def keyword?
 8. When do we use the class keyword?
 9. What is the difference between break and continue?
 10. What are True, False, and None in Python?
-

2.4 INPUT & OUTPUT

Print Statement (Output ())

The print () function is used to display output on the screen.

Syntax: print(value)

1: Print Text

```
print ("Hello, World!")
Output: Hello, World!
```

2: Print a Number

```
print (100)
Output: 100
```

3: Print a Variable

```
name = "Amit"
age = 25
```

```
print(name)
```

```
print(age)
```

Output:

```
Amit
```

```
25
```

4: Print Multiple Values

```
name = " Amit"
```

```
age = 25
```

```
print (name, age)
```

Output: Amit 25

5: Using sep

The sep parameter specifies the separator between multiple values.

```
print ("Python", "Java", "C++", sep=" - ")
```

Output: Python - Java - C++

6: Using end

By default, print () moves to a new line after displaying output. The end parameter changes this behaviour.

```
print ("Hello", end=" ")
```

```
print("World")
```

Output: Hello World

print () is used to display text, numbers, variables, and other results as output in Python.

Input from the User

we use the input () function to take input from the user.

Syntax: input("message")

1: Taking Name as Input

```
name = input ("Enter your name: ")
```

```
print ("Hello", name)
```

Output:

```
Enter your name: Rahul
```

```
Hello Rahul
```

2: Taking Age as Input

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

```
print(age)
```

By default, input () takes the value as a string.

3: Taking Integer Input

To convert user input into an integer, use int ().

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

```
print(age)
```

4: Taking Float Input

```
salary = float (input ("Enter your salary: "))  
print(salary)
```

5: Adding Two Numbers from User Input

```
num1 = int (input ("Enter first number: "))  
num2 = int (input ("Enter second number: "))  
sum = num1 + num2  
print ("Sum =", sum)
```

Output:

Enter first number: 10

Enter second number: 20

- Sum = 30

input ()

always returns a **string**. Use type conversion when you need numbers:

int () → Integer

float () → Decimal number

str () → String

The input () function allows a Python program to receive information directly from the user.

Basic Python Program Structure

A basic Python program usually follows a simple structure:

Comments – Describe the purpose of the program.

Input – Take data from the user, if needed.

Processing – Perform calculations or operations.

Output – Display the result.

Example:

```
# Program to add two numbers  
# Input  
num1 = int (input ("Enter first number: "))  
num2 = int (input ("Enter second number: "))  
# Processing  
sum = num1 + num2  
# Output:  
print ("Sum =", sum)
```

Output:

Enter first number: 10

Enter second number: 20

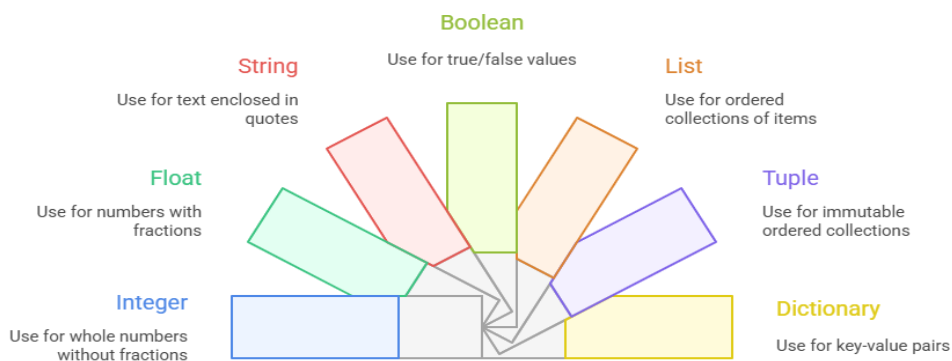
Sum = 30

Program Flow: Start → Input → Processing → Output → End

2.5 DATA TYPES

A data type tells Python what type of value is stored in a variable.

Which data type should be used for the given data?



Made with Napkin

Example:

```
name = "Amit"  
age = 25  
salary = 50000.50  
is_student = True
```

Here:

"Amit" is a string
25 is an integer
50000.50 is a float
True is a Boolean

Data types help Python understand:

What kind of data is stored.
What operations can be performed.
How the data should be processed.

Example

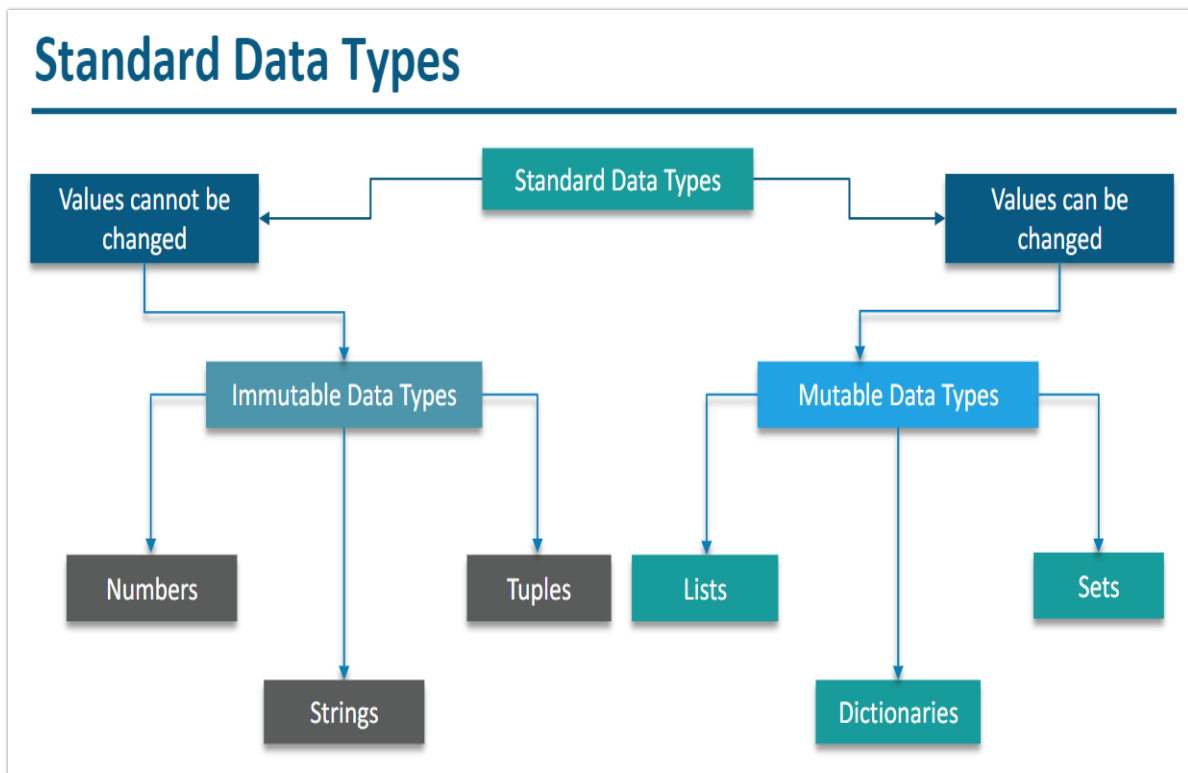
```
a = 10  
b = 20  
print(a + b)  
Output: 30
```

But:

```
a = "10"
b = "20"
print(a + b)
Output: 1020
```

The first performs addition, while the second performs string concatenation.

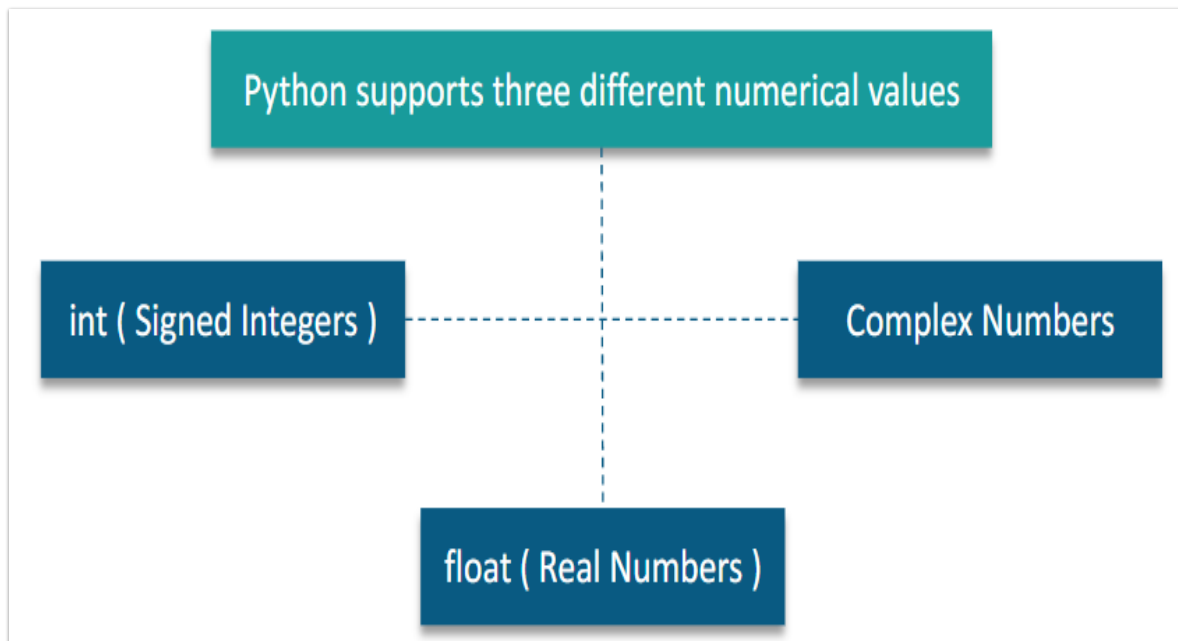
Data Types in Python



Python provides several built-in data types.

Category	Data Types
Numeric	int, float, complex
Sequence	str, list, tuple, range
Mapping	dict
Set	set, frozen set
Boolean	bool
None	None Type

5.1 NUMERIC DATA TYPES



1.1 Integer (int)

An integer is a whole number without a decimal point.

Example:

```
age = 25
temperature = -10
number = 0
print(type(age))
Output: <class 'int'>
```

1.2 Float (float)

A float is a number containing a decimal point.

Example:

```
price = 99.50
percentage = 85.5
print(type(price))
Output: <class 'float'>
```

1.3 Complex (complex)

A complex number contains a real part and an imaginary part.

Python uses j for the imaginary part.

Example:

```
number = 3 + 4j
```

```
print(number)
print(type(number))
Output:
(3+4j)
<class 'complex'>
```

5.2 String (STR)

A string is a sequence of characters.

Example:

```
name = "Amit"
city = "Pune"
```

Strings can contain:

```
Letters
Numbers
Symbols
Spaces
message = "Welcome to Python"
```

Strings are written inside:

```
"Python"
'Python'
```

5.3 LIST (LIST)

A list is an ordered collection of items.

Lists are:

```
Ordered
Mutable
Allow duplicate values
```

Example:

```
numbers = [10, 20, 30, 40]
print(numbers)
```

A list can store different types of data:

```
data = ["Rahul", 25, 50000.50, True]
Modifying a List
numbers = [10, 20, 30]
numbers[0] = 100
print(numbers)
Output: [100, 20, 30]
```

5.4 TUPLE (TUPLE)

A tuple is also an ordered collection of items.

Tuples are:

- Ordered
- Allow duplicate values
- Immutable

Example:

```
numbers = (10, 20, 30)
print(numbers)
You can access tuple values using indexing:
print (numbers [0])
Output: 10
```

5.5 SET (SET)

A set is a collection of unique values.

Sets:

- Do not allow duplicates.
- Are mutable.
- Do not support **indexing**.

Example:

```
numbers = {10, 20, 30, 40}
print(numbers)
```

Duplicate Example

```
numbers = {10, 20, 20, 30, 30}
print(numbers)
The duplicate values are removed.
Output: {10, 20, 30}
```

5.6 DICTIONARY (DICT)

A dictionary stores data in key-value pairs.

Example:

```
student = {
    "name": "Amit",
    "age": 25,
    "city": "Pune"
}
Accessing a Value
print(student["name"])
Output: Amit
```

Dictionaries are mutable, so their values can be changed.

```
student["age"] = 26
```

5.7 RANGE (RANGE)

The range type represents a sequence of numbers.
It is commonly used with loops.

Example:

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

Output: [1, 2, 3, 4, 5]

5.8 BOOLEAN (BOOL)

A Boolean data type has only two values:

True

False

Example:

```
is_logged_in = True
is_student = False
```

Boolean values are commonly used in conditional statements.

Example:

```
age = 20
print (age >= 18)
```

Output: True

5.9 NONE (NONETYPE)

None represents no value.

Example:

```
x = None
print(x)
print(type(x))
```

Output:

None

<class 'NoneType'>

None is different from:

Value	Meaning
0	Zero
“ ”	Empty string
False	Boolean false
None	No value

Checking Data Types Using type ()

The type () function is used to check the data type of a variable.

Example:

```
x = 10
print(type(x))
```

More Examples:

```
print (type (10))
print (type (10.5))
print(type("Python"))
print (type (True))
print (type ([1, 2, 3]))
```

2.6 TYPE CONVERSION

• Function	• Converts To
• int ()	• Integer
• float ()	• Float
• str ()	• String
• bool ()	• Boolean
• list ()	• List
• tuple ()	• Tuple
• set ()	• Set

Type conversion means changing one data type into another.

Example: String to Integer

```
x = "100"
y = int(x)
print(y)
print(type(y))
Integer to Float
x = 10
print(float(x))
Output: 10.0
```

2.7 MUTABLE AND IMMUTABLE DATA TYPES

7.1 Mutable Data Types

Mutable data can be changed after creation.

Examples:

list
set
dict

Example:

```
numbers = [10, 20, 30]
numbers [0] = 100
print(numbers)
```

7.2 Immutable Data Types

Immutable data cannot be changed after creation.

Examples:

int
float complex
str
tuple
bool
frozenset
range
NoneType

Practice Questions

1. What are data types in Python?
 2. What is the difference between int and float?
 3. Write a program to store your name, age, and salary using appropriate data types.
 4. Write a program to check the data type of a variable using type().
 5. Create a list containing five numbers.
 6. Create a tuple containing the names of three cities.
 7. Create a dictionary to store student name, age, and marks.
 8. Create a set and add some duplicate values. What happens to the duplicates?
 9. Write a program to convert an integer into a float.
 10. Write a program to convert a string "100" into an integer.
 11. Write a program to check whether a value is of type str.
 12. Create a Boolean variable and print its value.
 13. Find the length of a string using len().
 14. Create a list and access its first and last elements.
 15. Write a program to demonstrate the difference between mutable and immutable data types.
-

2.8 OPERATORS

Operators are special symbols or words used to perform operations on values and variables.

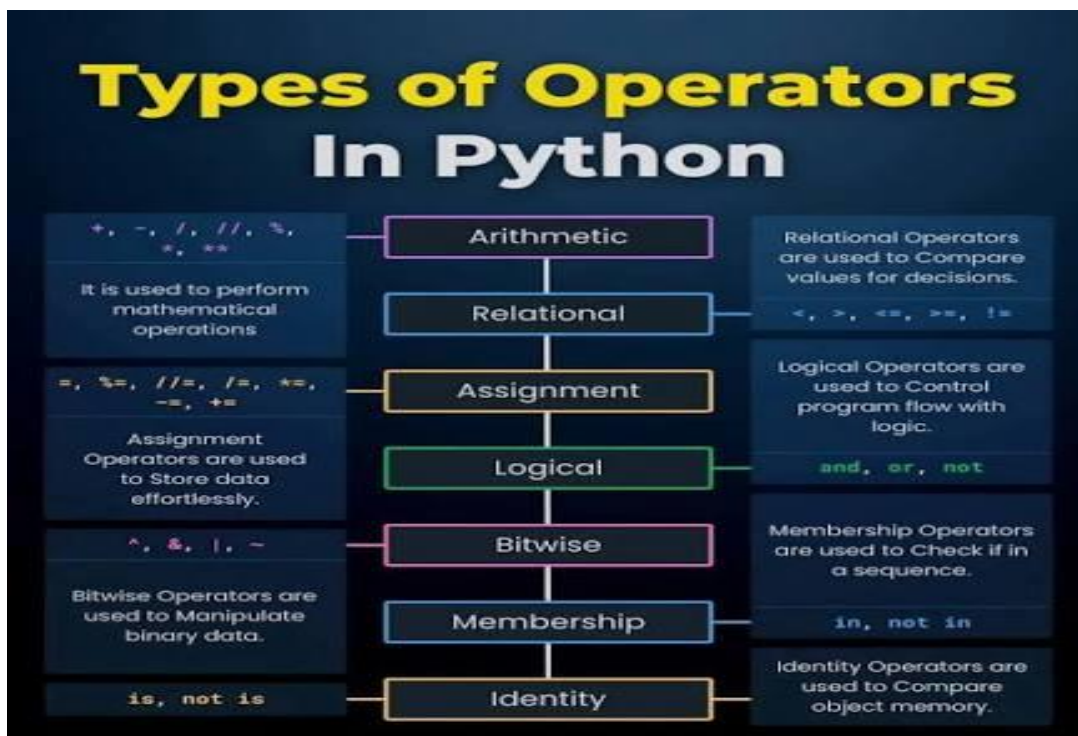
Operator = Symbol or keyword that performs an operation

Example:

```
a = 10
b = 5
print (a + b)
```

Output: 15

Python operators are mainly divided into these types:



8.1

ARITHMETIC OPERATORS

Arithmetic operators are used to perform mathematical operations.

1. Addition
2. Subtraction
3. Multiplication
4. Division
5. Power
6. Remainder

Arithmetic operators work mostly with numbers.

List of Arithmetic Operators

operator	Name	Example
+	Addition	10 + 5
-	Subtraction	10 - 5
*	Multiplication	10 * 5
/	Division	10 / 5
//	Floor division	10 // 3
%	Modulus	10 % 3
**	Exponent / Power	2 ** 3

1.1 Addition +:

The + operator adds two numbers.

Example:

```
a = 10
b = 5
result = a + b
print(result)
Output: 15 Code
```

1.2 Subtraction –

The - operator subtracts one number from another.

Example:

```
a = 10
b = 5
result = a - b
print(result)
Output: 5
```

1.3 Multiplication *

The * operator multiplies two numbers.

Example:

```
a = 10
b = 5
result = a * b
print(result)
Output: 50
```

1.4 Division /

The / operator divides one number by another. Division using / always gives a float result.

Example:

```
a = 10
b = 5
result = a / b
print(result)
print(type(result))
```

Output:

```
2.0
<class 'float'>
```

Even though $10 / 5$ is mathematically 2, Python gives 2.0. That means the result is float.

1.5 Floor Division //

The // operator divides and gives the whole-number part.

Example:

```
a = 10
b = 3
result = a // b
print(result)
```

Output: 3

Explanation: $10 / 3 = 3.333...$ Floor division gives 3 Important point: / gives normal division // gives floor division

Example:

```
print(10 / 3)
print(10 // 3)
```

Output: 3.3333333333333335 3

1.6 Modulus %

The % operator gives the remainder after division.

Example:

```
a = 10
b = 3
result = a % b
print(result)
```

Output:1

Explanation: 10 divided by 3 => 3 goes into 10 three times: $3 \times 3 = 9$

Remainder: $10 - 9 = 1$

1.7 Exponent / Power **

The ** operator is used to calculate power.

Example:

```
result = 2 ** 3  
print(result)
```

Output:8

Explanation: 2 ** 3 means 2 raised to the power 3

Example:

```
a = 20  
b = 6  
print (a + b)  
print (a - b)  
print (a * b)  
print (a / b)  
print (a // b)  
print (a % b)  
print (a ** 2)
```

Output:

```
26  
14  
120  
3.3333333333333335  
3  
2  
400
```

8.2 COMPARISON OPERATORS

Comparison operators are used to compare two values. The result of a comparison is always a Boolean value:

True

False

Example:

```
print (10 > 5)
```

Output: True Because 10 is greater than 5

List of Comparison Operators

Operator	Meaning	Example
==	Equal to	10 == 10
!=	Not equal to	10 != 5
>	Greater than	10 > 5
<	Less than	5 < 10

<code>>=</code>	Greater than or equal to	<code>10 >= 10</code>
<code><=</code>	Less than or equal to	<code>5 <= 10</code>

2.1 Equal To `==`

The `==` operator checks whether two values are equal.

Example:

```
print (10 == 10)
print (10 == 5)
```

Output:

```
True
False
```

2.2 Not Equal To `!=`

The `!=` operator checks whether two values are not equal.

Example:

```
print (10 != 5)
print (10 != 10)
```

Output:

```
True
False
```

Explanation: `10 != 5` True because 10 is not equal to 5
`10 != 10` False because 10 is equal to 10

2.3 Greater Than `>`

The `>` operator checks whether the left value is greater than the right value.

Example:

```
print (10 > 5)
print (5 > 10)
```

Output:

```
True
False
```

2.4 Less Than `<`

The `<` operator checks whether the left value is less than the right value.

Example:

```
print (5 < 10)
print (10 < 5)
```

Output:

```
True
False
```

2.5 Greater Than or Equal To >=

The >= operator checks whether the left value is greater than or equal to the right value.

Example:

```
print (10 >= 5)
print (10 >= 10)
print (5 >= 10)
```

Output:

```
True
True
False
```

Explanation: 10 >= 5 True because 10 is greater than 5 10 >= 10 True because 10 is equal to 10 5 >= 10 False

2.6 Less Than or Equal To <=

The <= operator checks whether the left value is less than or equal to the right value.

Example:

```
print (5 <= 10)
print (10 <= 10)
print (20 <= 10)
```

Output:

```
True
True
False
```

Explanation:

5 <= 10 True because 5 is less than 10
10 <= 10 True because 10 is equal to 10
20 <= 10 False Comparison with Variables

Example:

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

Output:

```
False
True
False
True
False
True
```

True
False
True

8.3 LOGICAL OPERATORS

Logical operators are used to combine Boolean values. Python has three logical operators:

and
or
not

These operators work with True and False.

List of Logical Operators

Operator	Meaning
AND	True when both sides are True
OR	True when at least one side is True
NOT	Reverses True/False

3.1 AND Operator:

The and operator gives True only when both values are True.

Example:

```
print (True and True)
print (True and False)
print (False and True)
print (False and False)
```

Output:

True
False
False
False

Truth table:

Left Value	Right Value	Result
True	True	True
True	False	False
False	True	False
False	False	False

AND means both conditions must be True.

Example:

```
age_valid = True
id_available = True
print (age_valid and id_available)
```

Output: True

3.2 or Operator:

The or operator gives True when at least one value is True.

Example:

```
print (True or True)
print (True or False)
print (False or True)
print (False or False)
```

Output:

```
True
True
True
False
```

Truth table:

Left Value	Right Value	Result
True	True	True
True	False	True
False	True	True
False	False	False

OR means at least one value must be True.

Example:

```
has_email = True
has_phone = False
print (has_email or has_phone)
```

Output: True

3.3 not Operator

The not operator reverses a Boolean value.

Example:

```
print (not True)
print (not False)
```

Output:

```
False
```

True

not True becomes False not False becomes True

Example:

```
is_logged_in = True
print (not is_logged_in)
```

Output:

False

Logical Operators with Comparisons

Comparison operators return Boolean values. So, they can be used with logical operators.

Example:

```
age = 20
marks = 85
print (age > 18 and marks > 80)
```

Output: True

Explanation:

```
age > 18    True
marks > 80   True
True and True = True
```

8.4 ASSIGNMENT OPERATORS

operators are used to assign values to variables. The most basic assignment operator is: =

Example: `x = 10`
 `print(x)`
Output: 10

Here, 10 is assigned to x.

List of Assignment Operators

Operator	Example	Same As
=	X = 10	X – 10
+=	X += 5	X = x + 5
-=	X -=5	X = x – 5
*=	X *= 5	X = x * 5
/=	X /= 5	X = x / 5
//=	X //= 5	X = x // 5
%=	X %= 5	X = x % 5

<code>**=</code>	<code>X **= 5</code>	<code>X = x ** 5</code>
------------------	----------------------	-------------------------

4.1 Basic Assignment =

Example:

```
x = 10
```

```
print(x)
```

Output: 10

4.2 Add and Assign +=

Example:

```
x = 10
```

```
x += 5
```

```
print(x)
```

Output: 15

Explanation: `x += 5` Same as: `x = x + 5`

4.3 Subtract and Assign -=

Example:

```
x = 10
```

```
x -= 3
```

```
print(x)
```

Output: 7

Explanation: `x -= 3` Same as: `x = x - 3`

4.4 Multiply and Assign *=

Example:

```
x = 10 x *= 2
```

```
print(x)
```

Output: 20

Explanation: `x *= 2` Same as: `x = x * 2`

4.5 Divide and Assign /=

Example:

```
x = 10
```

```
x /= 2
```

```
print(x)
```

Output: 5.0

Important: `/=` gives float result because `/` gives float result.

4.6 Floor Divide and Assign //=

Example:

```
x = 10
```

```
x //= 3
```

```
print(x)
```

Output: 3

Explanation: `x //= 3` Same as: `x = x // 3`

4.7 Modulus and Assign %=

Example:

```
x = 10
x %= 3
print(x)
```

Output: 1

Explanation: x %= 3 Same as: x = x % 3

4.8 Power and Assign **=

Example:

```
x = 2
x **= 3
print(x)
```

Output:8

Explanation: x **= 3 Same as: x = x ** 3

8.5 MEMBERSHIP OPERATORS' MEMBERSHIP

operators are used to check whether a value exists inside another value. Python has two membership operators:

- in
- not in

For nw, we will use membership operators with strings because strings are already covered.

in Operator

The in operator checks whether something is present.

Example:

```
text = "Python"
print ("P" in text)
print ("Py" in text)
print ("Java" in text)
```

Output:

True

True

False

Explanation:

"P" exists in "Python"	True
"Py" exists in "Python"	True
"Java" does not exist	False

not in Operator:

The not in operator checks whether something is not present.

Example:

```
ext = "Python"
print ("Java" not in text)
print ("Py" not in text)
```

Output:

```
True
False
```

Membership Is Case-Sensitive

Python checks uppercase and lowercase carefully.

Example:

```
text = "Python"
print ("P" in text)
print ("p" in text)
```

Output:

```
True
False
```

Explanation: "P" and "p" are different in Python.

8.6 IDENTITY OPERATORS

Identity operators are used to check whether two variables refer to the same object in memory.

Python has two identity operators:

- **is**
- **is not**

Important: == checks value is checks identity/memory object

is Operator

The is operator checks whether two variables point to the same object.

Example:

```
a = None
b = None
print (a is b)
```

Output: True

Explanation: a refers to None b refers to None Both refer to the same None object.

is not Operator

The is not operator checks whether two variables do not point to the same object.

Example:

```
a = None
b = 10
print (a is not b)
```

Output: True

Explanation: a refers to None b refers to 10 They are not the same object.

Difference Between == and is

== checks whether values are equal. is checks whether both variables refer to the same object.

Example:

```
a = 100
b = 100
print (a == b)
print (a is b)
```

Possible output:

True

True

For some simple values, Python may reuse the same object internally. But beginners should remember this rule: Use == for value comparison. Use is mainly with None.

Example:

```
result = None
print (result is None)
print (result is not None)
```

Output: True

False

8.7 BITWISE OPERATORS

Bitwise operators work on binary numbers. Binary means numbers written using only:

0 and 1

Computers store numbers internally in binary form.

Example:

```
Decimal 5 = Binary 101
Decimal 3 = Binary 011
```

Bitwise operators compare or shift bits.

List of Bitwise Operators

Operator	Name
<code>&</code>	Bitwise AND
<code> </code>	Bitwise OR
<code>^</code>	Bitwise XOR
<code>~</code>	Bitwise NOT
<code><<</code>	Left shift
<code>>></code>	Right shift

7.1 Bitwise AND &

Bitwise AND compares bits.

Rule:

$$1 \& 1 = 1$$

$$1 \& 0 = 0$$

$$0 \& 1 = 0$$

$$0 \& 0 = 0$$

Example:

```
a = 5
```

```
b = 3
```

```
print (a & b)
```

Output: 1

7.2 Bitwise OR |

Bitwise OR compares bits.

Rule:

$$1 | 1 = 1$$

$$1 | 0 = 1$$

$$0 | 1 = 1$$

$$0 | 0 = 0$$

Example:

```
a = 5
```

```
b = 3
```

```
print (a | b)
```

Output: 7

7.3 Bitwise XOR ^

Bitwise XOR gives 1 when bits are different.

Rule:

$$1 \wedge 1 = 0$$

$$1 \wedge 0 = 1$$

$$0 \wedge 1 = 1$$

$$0 \wedge 0 = 1$$

Example:

```
a = 5
b = 3
print (a ^ b)
```

Output: 6

7.4 Bitwise NOT ~

Bitwise NOT flips bits. In Python, the result of ~x is:

Example:

```
x = 5
print(~x)
```

Output: -6

Explanation: $\sim 5 = -(5 + 1)$ $\sim 5 = -6$ For beginners,

remember: Bitwise NOT does not simply make 5 into -5. It gives $-(\text{number} + 1)$.

7.5 Left Shift <<

Left shift moves bits to the left.

$x \ll n$ means x multiplied by 2 power n

Example:

```
x = 5
print (x << 1)
```

Output: 10

Explanation:

$5 \ll 1$

$5 \times 2 = 10$

Binary view: 5 in binary = 101

After left shift by 1: 1010 1010 in decimal = 10

7.6 Right Shift >>

Right shift moves bits to the right.

$x \gg n$ means x divided by 2 power n and gives whole-number result

Example:

```
x = 10
print (x >> 1)
```

Output: 5

Explanation:

$10 \gg 1$

$10 // 2 = 5$

Binary view: 10 in binary = 1010

After right shift by 1: 101 101 in decimal = 5

Practice Questions

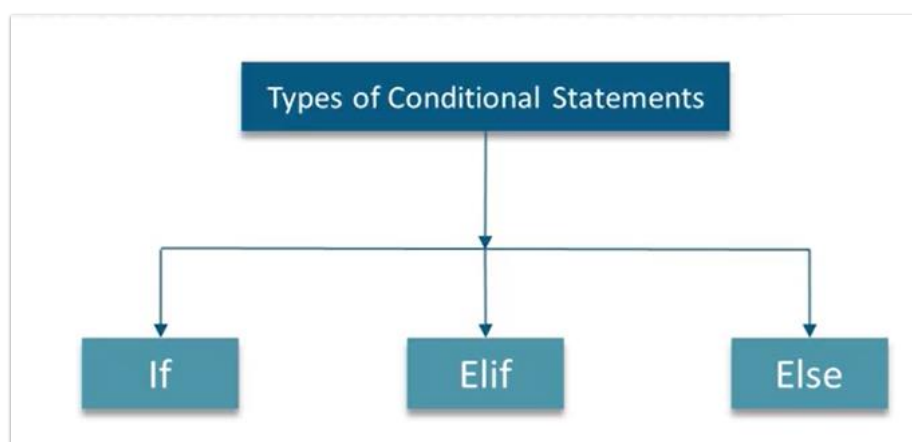
1. Write a program to add two numbers using the + operator.
2. Write a program to find the remainder of two numbers using %.
3. Write a program to find the power of a number using **.
4. Write a program to compare two numbers using ==.
5. Write a program to check whether one number is greater than another.
6. Write a program to check whether a number is divisible by both 5 and 10 using and.
7. Write a program to check whether a number is divisible by 2 or 3 using or.
8. Use += to add a value to a variable.
9. Check whether an element exists in a list using in.
10. Check whether a character does not exist in a string using not in.
11. Use the is operator to compare two variables.
12. Perform bitwise AND using &.
13. Perform bitwise OR using |.
14. Write a program to check whether a number is even or odd.
15. Create a simple calculator using arithmetic operators.

2.9 CONDITIONAL STATEMENTS

Conditional statements are used to make decisions in a program.

They execute different blocks of code based on whether a condition is True or False.

Python provide some conditional statements such as:



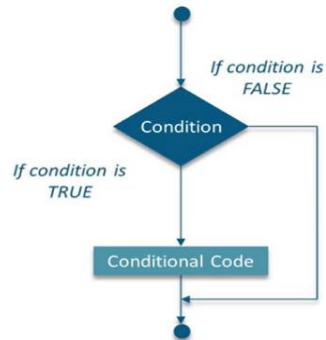
- **If Statement**
- **Else Statement**
- **Elif Statement**
- **Nested if Statement**

9.1 IF STATEMENT

The if statement executes code only when the condition is True.

Syntax:

```
if condition:  
    statement
```



Example:

```
age = 20  
if age >= 18:  
    print ("You are eligible to vote")  
Output: You are eligible to vote
```

9.2 IF-ELSE STATEMENT

Use if-else when there are two possible conditions.

Example:

```
age = 16  
if age >= 18:  
    print ("Eligible to vote")  
else:  
    print ("Not eligible to vote")  
Output: Not eligible to vote
```

9.3 IF-ELIF-ELSE STATEMENT

Use elif when there are multiple conditions.

Example: Grade Calculation

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

```
else:  
    print("Fail")
```

9.4 NESTED IF

An if statement inside another if statement is called a nested if.

```
age = 20  
has_license = True  
if age >= 18:  
    if has_license:  
        print ("You can drive")  
    else:  
        print ("You need a license")  
else:  
    print ("You are not eligible to drive")
```

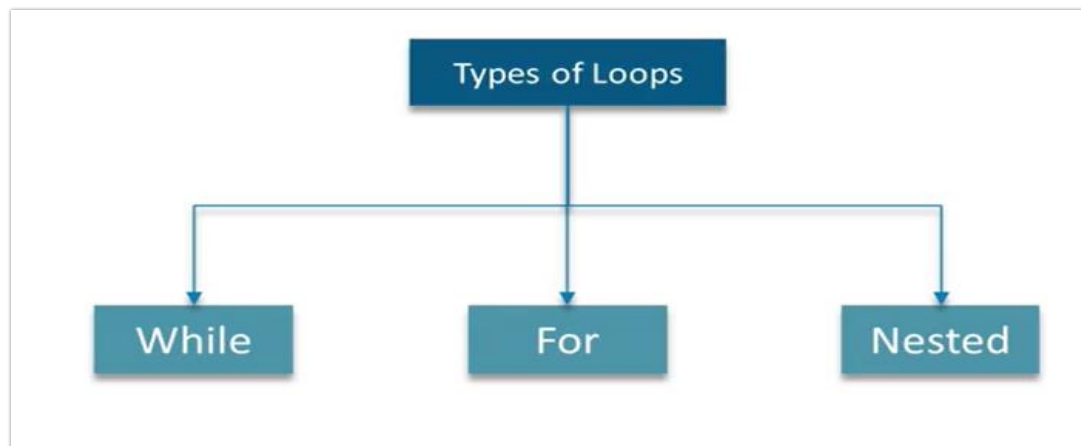
Practice Questions

1. What is a conditional statement in Python?
2. What is the purpose of the if statement?
3. What is the difference between if and if-else?
4. Write a program to check whether a number is positive or negative.
5. Write a program to check whether a number is even or odd.
6. Write a program to find the largest of two numbers.
7. Write a program to find the largest of three numbers.
8. Write a program to check whether a person is eligible to vote.
9. Write a program to check whether a student has passed or failed based on marks.
10. Write a program to check whether a number is divisible by 5 and 10.
11. Write a program to display a grade based on marks using if-elif-else.
12. Write a program to check whether a year is a leap year.
13. Write a program to check whether a character is a vowel or consonant.
14. Write a program to check whether a number is positive, negative, or zero.
15. Write a program to create a simple login system using username and password conditions.

2.10 LOOPS

- Loops are used to repeat a block of code.

- Python mainly provides two types of loops:



1. for loop
2. while loop

10.1 FOR LOOP

A for loop is used to iterate over a sequence such as:

String
List
Tuple
Set
Dictionary
Range

Example

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

Output:

1
2
3
4

Loop Through a String

```
name = "Python"  
for letter in name:  
    print(letter)
```

Output:

P
y
t
h
o
n

h
o
n

10.2 WHILE LOOP

A while loop executes as long as a condition is True.

Syntax:

```
while condition:  
    statement
```

Example:

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

Output:

```
1  
2  
3  
4  
5
```

Important: Update the loop variable; otherwise, you may create an infinite loop.

2.11 LOOP CONTROL STATEMENTS

11.1 BREAK

break statement immediately stops the loop.

Example:

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

Output:

```
1  
2  
3  
4
```

11.2 CONTINUE

The continue statement skips the current iteration.

Example:

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

Output:

```
1  
2  
4  
5
```

11.3 PASS

The pass statement is used as a placeholder when no action is required.

Example:

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

Practice Questions

1. What is a loop in Python?
2. What are the different types of loops in Python?
3. What is the difference between a for loop and a while loop?
4. Write a program to print numbers from 1 to 10 using a for loop.
5. Write a program to print numbers from 1 to 10 using a while loop.
6. Write a program to print all even numbers from 1 to 20.
7. Write a program to print all odd numbers from 1 to 20.
8. Write a program to find the sum of numbers from 1 to 10.
9. Write a program to print the multiplication table of a given number.
10. Write a program to find the factorial of a number using a loop.
11. Write a program to reverse a number using a loop.
12. Write a program to count the number of digits in a number.
13. Write a program to print each character of a string using a loop.
14. What is the use of the break statement in a loop?
15. What is the use of the continue statement in a loop?

2.12 BASIC PYTHON PROGRAMS

Check Even or Odd

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

```
print ("Odd number")
```

Check Positive, Negative, or Zero

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

Find the Largest Number

```
a = int (input ("Enter first number: "))  
b = int (input ("Enter second number: "))  
if a > b:  
    print ("Largest number is:", a)  
else:  
    print ("Largest number is:", b)
```

Print Numbers from 1 to 10

```
for i in range (1, 11):  
    print(i)
```

Print Even Numbers from 1 to 20

```
for i in range (1, 21):  
    if i % 2 == 0:  
        print(i)
```

Sum of Numbers from 1 to n

```
n = int (input ("Enter a number: "))  
total = 0  
for i in range (1, n + 1):  
    total = total + i  
    print ("Sum =", total)
```

Multiplication Table

```
number = int (input ("Enter a number: "))  
for i in range (1, 11):  
    print (number, "x", i, "=", number * i)
```

Factorial of a Number

```
number = int (input ("Enter a number: "))  
factorial = 1  
for i in range (1, number + 1):  
    factorial = factorial * i  
print ("Factorial =", factorial)
```

Check Leap Year

```
year = int (input ("Enter a year: "))
```

```
if year % 400 == 0 or (year % 4 == 0 and year % 100 != 0):  
    print ("Leap year")  
else:  
    print ("Not a leap year")
```

Reverse a Number

```
number = int (input ("Enter a number: "))  
reverse = 0  
while number > 0:  
    digit = number % 10  
    reverse = reverse * 10 + digit  
    number = number // 10  
print ("Reverse =", reverse)
```



CHAPTER 3: STRING MANIPULATION

String manipulation means performing different operations on text or strings, such as accessing, searching, modifying, formatting, and validating text.

A string is a sequence of characters written inside quotes.

```
name = "Python"
```

```
message = "Hello World"
```

This chapter covers the following topics.

3.1 STRING BASICS

A string is used to store text.

```
name = "Amit"
```

```
city = "Pune"
```

Strings can be created using:

```
"Python"
```

```
'Python'
```

Useful basic operations include:

len() – finds the length of a string

+ – joins strings

– repeats a string

Example:

```
name = "Python"
```

```
print(len(name))
```

```
print (name + " Programming")
```

3.2 INDEXING & SLICING

Indexing

Each character in a string has a position called an index.

```
P y t h o n
```

```
0 1 2 3 4 5
```

Example:

```
word = "Python"
```

```
print (word [0])
```

```
print (word [1])
```

Output:

```
P
```

```
Y
```

Negative Indexing

Negative indexing starts from the right side.

Example:

```
print (word [-1])
```

Output: n**Slicing**

Slicing is used to extract a part of a string.

Example:

```
word = "Python"
```

```
print (word [0:3])
```

Output: Pyt

3.3 STRING METHODS

Python provides built-in methods to perform operations on strings.

Method	Description
upper ()	Converts to uppercase
lower ()	Converts to lowercase
title ()	Capitalizes each word
strip ()	Removes extra spaces
replace ()	Replaces text
split ()	Replaces text
join ()	Joins strings
find ()	Finds a position
count ()	Counts occurrences

Example:

```
text = "python programming"
```

```
print (text.upper())
```

```
print (text.replace("python", "Java"))
```

3.4 STRING FORMATTING

String formatting is used to insert variables or values into a string.

Using f-string

```
name = "Amit"
```

```
age = 25
```

```
print (f"My name is {name} and I am {age} years old.")
```

You can also use:

```
"{}".format(name)
```

F-strings are commonly used because they are simple and readable.

3.5 ESCAPE CHARACTERS

Escape characters are special characters used inside strings.

Escape Character	Meaning
<code>\n</code>	New line
<code>\t</code>	Tab space
<code>\'</code>	Single quote
<code>\"</code>	Double quote
<code>\\</code>	Backslash

Example:

```
print("Hello\nPython")
```

Output:

```
Hello
Python
```

3.6 STRING VALIDATION

String validation methods are used to check the contents of a string.

Method	Checks
<code>isalpha()</code>	Only letters
<code>isdigit()</code>	Only digits
<code>isalnum()</code>	Letters and numbers
<code>islower()</code>	Lowercase letters
<code>isupper()</code>	Uppercase letters
<code>isspace()</code>	Only spaces

Example:

```
text = "Python123"
print(text.isalnum())
```

Output: True

These methods usually return True or False.

3.7 REGULAR EXPRESSIONS (BASICS)

A Regular Expression (Regex) is used to search, match, or find patterns in text. Python uses the re module.

import re

Common functions include:

Function	Purpose
re.search()	Searches for a pattern
re.match()	Matches from the beginning
re.findall()	Finds all matches
re.sub()	Replaces matching text

Example:

```
import re
text = "My age is 25"
result = re.findall(r"\d+", text)
print(result)
```

Output: ['25']

SUMMARY

String manipulation includes:

1. Creating strings
2. Indexing
3. Slicing
4. Concatenation
5. Repetition
6. Finding length
7. Changing case
8. Removing spaces
9. Searching
10. Replacing
11. Splitting
12. Joining
13. String validation
14. String formatting
15. Escape characters
16. Regular expressions (advanced/basic introduction)

Strings are especially important in data cleaning, text processing, validation, and data analysis.

Practice Questions

1. What is a string in Python?
 2. How do you create a string in Python?
 3. Write a program to find the length of a string using `len()`.
 4. Write a program to access the first and last character of a string.
 5. Write a program to convert a string into uppercase using `upper()`.
 6. Write a program to convert a string into lowercase using `lower()`.
 7. Write a program to check whether a word exists in a string.
 8. Write a program to count the number of vowels in a string.
 9. Write a program to replace a word in a string using `replace()`.
 10. Write a program to remove extra spaces using `strip()`.
 11. Write a program to reverse a string.
 12. Write a program to check whether a string is a palindrome.
 13. Write a program to split a sentence into words using `split()`.
 14. Write a program to join a list of words into a string using `join()`.
 15. Write a program to count how many times a character appears in a string using `count()`.
-



INTRODUCTION

In programming, we often need to store and manage multiple values.

For example:

- Names of students
- Employee details
- Product information
- Marks of students
- Unique customer IDs

Python provides several built-in data structures to organize and store data efficiently.

The four most commonly used Python data structures are:

1. List
2. Tuple
3. Set
4. Dictionary

Data	Structure	Ordered	Mutable Allows Duplicates
List	Yes	Yes	Yes
Tuple	Yes	No	Yes
Set	No	Yes	No
Dictionary	Yes	Yes	Keys must be unique

*In modern Python versions, dictionaries maintain insertion order.

4.1 Lists

1.1 INTRODUCTION TO LISTS

A **list** is a collection of multiple items stored in a single variable.

Lists are:

- Ordered
- Mutable, meaning they can be changed
- Able to store duplicate values
- Able to store different data types

Example

```
students = ["Shruti", "Rahul", "Priya", "Amit"]  
print(students)
```

Output:

```
[shruti, 'Rahul', 'Priya', 'Amit']  
A list can also contain different types of data.  
data = ["shruti", 25, 85.5, True]  
print(data)
```

1.2 CREATING LISTS

Lists are created using square brackets [].

- **Empty List**
 - numbers = []
 - print(numbers)
- **List of Numbers**
 - numbers = [10, 20, 30, 40, 50]
 - print(numbers)
- **List of Strings**
 - cities = ["Pune", "Mumbai", "Delhi", "Nashik"]
 - print(cities)
- **Mixed Data Type List**
 - employee = ["Kunal", 101, 45000, True]
 - print(employee)

1.3 ACCESSING LIST ELEMENTS

Each element in a list has a position called an **index**.

Python indexing starts from 0.

```
Index:  0   1   2   3  
List: ["Apple", "Mango", "Banana", "Orange"]
```

Example

```
fruits = ["Apple", "Mango", "Banana", "Orange"]  
print (fruits [0])
```

Output: Apple

Accessing another element:

```
print (fruits [2])
```

Output: Banana

1.4 INDEXING

Python supports both **positive indexing** and **negative indexing**.

Positive Indexing

```
numbers = [10, 20, 30, 40, 50]  
print (numbers [0])  
print (numbers [3])
```

Output:

10
40

Negative Indexing

Negative indexing starts from the end of the list.

Index: -5 -4 -3 -2 -1

List: 10 20 30 40 50

```
numbers = [10, 20, 30, 40, 50]
```

```
print (numbers [-1])
```

```
print (numbers [-2])
```

Output:

50
40

1.5 SLICING

Slicing is used to extract a portion of a list.

Syntax: [start: stop]

The start index is included, but the stop index is excluded.

Example:

```
numbers = [10, 20, 30, 40, 50, 60]
```

```
print (numbers [1:4])
```

Output: [20, 30, 40]

- **From Beginning**

```
print (numbers [:3])
```

Output: [10, 20, 30]

- **Until the End**

```
print (numbers [2:])
```

Output: [30, 40, 50, 60]

- **Using Steps**

```
print (numbers [: :2])
```

Output: [10, 30, 50]

1.6 UPDATING LISTS

Lists are mutable, so their elements can be changed.

Example

```
fruits = ["Apple", "Mango", "Banana"]
```

```
fruits [1] = "Orange"
```

```
print(fruits)
```

Output: ['Apple', 'Orange', 'Banana']

1.7 ADDING ELEMENTS

Python provides several methods to add elements to a list.

7.1 APPEND ()

The append () method adds one item to the end of the list.

Example

```
numbers = [10, 20, 30]
numbers.append(40)
print(numbers)
Output: [10, 20, 30, 40]
```

7.2 EXTEND ()

The extend () method adds multiple elements to a list.

Example

```
numbers = [10, 20, 30]
numbers.extend([40, 50, 60])
print(numbers)
Output: [10, 20, 30, 40, 50, 60]
```

Difference Between append () and extend ()

Using append ():

```
a = [1, 2, 3]
a.append([4, 5])
print(a)
Output: [1, 2, 3, [4, 5]]
```

Using extend ():

```
a = [1, 2, 3]
a.extend([4, 5])
print(a)
Output: [1, 2, 3, 4, 5]
```

7.3 INSERT ()

The insert () method adds an element at a specific position.

Syntax: list.insert (index, value)

Example:

```
numbers = [10, 20, 40]
numbers.insert(2, 30)
print(numbers)
Output: [10, 20, 30, 40]
```

1.8 REMOVING ELEMENTS

Python provides different ways to remove elements from a list.

8.1 REMOVE ()

The remove () method removes a specific value.

```
fruits = ["Apple", "Mango", "Banana"]
fruits.remove("Mango")
print(fruits)
```

Output: ['Apple', 'Banana']

8.2 POP ()

The pop () method removes and returns an element.

Without an index, it removes the last item.

Example:

```
numbers = [10, 20, 30]
numbers.pop()
print(numbers)
```

Output: [10, 20]

Removing a specific index:

Example:

```
numbers = [10, 20, 30, 40]
numbers.pop(1)
print(numbers)
```

Output: [10, 30, 40]

8.3 CLEAR ()

The clear () method removes all elements from the list.

Example:

```
numbers = [10, 20, 30]
numbers.clear()
print(numbers)
```

Output: []

8.4 DEL

The del keyword can delete an element or an entire list.

Delete an Element

```
numbers = [10, 20, 30, 40]
del numbers[1]
print(numbers)
```

Output: [10, 30, 40]

Delete the Entire List

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

del numbers

1.9 LIST METHODS

Method	Description
append ()	Adds an item at the end
extend ()	Adds multiple items
insert ()	Adds an item at a specific position
remove ()	Removes a specific item
pop ()	Removes an item by index
clear ()	Removes all items
index ()	Finds the index of an item
count ()	Counts occurrences
sort ()	Sorts the list
reverse ()	Reverses the list
copy ()	Creates a copy

Some commonly used list methods are:

Example:

```
numbers = [10, 20, 10, 30, 10]
print(numbers.count(10))
print(numbers.index(20))
```

Output:

```
3
1
```

1.10 NESTED LISTS

A list inside another list is called a **nested list**.

Example:

```
students = [
    ["Amit", 85],
    ["Rahul", 90],
    ["Priya", 88]
]
```

```
print(students)
```

Accessing nested elements:

```
print (students [0][0])
print (students [1][1])
```

Output:

1.11 LIST COMPREHENSION

List comprehension provides a shorter way to create lists.

Traditional Method

```
numbers = [1, 2, 3, 4, 5]
squares = []
for number in numbers:
    squares.append(number ** 2)
print(squares)
```

Using List Comprehension

```
numbers = [1, 2, 3, 4, 5]
squares = [number ** 2 for number in numbers]
print(squares)
```

Output: [1, 4, 9, 16, 25]

List Comprehension with Condition

```
numbers = [1, 2, 3, 4, 5, 6]
even_numbers = [number for number in numbers if number % 2 == 0]
print(even_numbers)
```

Output: [2, 4, 6]

1.12 COPYING LISTS

- **Using Assignment**

```
list1 = [10, 20, 30]
list2 = list1
Both variables refer to the same list.
```

- **Using copy ()**

```
list1 = [10, 20, 30]
list2 = list1.copy()
print(list2)
Output: [10, 20, 30]
```

1.13 SORTING LISTS

13.1 sort ()

The sort () method changes the original list.

```
numbers = [40, 10, 30, 20]
numbers.sort()
print(numbers)
```

Output: [10, 20, 30, 40]

Descending Order

```
numbers.sort(reverse=True)
print(numbers)
```

Output: [40, 30, 20, 10]

13.2 sorted ()

The sorted () function returns a new sorted list.

```
numbers = [40, 10, 30, 20]
new_list = sorted(numbers)
print(new_list)
print(numbers)
```

1.14 ITERATING THROUGH LISTS

A for loop is commonly used to iterate through a list.

```
fruits = ["Apple", "Mango", "Banana"]
for fruit in fruits:
    print(fruit)
```

Output:

```
Apple
Mango
Banana
```

Using enumerate ():

```
fruits = ["Apple", "Mango", "Banana"]
for index, fruit in enumerate(fruits):
    print (index, fruit)
```

Output:

```
0 Apple
1 Mango
2 Banana`
```

4.2. TUPLES

2.1 INTRODUCTION TO TUPLES

A tuple is an ordered collection of items.

Unlike lists, tuples are **immutable**, which means their elements cannot be changed after creation.

Example

```
colors = ("Red", "Green", "Blue")
print(colors)
```

2.2 CREATING TUPLES

```
numbers = (10, 20, 30, 40)
print(numbers)
```

Single Element Tuple

A comma is required when creating a tuple with one item.

```
number = (10,)
print(number)
```

Without the comma:

```
number = (10)
print(type(number))
```

This creates an integer, not a tuple.

2.3 TUPLE INDEXING

```
colors = ("Red", "Green", "Blue")
print(colors[0])
print(colors[-1])
```

Output:

```
Red
Blue
```

2.4 TUPLE SLICING

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

Output: (20, 30, 40)

2.5 TUPLE METHODS

Tuples have two commonly used methods.

- **count ()**

```
numbers = (10, 20, 10, 30, 10)
print(numbers.count(10))
```

Output: 3

- **index ()**

```
numbers = (10, 20, 30, 40)
print(numbers.index(30))
```

Output: 2

2.6 PACKING AND UNPACKING

- **Tuple Packing**

```
student = ("Amit", 25, "Pune")
```

Multiple values are packed into one tuple.

- **Tuple Unpacking**

```
name, age, city = student
print(name)
print(age)
print(city)
```

Output:

```
Amit
25
Pune
```

2.7 NESTED TUPLES

A tuple can contain other tuples.

Example:

```
data = (
    ("Amit", 85),
    ("Rahul", 90),
    ("Priya", 88)
)
print (data [0])
Accessing an individual value:
print (data [0][0])
```

Output: Amit

2.8 ADVANTAGES OF TUPLES

Tuples are useful because:

- They are immutable.
- They protect data from accidental modification.
- They can be used to represent fixed collections.
- They are suitable for storing data that should not change.

Example: days = ("Monday", "Tuesday", "Wednesday", "Thursday", "Friday")

4.3. SETS

3.1 INTRODUCTION TO SETS

A set is a collection of **unique elements**.

Sets:

Do not allow duplicate values.

Are mutable.

Are useful for mathematical set operations.

Example:

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

Duplicate values are automatically removed.

3.2 CREATING SETS

```
fruits = {"Apple", "Mango", "Banana"}
print(fruits)
```

Empty Set

```
data = set ()
print(data)
```

Do not use {} for an empty set because {} creates an empty dictionary.

3.3 SET METHODS

Some important set methods include:

Method	Description
add ()	Adds one element
update ()	Adds multiple elements
remove ()	Removes an element
discard ()	Removes an element safely
pop ()	Removes an element
clear ()	Removes all elements

3.4 ADDING & REMOVING ELEMENTS

- **add ()**

```
numbers = {10, 20, 30}
numbers.add(40)
print(numbers)
```

- **update ()**

```
numbers = {10, 20, 30}
numbers.update([40, 50, 60])
print(numbers)
```

- **remove ()**

```
numbers = {10, 20, 30}
numbers.remove(20)
print(numbers)
```

- **discard ()**

```
numbers = {10, 20, 30}
numbers.discard(20)
print(numbers)
```

discard () does not raise an error if the value does not exist.

3.5 UNION

Union combines all unique elements from two sets.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
result = set1.union(set2)
print(result)
```

- **Conceptually:** {1, 2, 3, 4, 5}
- **You can also use the | operator:** result = set1 | set2

3.6 INTERSECTION

Intersection returns elements that are common to both sets.

```
set1 = {1, 2, 3}
set2 = {2, 3, 4}
result = set1.intersection(set2)
print(result)
```

- **Conceptually:** {2, 3}
- **You can also use:** result = set1 & set2

3.7 DIFFERENCE

Difference returns elements that exist in the first set but not in the second set.

```
set1 = {1, 2, 3}
set2 = {2, 3, 4}
result = set1.difference(set2)
print(result)
```

- **Conceptually:** {1}
- **Using an operator:** result = set1 - set2

3.8 SYMMETRIC DIFFERENCE

Symmetric difference returns elements that exist in either set but not in both.

```
set1 = {1, 2, 3}
set2 = {2, 3, 4}
result = set1.symmetric_difference(set2)
print(result)
```

- **Conceptually:** {1, 4}
- **Using an operator:** result = set1 ^ set2

3.9 FROZEN SET

A frozen set is an immutable version of a set.

```
numbers = frozenset([10, 20, 30])  
print(numbers)
```

You cannot add or remove elements from a frozen set.

```
numbers.add(40)
```

This will produce an error.

3.10 SET COMPREHENSION

Set comprehension provides a short way to create a set.

```
numbers = [1, 2, 3, 4, 5]  
squares = {number ** 2 for number in numbers}  
print(squares)
```

Output: {1, 4, 9, 16, 25}

4. 4 DICTIONARIES

4.1 INTRODUCTION TO DICTIONARIES

A dictionary stores data in **key-value pairs**.

Example:

```
student = {  
    "name": "Manali",  
    "age": 25,  
    "city": "Pune"  
}  
print(student)
```

In this example:

"name" is a key.

"Manali" is its value.

4.2 CREATING DICTIONARIES

```
employee = {  
    "id": 101,  
    "name": "Rahul",  
    "salary": 50000  
}  
print(employee)
```

An empty dictionary:

```
data = {}  
print(data)
```

4.3 ACCESSING VALUES

Values can be accessed using keys.

```
student = {  
    "name": " Manali ",  
    "age": 20  
}  
print(student["name"])
```

Output:

Using get ():

```
print(student.get("age"))
```

The get () method is useful when you want to safely access a value.

4.4 ADDING & UPDATING DATA

Adding Data

```
student = {  
    "name": " Manali ",  
    "age": 25  
}  
student["city"] = "Pune"  
print(student)
```

Output: {'name': 'Manali', 'age': 25, 'city': 'Pune'}

Updating Data

```
student["age"] = 26  
print(student)
```

4.5 REMOVING ITEMS

pop ()

```
student = {  
    "name": "Sonali",  
    "age": 25,  
    "city": "Pune"  
}  
student.pop("age")  
print(student)
```

pop item ()

Removes the last inserted key-value pair.

```
student.popitem()
```

del

```
del student["city"]
```

clear ()

```
student.clear()
```

```
print(student)
```

Output: {}

Example

```
student = {  
    "name": "Amit",  
    "age": 25  
}  
print(student.keys())  
print(student.values())  
print(student.items())
```

4.6 DICTIONARY METHODS

Some important dictionary methods are:

Method	Description
keys ()	Returns all keys
values ()	Returns all values
items ()	Returns key-value pairs
get ()	Returns a value using a key
update ()	Updates dictionary data
pop ()	Removes a specific item
pop item ()	Removes the last item
clear ()	Removes all items
copy ()	Creates a copy

4.7 NESTED DICTIONARIES

A dictionary inside another dictionary is called a nested dictionary.

Example:

```
students = {  
    "student1": {  
        "name": "Manali",  
        "marks": 85
```

```
},
"student2": {
    "name": "Rahul",
    "marks": 90
}
}
print(students)
Accessing nested data:
print(students["student1"]["name"])
Output: Manali
```

4.8 DICTIONARY COMPREHENSION

Dictionary comprehension is a shorter way to create dictionaries.

Example

```
numbers = [1, 2, 3, 4, 5]
squares = {number: number ** 2 for number in numbers}
print(squares)
```

Output: {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

Example with Condition

```
numbers = [1, 2, 3, 4, 5, 6]
even_numbers = {
    number: number ** 2
    for number in numbers
    if number % 2 == 0
}
print(even_numbers)
```

Output: {2: 4, 4: 16, 6: 36}

4.9 ITERATING THROUGH DICTIONARIES

- **Iterating Through Keys**

```
student = {
    "name": "Manali ",
    "age": 25,
    "city": "Pune"
}
for key in student:
    print(key)
```

- **Iterating Through Values**

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

- **Iterating Through Keys and Values**

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

Output

```
name: Manali  
age: 25  
city: Pune
```

- **Quick Comparison**

Feature	List	Tuple	Set	Dictionary
Ordered	Yes	Yes	No	Yes
Mutable	Yes	No	Yes	Yes
Duplicates	Yes	Yes	No	Values can repeat
Indexing	Yes	Yes	No	By key
Syntax	<code>[]</code>	<code>()</code>	<code>{}</code>	<code>{key: value}</code>

Practice Questions

- **Lists**

1. Create a list containing five numbers.
2. Add a new element using `append ()`.
3. Insert an element at index 2.
4. Remove an element using `remove ()`.
5. Create a list of squares using list comprehension.
6. Sort a list in ascending and descending order.

- **Tuples**

1. Create a tuple containing five cities.
2. Access the first and last elements.
3. Slice the tuple.
4. Perform tuple packing and unpacking.
5. Create a nested tuple.

- **Sets**

1. Create a set containing duplicate values.
2. Add and remove elements.
3. Find the union of two sets.
4. Find the intersection of two sets.
5. Find the difference between two sets.
6. Create a set using set comprehension.

- **Dictionaries**

1. Create a dictionary containing student information.
2. Add a new key-value pair.
3. Update an existing value.

4. Remove an item using pop ().
5. Create a nested dictionary.
6. Iterate through keys and values.
7. Create a dictionary using dictionary comprehension.

- **Mini Project: Student Data Management**

```
students = {  
    101: {  
        "name": "Sonali",  
        "marks": 85 },  
    102: {  
        "name": "Rahul",  
        "marks": 90 },  
    103: {  
        "name": "Priya",  
        "marks": 88 }  
}  
for student_id, details in students.items():  
    print ("Student ID:", student_id)  
    print ("Name:", details["name"])  
    print ("Marks:", details["marks"])  
    print ()
```

This project demonstrates how **dictionaries**, **nested dictionaries**, and **loops** can be used together to manage structured data.⁴



5.1 INTRODUCTION TO FUNCTIONS

A function is a block of reusable code that performs a specific task.

1.1 ADVANTAGES OF FUNCTIONS

- Reuse code.
- Reduce repetition.
- Make programs easier to understand.
- Make code easier to maintain.

Types of Functions in Python

There are mainly four types of functions in Python:

1. Built-in Functions

Functions already provided by Python.

Examples: print (), len(), sum (), type ()

2. User-Defined Functions

Functions created by the programmer using the def keyword.

Example:

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

3. Lambda Functions

Small anonymous functions created using the lambda keyword.

```
square = lambda x: x * x
```

4. Recursive Functions

Functions that call themselves.

Example:

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

5.2 DEFINING AND CALLING FUNCTIONS

2.1 DEFINING A FUNCTION

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

2.2 CALLING A FUNCTION

`greet ()`

Function Syntax:

```
def function_name():  
    # code
```

2.3 RETURN STATEMENT

The return statement sends a result back from a function.

Example:

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

5.3 TYPES OF FUNCTIONS

3.1 BUILT-IN FUNCTIONS

Examples:

```
print()  
len()  
type()  
sum()
```

3.2 USER-DEFINED FUNCTIONS

Functions created by the programmer.

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

3.3 RECURSIVE FUNCTIONS

A recursive function calls itself.

3.4 LAMBDA FUNCTIONS

A lambda function is a small anonymous function.

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

5.4 RETURN VALUES

A function can return different types of values.

4.1 SINGLE VALUE

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

4.2 MULTIPLE VALUES

```
def calculate (a, b):  
    return a + b, a - b
```

A function can also return lists, tuples, and dictionaries.

If a function does not use return, it returns None.

5.5 FUNCTION ARGUMENTS

5.1 POSITIONAL ARGUMENTS

Arguments are passed based on their position.

Example:

```
def student (name, age):  
    print (name, age)  
    student ("Amit", 25)
```

5.2 KEYWORD ARGUMENTS

Arguments are passed using parameter names.

Example:

```
student (name="Amit", age=25)
```

5.3 DEFAULT ARGUMENTS

A default value is assigned to a parameter.

Example:

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

5.4 VARIABLE -LENGTH ARGUMENTS(*ARGS)

Used to accept multiple positional arguments.

Example:

```
def add(*numbers):  
    return sum(numbers)
```

5.5 KEYWORD VARIABLE -LENGTH ARGUMENTS (**KWARGS)

Used to accept multiple keyword arguments.

Example:

```
def display(**data):  
    print(data)
```

5.6 VARIABLE SCOPE

6.1 LOCAL VARIABLES

A local variable is created inside a function and can normally be used only within that function.

Example:

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

6.2 GLOBAL VARIABLES

A **global variable** is a variable that is declared outside of a function and can be accessed from different parts of the program.

Example:

```
name = "Amit" # Global variable  
def display():  
    print(name)  
display()
```

Output: Amit

Here, name is a global variable because it is defined outside the function.

2.1 ACCESSING GLOBAL VARIABLES

A global variable can be **accessed inside a function** without declaring it again.

Example:

```
x = 100  
def show():  
    print(x)  
show()
```

Output: 100

The function can read the value of the global variable.

2.2 MODIFYING GLOBAL VARIABLES

By default, a function can **read** a global variable, but if you want to modify it inside the function, you need to use the global keyword.

Example:

```
count = 10
def update():
    global count
    count = 20
    update()
print(count)
```

Output: 20

Without global, Python treats an assignment inside the function as creating a local variable.

2.3 GLOBAL KEYWORD

The global keyword is used when you need to modify a global variable inside a function.

The **global keyword** is used inside a function to tell Python that a variable belongs to the global scope.

Example:

```
count = 0
def update ():
    global count
    count = count + 1
```

5.7 ANONYMOUS (LAMBDA) FUNCTIONS

7.1 WHAT IS A LAMBDA FUNCTION?

A **lambda function** is a small, anonymous function that is written in a single line using the lambda keyword.

Unlike a normal function, a lambda function does not use the def keyword.

Example:

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

Output: 25

Lambda functions are commonly used when a function is needed for a **short period of time** or as an argument to functions such as map(), filter(), and sorted().

7.2 SYNTAX:

lambda arguments: expression

7.3 LAMBDA WITH SINGLE ARGUMENT

A lambda function can accept a single argument.

```
square = lambda x: x * x
```

Example:

```
square = lambda x: x ** 2  
print(square(4))
```

Output: 16

Another example:

```
double = lambda x: x * 2  
print(double(10))
```

Output: 20

7.4 LAMBDA WITH MULTIPLE ARGUMENTS

A lambda function can accept multiple arguments.

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

Example:

```
print (add (10, 20))
```

Output: 30

Example with three arguments:

```
total = lambda a, b, c: a + b + c  
print (total (10, 20, 30))
```

Output: 60

7.5 LAMBDA IN CONDITIONAL EXPRESSIONS

A lambda function can use a conditional expression to return different values.

Example:

```
check = lambda x: "Even" if x % 2 == 0 else "Odd"
```

Example:

```
print(check(10))  
print(check(7))
```

Output:

Even
Odd

Another example:

```
maximum = lambda a, b: a if a > b else b
```

```
print(maximum(10, 20))
```

Output: 20

7.6 LAMBDA WITH MAP()

The map() function applies a function to each item in an iterable.

Syntax:

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

Example:

```
numbers = [1, 2, 3, 4, 5]
squares = list(map(lambda x: x ** 2, numbers))
print(squares)
```

Output: [1, 4, 9, 16, 25]

Here, the lambda function calculates the square of every number.

7.7 LAMBDA WITH FILTER()

The filter() function is used to select elements that satisfy a condition.

Example:

```
numbers = [1, 2, 3, 4, 5, 6]
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers)
```

Output: [2, 4, 6]

Only numbers that satisfy the condition $x \% 2 == 0$ are selected.

7.8 LAMBDA WITH REDUCE ()

The reduce () function applies a function cumulatively to the elements of an iterable.

reduce () is available in the functools module.

Example:

```
from functools import reduce
numbers = [1, 2, 3, 4, 5]
total = reduce (lambda a, b: a + b, numbers)
print(total)
```

Output: 15

The calculation happens as:

$1 + 2 = 3$

$3 + 3 = 6$

$6 + 4 = 10$

$$10 + 5 = 15$$

7.9 SORTING USING LAMBDA

Lambda functions are commonly used with `sorted()` to specify the sorting criteria.

Example: Sort by Length

```
names = ["Shruti", "Amit", "Priya", "Raj"]
result = sorted(names, key=lambda x: len(x))
print(result)
```

Output: ['Raj', 'Amit', 'Priya', 'Shruti']

Sorting a List of Dictionaries

```
students = [
    {"name": "Shruti", "marks": 85},
    {"name": "Amit", "marks": 92},
    {"name": "Priya", "marks": 78}
]
result = sorted(students, key=lambda x: x["marks"])
print(result)
```

This sorts the students according to their marks.

For descending order:

```
result = sorted(
    students,
    key=lambda x: x["marks"],
    reverse=True
)
```

7.10 PRACTICAL USE CASES

Lambda functions are useful in many situations:

- **Quick Calculations**

```
multiply = lambda a, b: a * b
print(multiply(5, 4))
```

- **Data Transformation**

Using `map()`:

```
prices = [100, 200, 300]
new_prices = list(map(lambda x: x * 1.10, prices))
print(new_prices)
```

- **Filtering Data**

Using `filter()`:

```
ages = [12, 18, 25, 15, 30]
adults = list(filter(lambda x: x >= 18, ages))
print(adults)
```

- **Sorting Data**

```
numbers = [40, 10, 30, 20]
result = sorted(numbers, key=lambda x: x)
print(result)
```

- **Working with Data**

Lambda functions are frequently used with **Pandas** for operations such as transforming columns, filtering data, and creating calculated values.

5.8 RECURSIVE FUNCTIONS

8.1 WHAT IS RECURSION?

Recursion is a programming technique in which a function **calls itself** to solve a problem.

A recursive function keeps calling itself with a smaller or simpler input until it reaches a condition that stops the recursion.

Example:

```
def count(n):
    if n == 0:
        return
    print(n)
    count(n - 1)
count(5)
```

Output:

```
5
4
3
2
1
```

Here, count() calls itself until n becomes 0.

8.2 RECURSIVE FUNCTION SYNTAX

The basic structure of a recursive function is:

Syntax:

```
def function(parameters):
    if base_condition:
```

```
    return result
    return function(smaller_input)
```

A recursive function generally contains two important parts:

- **Base Case**
- **Recursive Case**

8.3 BASE CASE

The **base case** is the condition that **stops the recursion**.

Without a base case, the function will continue calling itself indefinitely and eventually cause a RecursionError.

Example:

```
def count(n):
    if n == 0:    # Base case
        return
    print(n)
    count(n - 1)
```

Here:

```
if n == 0:
    return
is the base case.
```

8.4 RECURSIVE CASE

The **recursive case** is the part where the function **calls itself** with a smaller or simpler input.

Example:

```
def count(n):
    if n == 0:
        return
    print(n)
    count(n - 1)  # Recursive case
```

Here:

```
count(n - 1)
is the recursive case.
```

Recursion Flow

For count(3):

```
count(3)
↓
count(2)
↓
count(1)
```

↓
count(0)
↓
Stop

8.5 ADVANTAGES AND LIMITATIONS

- **Advantages**

1. **Simple code** – Some problems can be expressed more clearly using recursion.
2. **Useful for complex problems** – Commonly used in tree and graph problems.
3. **Reduces code complexity** – Certain repetitive problems can be solved with fewer lines.
4. **Useful for divide-and-conquer algorithms** – Examples include merge sort and quicksort.

- **Limitations**

1. **More memory usage** – Each function call uses memory on the call stack.
2. **Slower in some cases** – Repeated function calls can create additional overhead.
3. **Recursion depth limitation** – Too many recursive calls can cause RecursionError.
4. **Can be difficult to understand** – Beginners may find recursive logic harder to follow.
5. **Not always efficient** – Some recursive solutions perform the same calculations repeatedly.

8.6 PRACTICAL EXAMPLES

6.1 FACTORIAL USING RECURSION

The **factorial** of a number n is:

$$n! = n \times (n-1) \times (n-2) \times \dots \times 1$$

For example:

$$\begin{aligned} 5! &= 5 \times 4 \times 3 \times 2 \times 1 \\ &= 120 \end{aligned}$$

Python Program

```
def factorial(n):  
    if n == 0 or n == 1:  
        return 1  
    return n * factorial(n - 1)  
print(factorial(5))
```

Output: 120

Working

```
factorial(5)
= 5 × factorial(4)
= 5 × 4 × factorial(3)
= 5 × 4 × 3 × factorial(2)
= 5 × 4 × 3 × 2 × factorial(1)
= 120
```

Here, $n == 0$ or $n == 1$ is the **base case**.

6.2 FIBONACCI SERIES USING RECURSION

The **Fibonacci series** is a sequence in which each number is the sum of the previous two numbers.

0, 1, 1, 2, 3, 5, 8, 13, ...

The formula is:

$$F(n) = F(n-1) + F(n-2)$$

Python Program

```
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)
for i in range(8):
    print(fibonacci(i), end=" ")
```

Output: 0 1 1 2 3 5 8 13

Here:

```
if n <= 1:
    return n
is the base case.
```

And:

```
return fibonacci(n - 1) + fibonacci(n - 2)
is the recursive case.
```

6.3 SUM OF NATURAL NUMBERS USING RECURSION

Natural numbers can be represented as:

1, 2, 3, 4, 5, ...

For example:

$1 + 2 + 3 + 4 + 5 = 15$

Python Program

```
def sum_natural(n):
    if n == 0:
```

```
return 0
return n + sum_natural(n - 1)
print(sum_natural(5))
```

Output:15

Working

```
sum_natural(5)
= 5 + sum_natural(4)
= 5 + 4 + sum_natural(3)
= 5 + 4 + 3 + sum_natural(2)
= 5 + 4 + 3 + 2 + sum_natural(1)
= 5 + 4 + 3 + 2 + 1 + sum_natural(0)
= 15
```

Practice Questions

1. What is a function in Python?
2. Why do we use functions?
3. What is the difference between a built-in function and a user-defined function?
4. Write a function to print "Hello, Python!".
5. Write a function to add two numbers.
6. Write a function to find the square of a number.
7. Write a function to check whether a number is even or odd.
8. Write a function to find the largest of two numbers.
9. Write a function with a default argument.
10. Write a function using *args to find the sum of multiple numbers.
11. Write a function using **kwargs to display student details.
12. What is a lambda function? Write a lambda function to find the square of a number.
13. What is a recursive function?
14. Write a recursive function to calculate the factorial of a number.
15. Write a function to check whether a number is prime.



6.1 EXCEPTION HANDLING

1.1 INTRODUCTION TO EXCEPTION HANDLING

Exception handling is a mechanism in Python used to **handle errors that occur during program execution** without abruptly stopping the program.

1.1 WHAT IS AN EXCEPTION?

An **exception** is an error that occurs during the execution of a program and interrupts its normal flow.

Example:

```
a = 10
b = 0
print(a / b)
```

This produces:

Zero Division Error: division by zero

Python raises an exception because a number cannot be divided by zero.

1.2 TYPES OF ERRORS

Python programs can have different types of errors. The three common categories are:

1. Syntax Errors
2. Runtime Errors
3. Logical Errors

2.1 SYNTAX ERRORS

A **syntax error** occurs when the rules or syntax of Python are not followed.

Example:

```
if 10 > 5
    print("Greater")
```

Here, the colon : is missing after the condition.

Python will report a SyntaxError.

Correct Code

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

2.2 RUNTIME ERRORS

A **runtime error** occurs while the program is running.

These are generally represented by exceptions in Python.

Example:

```
a = 10
b = 0
print(a / b)
```

Output: Zero Division Error

Other common runtime exceptions include:

```
ValueError
TypeError
IndexError
KeyError
FileNotFoundError
ZeroDivisionError
```

2.3 LOGICAL ERRORS

A **logical error** occurs when the program runs successfully but produces an incorrect result.

Example:

```
length = 10
width = 5
area = length + width
print(area)
```

The program runs without an error, but the calculation is incorrect.

The correct formula for the area of a rectangle is:

```
area = length * width
```

Logical errors are usually found by **testing and checking the program's output**.

1.3 WHY EXCEPTION HANDLING IS IMPORTANT

Exception handling is important because it:

- Prevents the program from stopping unexpectedly.
- Allows errors to be handled properly.
- Makes programs more reliable.
- Provides meaningful error messages.
- Allows the program to continue when possible.
- Helps handle unexpected situations such as invalid input or missing files.

Example Without Exception Handling

```
num = int(input("Enter a number: "))
print(100 / num)
```

If the user enters 0, the program produces an exception.

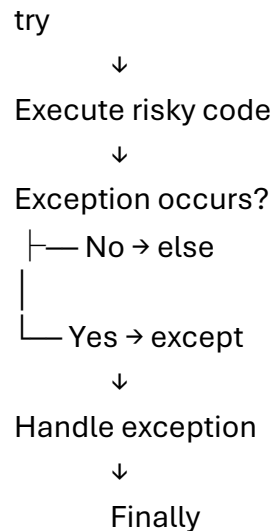
With Exception Handling

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

```
print(100 / num)
except ZeroDivisionError:
    print("Cannot divide by zero.")
Now the error is handled gracefully.
```

1.4 EXCEPTION HANDLING FLOW

The basic exception handling flow is:



The main keywords used are:

```
try
except
else
finally
raise
assert
```

1.2 HANDLING EXCEPTIONS

2.1 TRY BLOCK

The try block contains code that may cause an exception.

Syntax

```
try:
    # Code that may cause an exception
```

Example:

```
try:
    result = 10 / 0
```

However, a try block should normally be followed by an except or another appropriate exception-handling clause.

2.2 EXCEPT BLOCK

The except block is used to **handle an exception** that occurs in the try block.

Example:

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Cannot divide by zero.")
```

Output: Cannot divide by zero.

2.3 HANDLING MULTIPLE EXCEPTIONS

A program may encounter different types of exceptions.

You can use multiple except blocks to handle them separately.

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
```

except ValueError:

```
    print("Please enter a valid number.")
```

except ZeroDivisionError:

```
    print("Cannot divide by zero.")
```

If the user enters:

abc

the ValueError block executes.

If the user enters:

0

the ZeroDivisionError block executes.

2.4 GENERIC EXCEPTION (EXCEPTION)

Exception is the general base class for most built-in exceptions.

It can be used to handle an exception when the exact type is not known.

Example:

```
try:
    num = int(input("Enter a number: "))
    print(10 / num)
```

except Exception as e:

```
    print("An error occurred:", e)
```

Here, e stores information about the exception.

Important

A specific exception should generally be handled before using a generic Exception.

For example:

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

```
print(10 / num)
except ValueError:
    print("Invalid input.")
except Exception as e:
    print("Other error:", e)
```

2.5 NESTED TRY-EXCEPT

A try-except block can be placed inside another try block. This is called **nested exception handling**.

Example:

```
try:
    num = int(input("Enter a number: "))
    try:
        result = 100 / num
        print(result)
    except ZeroDivisionError:
        print("Cannot divide by zero.")
except ValueError:
    print("Please enter a valid number.")
```

Here, one try-except block is placed inside another.

1.3 EXCEPTION CLAUSES

3.1 ELSE CLAUSE

The else block executes when **no exception occurs** in the try block.

Example:

```
try:
    num = int(input("Enter a number: "))
    result = 100 / num
except ZeroDivisionError:
    print("Cannot divide by zero.")
else:
    print("Result:", result)
```

If the operation is successful, the else block executes.

Structure

```
try:
    # Risky code
except:
    # Error handling
else:
    # Executes if no exception occurs
```

3.2 FINALLY CLAUSE

The finally block **always executes**, whether an exception occurs or not. It is commonly used for cleanup operations.

Example:

```
try:
    print(10 / 2)
except ZeroDivisionError:
    print("Cannot divide by zero.")
finally:
    print("Program completed.")
```

Output: 5.0

Program completed.

The finally block is useful for tasks such as:

- Closing files
- Releasing resources
- Closing database connections
- Cleanup operations

3.3 RAISE STATEMENT

The raise statement is used to **manually raise an exception**.

Example:

```
age = 15
if age < 18:
    raise ValueError("Age must be 18 or above.")
```

This manually raises a ValueError.

Another Example:

```
marks = -10
if marks < 0:
    raise ValueError("Marks cannot be negative.")
```

3.4 ASSERT STATEMENT

The assert statement is used to check whether a condition is True. If the condition is False, Python raises an AssertionError.

Syntax:

```
assert condition
```

Example:

```
age = 20
assert age >= 18
print("Eligible")
```

Since age >= 18 is True, the program continues.

Example with Message:

```
age = 15
assert age >= 18, "Age must be 18 or above"
```

This produces an AssertionError because the condition is false.

Note: Assertions are mainly useful for detecting programming assumptions during development and testing. They should not normally be used as the main mechanism for validating user input or enforcing runtime business rules.

1.4 CUSTOM EXCEPTION HANDLING

4.1 CREATING USER-DEFINED EXCEPTIONS

A **custom exception** is an exception created by the programmer for a specific situation in an application.

Python allows us to create our own exception classes.

4.2 INHERITING FROM THE EXCEPTION CLASS

A custom exception should generally inherit from Python's Exception class.

Example

```
class AgeError(Exception):
    pass
```

\Here, AgeError is a custom exception.

4.3 RAISING CUSTOM EXCEPTIONS

The raise statement can be used to raise a custom exception.

Example

```
class AgeError(Exception):
    pass
age = 15
if age < 18:
    raise AgeError("Age must be 18 or above.")
```

The program raises:

AgeError: Age must be 18 or above.

4.4 HANDLING A CUSTOM EXCEPTION

Custom exceptions can also be handled using try-except.

```
class AgeError(Exception):
    pass
try:
    age = 15
    if age < 18:
```

```
raise AgeError("Age must be 18 or above.")
except AgeError as e:
    print(e)
```

Output: Age must be 18 or above.

4.5 REAL-WORLD EXAMPLES

Custom exceptions are useful when an application has specific business rules.

Bank Account

```
class InsufficientBalanceError(Exception):
    pass
    balance = 5000
    withdraw = 7000
    try:
        if withdraw > balance:
            raise InsufficientBalanceError("Insufficient balance.")
        balance -= withdraw
        print("Withdrawal successful.")
    except InsufficientBalanceError as e:
        print(e)
```

Student Marks

```
class InvalidMarksError(Exception):
    pass
    marks = 110
    try:
        if marks < 0 or marks > 100:
            raise InvalidMarksError("Marks must be between 0 and 100.")
        print("Valid marks.")
    except InvalidMarksError as e:
        print(e)
```

Online Shopping

```
class OutOfStockError(Exception):
    pass
    stock = 0
    try:
        if stock == 0:
            raise OutOfStockError("Product is out of stock.")
        print("Order placed.")
    except OutOfStockError as e:
        print(e)
```

Practice Questions

1. What is an exception in Python?
2. What is the difference between a syntax error, runtime error, and logical error?
3. Why is exception handling important?
4. What is the purpose of the try block?
5. What is the purpose of the except block?
6. Write a program to handle a ZeroDivisionError.
7. Write a program to handle a ValueError.
8. How can you handle multiple exceptions in Python?
9. What is a generic Exception?
10. What is the purpose of the else clause in exception handling?
11. What is the purpose of the finally clause?
12. Write a program using try, except, and finally.
13. What is the use of the raise statement?
14. What is the purpose of the assert statement?
15. Create and raise a custom exception by inheriting from the Exception class

6.2 FILE HANDLING

2.1 INTRODUCTION TO FILE HANDLING

File handling in Python allows us to create, read, write, update, and manage files stored on a computer.

It is useful when data needs to be stored permanently instead of being lost when a program ends.

1.1 WHAT IS FILE HANDLING?

File handling is the process of working with files using a Python program. Python provides built-in functions and methods to perform operations such as:

- Create files
- Open files
- Read files
- Write data
- Append data
- Update data

Close files

The main function used for file handling is:

`open()`

Example:

```
file = open("data.txt", "r")
```

1.2 IMPORTANCE OF FILE HANDLING

File handling is important because it allows programs to:

- Store data permanently.

- Read previously stored information.

- Save user information.

- Store logs and reports.

- Process large amounts of data.

- Work with text documents and binary files.

- Exchange data between programs.

Real-World Examples

File handling is used for:

- Storing student records

- Saving employee information

- Reading CSV or text files

- Generating reports

- Maintaining application logs

- Storing configuration information

2.3 TYPES OF FILES

There are two common types of files:

- Text Files

- Binary Files

3.1 TEXT FILES

Text files store data as readable characters.

Examples:

- `.txt`

- `.csv`

- `.py`

- `.html`

- Example

```
file = open("data.txt", "r")
```

Text files can contain:

Name: Amit

Age: 25

City: Pune

3.2 BINARY FILES

Binary files store data in binary format rather than ordinary readable text.

Examples include:

.jpg

.png

.mp3

.mp4

.pdf

.exe

Binary files are commonly opened using modes such as rb and wb.

Example: `file = open("image.jpg", "rb")`

2.4 FILE PATHS

A file path specifies the location of a file or folder on a computer.

There are two main types:

Absolute Path

Relative Path

4.1 ABSOLUTE PATH

An absolute path gives the complete location of a file from the root of the file system.

Example:

Windows:

C:\Users\Sonali\Documents\data.txt

Linux:

/home/sonali/Documents/data.txt

Python Example

```
file = open(r"C:\Users\Sonali\Documents\data.txt", "r")
```

The r before the string is a raw string prefix, which helps prevent backslashes from being interpreted as escape characters.

4.2 RELATIVE PATH

A relative path specifies the location of a file relative to the current working directory.

Example:

```
file = open("data.txt", "r")
```

If data.txt is in the current working directory, Python can access it directly.

Another example: file = open("data/data.txt", "r")

2.2 FILE OPERATIONS

Common file operations include:

- Opening
- Reading
- Writing
- Appending
- Updating
- Closing

2.1 OPENING FILES

1.1 OPEN () FUNCTION

The open() function is used to open a file.

Syntax:

```
open(file, mode)
```

Example:

```
file = open("data.txt", "r")
```

Here:

"data.txt" → file name

"r" → read mode

After opening a file, you can perform operations such as reading or writing.

1.2 FILE MODES

File modes determine what operation Python can perform on a file.

Mode	Meaning
r	Read
w	Write
a	Append
x	Create
r+	Read and write

w+	Write and read
a+	Append and read
rb	Read binary
wb	Write binary

r — Read Mode

The r mode is used to read an existing file.

```
file = open("data.txt", "r")
```

If the file does not exist, Python raises FileNotFoundError.

Example:

```
file = open("data.txt", "r")
content = file.read()
print(content)
file.close()
```

w — Write Mode

The w mode is used to write data to a file.

If the file does not exist, Python creates it.

If the file already exists, its existing content is overwritten.

```
file = open("data.txt", "w")
```

Example:

```
file.write("Hello Python")
file.close()
```

a — Append Mode

The a mode is used to add data at the end of an existing file.

If the file does not exist, Python creates it.

```
file = open("data.txt", "a")
```

Example:

```
file.write("\nWelcome to Python")
file.close()
```

The existing content is not overwritten.

x — Create Mode

The x mode is used to create a new file.

```
file = open("newfile.txt", "x")
file.close()
```

If the file already exists, Python raises:
`FileExistsError`

r+ — Read and Write

The `r+` mode allows both reading and writing.
The file must already exist.

```
file = open("data.txt", "r+")
```

Example:

```
print(file.read())
file.write("\nPython")
file.close()
```

w+ — Write and Read

The `w+` mode allows both writing and reading.
If the file exists, its previous contents are removed when opened.

```
file = open("data.txt", "w+")
```

Example:

```
file.write("Python Programming")
file.seek(0)
print(file.read())
file.close()
```

a+ — Append and Read

The `a+` mode allows appending and reading.

```
file = open("data.txt", "a+")
```

Example:

```
file.write("\nNew Data")
file.seek(0)
print(file.read())
file.close()
```

Binary Modes — rb and wb

Binary modes are used for files such as images, audio, videos, and PDFs.

7.1 Read Binary

```
file = open("image.jpg", "rb")
data = file.read()
```

```
file.close()
```

7.2 Write Binary

```
file = open("copy.jpg", "wb")  
file.write(data)  
file.close()
```

READING FILES

Python provides several methods for reading file contents:

```
read()  
readline()  
readlines()
```

2.1 READ()

The read() method reads the contents of a file.

Example:

Suppose data.txt contains:

Hello Python

Welcome to Programming

Python:

```
file = open("data.txt", "r")
```

```
content = file.read()
```

```
print(content)
```

```
file.close()
```

Output:

Hello Python

Welcome to Programming

You can also specify the number of characters to read:

```
content = file.read(5)
```

READLINE()

The readline() method reads one line at a time.

```
file = open("data.txt", "r")
```

```
line = file.readline()
```

```
print(line)
```

```
file.close()
```

If the file contains:

Hello Python

Welcome to Programming
the first readline() reads:
Hello Python

READLINES()

The readlines() method reads all lines and returns them as a list.

```
file = open("data.txt", "r")  
lines = file.readlines()  
print(lines)  
file.close()
```

Example result:

```
['Hello Python\n', 'Welcome to Programming\n']
```

2.3 WRITING FILES

Python provides two commonly used methods for writing data:

```
write()  
writelines()
```

WRITE()

The write() method writes a string to a file.

```
file = open("data.txt", "w")  
file.write("Hello Python")  
file.close()
```

You can write multiple lines using newline characters:

```
file = open("data.txt", "w")  
file.write("Python\n")  
file.write("SQL\n")  
file.write("Power BI\n")  
file.close()
```

WRITELINES()

The writelines() method writes multiple strings to a file.

```
file = open("data.txt", "w")  
lines = [  
    "Python\n",  
    "SQL\n",  
    "Power BI\n"  
]
```

```
file.writelines(lines)
file.close()
```

Important: `writelines()` does not automatically add newline characters. If you want each item on a separate line, include `\n`.

APPENDING DATA

APPEND MODE (A)

Appending means adding new data to the existing content without deleting it.

```
file = open("data.txt", "a")
file.write("\nMachine Learning")
file.close()
```

If the file originally contains:

Python

SQL

After appending:

Python

SQL

Machine Learning

4.2 UPDATING EXISTING FILES

To update a file, you can use modes such as `r+`, `w+`, or `a+`, depending on what you need to do.

For example, using `r+`:

```
file = open("data.txt", "r+")
content = file.read()
print(content)
file.write("\nNew information")
file.close()
```

For more controlled updates, it is common to:

- Read the existing content.

- Modify the content.

- Open the file in write mode.

- Write the updated content.

Example:

```
file = open("data.txt", "r")
content = file.read()
file.close()
```

```
content = content.replace("Python", "Python Programming")
file = open("data.txt", "w")
file.write(content)
file.close()
```

2.3 CLOSING FILES

3.1 CLOSE() METHOD

The close() method is used to close an opened file.

```
file = open("data.txt", "r")
content = file.read()
print(content)
file.close()
```

Closing files is important because it:

Releases system resources.

Ensures data is properly written.

Prevents unnecessary file handles from remaining open.

You can check whether a file is closed using:

```
print(file.closed)
```

USING WITH OPEN() CONTEXT MANAGER

The with open() statement is the recommended way to work with files in most situations.

It automatically closes the file when the block finishes, even if an exception occurs.

Syntax:

```
with open("filename", "mode") as file:
    # File operations
```

Example:

```
with open("data.txt", "r") as file:
    content = file.read()
    print(content)
```

There is no need to explicitly write:

```
file.close()
```

The context manager handles it automatically.

ADVANTAGES OF CONTEXT MANAGERS

Using with open() has several advantages:

3.1. Automatic Closing

The file is automatically closed after the block.

3.2. Safer

It helps ensure that resources are released even when an exception occurs.

3.3. Cleaner Code

It reduces the need for manual close() calls.

3.4. Recommended Practice

It is generally preferred when working with files in Python.

- **Complete Example**

Writing to a file

with open("student.txt", "w") as file:

file.write("Name: Sonali\n")

file.write("Course: Data Analytics\n")

Reading the file

with open("student.txt", "r") as file:

content = file.read()

print(content)

Output:

Name: Sonali

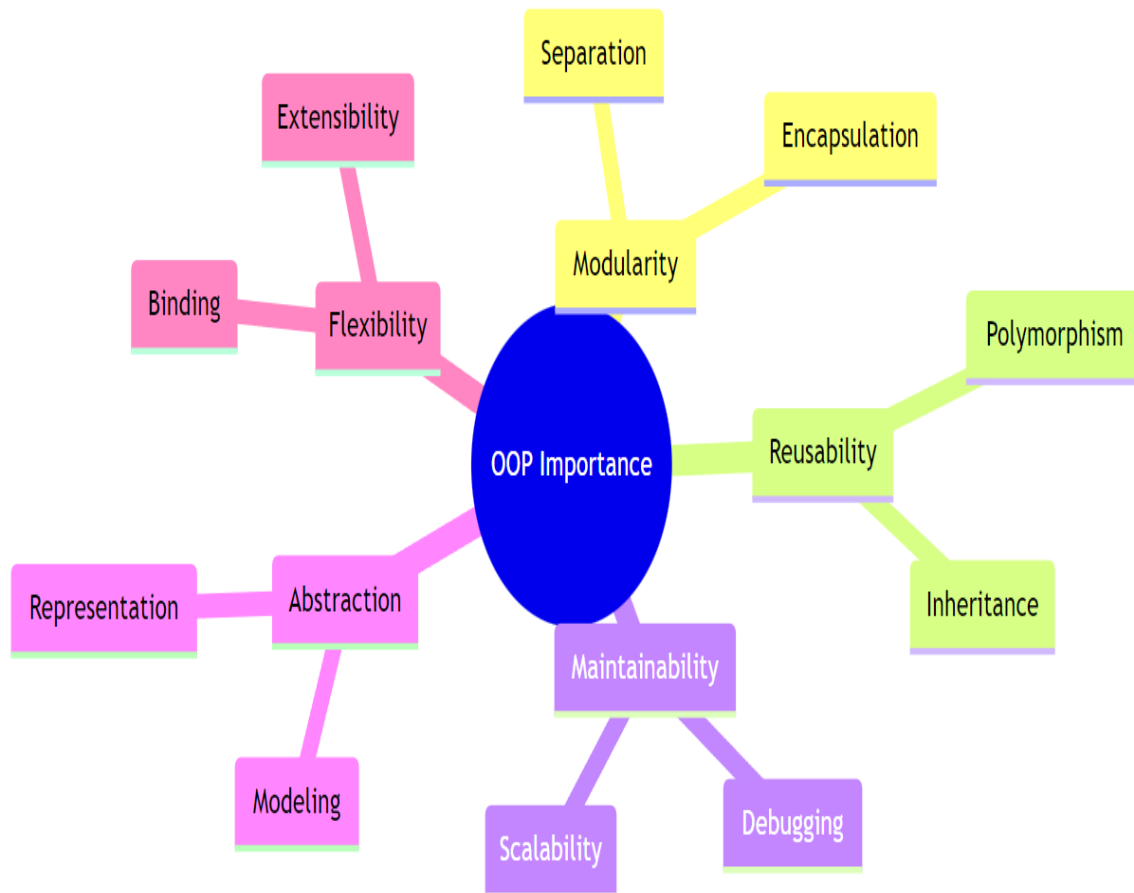
Course: Data Analytics

Practice Questions

1. What is file handling in Python?
2. Why is file handling important?
3. What is the difference between a text file and a binary file?
4. What is the difference between an absolute path and a relative path?
5. What is the purpose of the open() function?
6. What is the difference between file modes r, w, a, and x?
7. Write a program to create and write data into a text file.
8. Write a program to read the complete content of a file using read().
9. What is the difference between read(), readline(), and readlines()?
10. Write a program to write multiple lines using writelines().
11. Write a program to add new data to an existing file using append mode a.
12. What is the difference between r+, w+, and a+ modes?
13. What is the purpose of the close() method?
14. Write a program using with open() to read a file.
15. What are the advantages of using the with open() context manager?



Object-Oriented Programming, or **OOP**, is an important programming concept in Python. It helps us organize code using **classes and objects**. This chapter can be structured as follows.



7.1 INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING

1.1 WHAT IS OBJECT-ORIENTED PROGRAMMING (OOP)?

Object-Oriented Programming (OOP) is a programming approach where programs are designed using objects and classes.

An object represents a real-world entity and contains:

Data → Attributes

Behavior → Methods

- **Real-World Example:**

Consider a Student.

A student can have:

Attributes:

Name

Age
Course
Marks

Methods:

Study
Attend class
Take exam

In Python, we can represent a student using a class and objects.

```
class Student:
    def study(self):
        print("Student is studying")
student1 = Student()
student1.study()
```

Output: Student is studying

- **Real-World Example**

A **Car** can be considered an object.

Attributes:

An attribute is a variable that belongs to a class or an object. It is used to store information or properties about an object.

Color
Brand
Model

These properties are called **attributes**.

Example:

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

Here:

name is an attribute.
age is an attribute.

- **Creating an object:**

```
s1 = Student ("Amit", 25)
print(s1.name)
print(s1.age)
```

Output:

Amit
25

Types of Attributes

Instance Attribute:

An **instance attribute** belongs to a specific object.

Example:

```
class Student:
    def __init__(self, name):
        self.name = name
s1 = Student("Amit")
s2 = Student("Rahul")
print(s1.name)
print(s2.name)
```

Each object can have different values.

Class Attribute

A **class attribute** is shared by all objects of a class.

Example:

```
class Student:
    college = "ABC College"
    def __init__(self, name):
        self.name = name
s1 = Student("Amit")
s2 = Student("Rahul")
print (s1. college)
print (s2. college)
```

Methods:

A **method** is a **function defined inside a class**. Methods describe the **behavior or actions** that an object can perform.

Start

Stop

Drive

Real-world example

For a Car:

color → Attribute

brand → Attribute

start () → Method

stop () → Method

drive () → Method

So:

Attributes store data, while methods perform actions.

Example:

```
class Student:
    def display(self):
        print ("Student Information")
s1 = Student ()
```

```
s1.display()
```

Output: Student Information

Here, display () is a **method**.

Types of Methods in Python

There are mainly **three types of methods**:

1. Instance Method
2. Class Method
3. Static Method

1. Instance Method

An instance method works with the data of a specific object and uses self.

class Student:

Example:

```
def __init__(self, name):  
    self.name = name  
def display(self):  
    print("Name:", self.name)  
s1 = Student("Amit")  
s1.display()
```

Output: Name: Amit

2. Class Method

A class method works with **class-level data** and uses cls.

The @classmethod decorator is used.

Example:

```
class Student:  
    college = "ABC College"  
    @classmethod  
    def show_college(cls):  
        print(cls.college)  
Student.show_college()
```

Output: ABC College

3. Static Method

A static method does not use self or cls. It is used for tasks related to the class but does not need object or class data.

The @staticmethod decorator is used.

Example:

```
class Calculator:
```

```
@staticmethod
def add (a, b):
    return a + b
print (Calculator.add(10, 20))
```

Output: 30

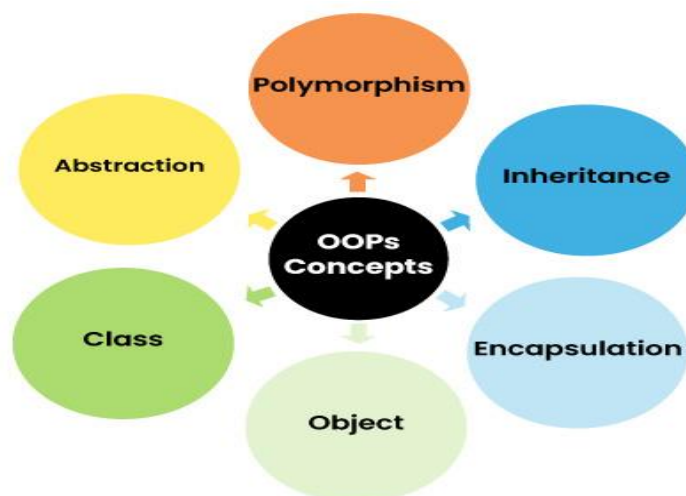
1.2 NEED FOR OOP

OOP is needed to organize large and complex programs.

It helps us:

- Organize code into classes and objects.
- Reuse existing code.
- Reduce code duplication.
- Protect data.
- Make programs easier to maintain.
- Represent real-world entities in programs.
- Make large applications
- easier to develop.

KEY COMPONENTS OF OOP'S



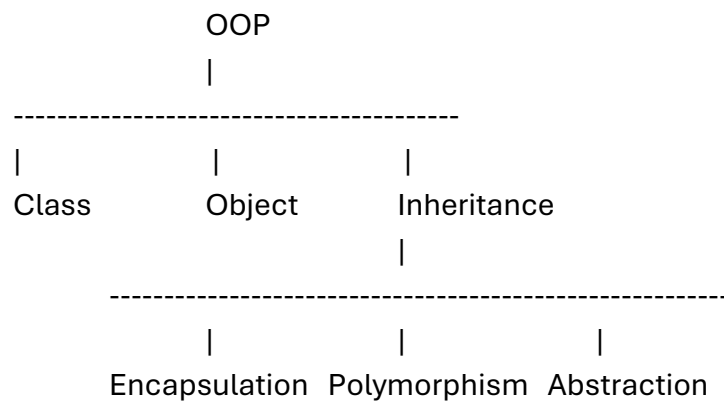
1.3

FEATURES OF OOP

The major features of OOP are:

1. Class
2. Object
3. Encapsulation
4. Inheritance
5. Polymorphism
6. Abstraction
- 7.

Simple Representation



1.4 ADVANTAGES OF OOP

1. **Code Reusability** - Inheritance allows existing code to be reused.
2. **Maintainability** - Code can be organized into separate classes.
3. **Security** - Encapsulation helps control access to data.
4. **Flexibility** - Polymorphism allows the same interface to work with different objects.
5. **Scalability** - OOP is useful for developing large applications.
6. **Easy Debugging** - Problems can often be isolated within specific classes or methods.

1.5 APPLICATIONS OF OOP

OOP is commonly used in:

- Web applications
- Desktop applications
- Mobile applications
- Game development
- Banking systems
- E-commerce applications
- Data processing applications
- Enterprise software
- Artificial Intelligence and Machine Learning systems

7.2 PROCEDURAL PROGRAMMING VS OBJECT-ORIENTED PROGRAMMING

2.1 PROCEDURAL PROGRAMMING

Procedural programming organizes a program mainly around functions and procedures.

The program is generally divided into a sequence of steps.

Example:

```
def add(a, b):
    return a + b
result = add(10, 20)
print(result)
```

The focus is mainly on what operations the program performs.

2.2 OBJECT-ORIENTED PROGRAMMING

OOP organizes programs around classes and objects.

Example:

```
class Calculator:
    def add(self, a, b):
        return a + b
calc = Calculator()
print(calc.add(10, 20))
```

The focus is on objects and their behavior.

2.3 DIFFERENCE BETWEEN PROCEDURAL AND OOP

Procedural Programming	Object-Oriented Programming
Focuses on functions	Focuses on objects
Data and functions are generally separate	Data and methods can be combined in objects
Less suitable for very large systems	Suitable for large applications
Code reuse mainly through functions	Code reuse through inheritance and composition
Generally follows a step-by-step approach	Models entities and their behavior
Examples: C, basic scripting	Examples: Python, Java, C++

2.4 REAL-WORLD EXAMPLES

Procedural Approach

For a bank application, you might have separate functions:

```
deposit()
withdraw()
check_balance()
```

OOP Approach

You can create a BankAccount class containing:

Attributes:

```
account_number
balance
```

Methods:

```
deposit()
withdraw()
check_balance()
```

This keeps related data and behavior together

7.3 CLASSES AND OBJECTS

3.1 WHAT IS A CLASS?

A **class** is a blueprint or template used to create objects. It defines the attributes (data) and methods (actions) that objects can have.

Example:

```
class Student:
    name = "Amit"
    def study(self):
        print("Student is studying")
```

Here, Student is a class.

- **Real-World Example**

Think of a class as a blueprint for a house.

Blueprint → Class

Actual houses built from the blueprint → Objects

Example:

```
class Student:
    def display(self):
        print("Student Information")
```

Here, Student is a **class**, and display () is a **method**.

Creating Objects

```
s1 = Student ()
s2 = Student ()
```

Here, s1 and s2 are **objects** created from the student class.

Class with Attributes and Methods

```
class Student:
    def __init__(self, name):
        self.name = name
    def display(self):
        print("Name:", self.name)
s1 = Student("Amit")
s1.display()
```

Output: Name: Amit

3.2 WHAT IS AN OBJECT?

An object is an instance of a class. It is a real entity created from a class.
A class defines the structure, while an object contains actual data and can use the methods defined in the class.

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

Example:

```
class Student:
    pass
student1 = Student()
Here:
Student → Class
student1 → Object
```

- **Real-World Example**

Think of a class as a blueprint for a car.

Class → Car

Objects → BMW, Audi, Tesla

Here:

Car → Class

BMW, Audi, Tesla → Objects

Example:

```
class Student:
    def display(self):
        print ("Student Information")
Here, Student is a class.
Creating an Object
s1 = Student ()
s1. display ()
```

Here:

s1 is an object

Student is a class

display () is a method

Creating Multiple Objects

```
s1 = Student ()
s2 = Student ()
s3 = Student ()
```

All these objects are created from the same Student class.

Example with Different Data:

```
class Student:
    def __init__(self, name):
        self.name = name
s1 = Student("Amit")
s2 = Student("Rahul")
```

```
print(s1.name)
print(s2.name)
```

Output:

```
Amit
Rahul
```

Example:

```
class Student:
    pass
s1 = Student ()
```

Here, s1 is an object of the student class.

3.3 CREATING A CLASS

The class keyword is used to create a class.

Syntax:

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

Example:

```
class Student:
    name = "Amit"
    def display(self):
        print("Student Name:", self.name)
```

3.4 CREATING OBJECTS

Objects are created by calling the class.

```
student1 = Student()
student2 = Student()
```

Each object is an instance of the Student class.

3.5 ACCESSING CLASS MEMBERS

Members of a class can be accessed using the dot (.) operator.

Example:

```
class Student:
    name = "Amit"
    def display(self):
        print(self.name)
student = Student()
print(student.name)
student.display()
```

Output:

Amit
Amit

3.6 CREATING MULTIPLE OBJECTS

A class can be used to create multiple objects.

Example:

```
class Student:
    def __init__(self, name, age):
        self.name = name
        self.age = age
student1 = Student("shruti", 20)
student2 = Student("Amit", 22)
print(student1.name)
print(student2.name)
```

Output:

Shruti
Amit

Each object can have its own data.

3.7 OBJECT LIFECYCLE

The basic lifecycle of an object is:

Class Definition

↓

Object Creation

↓

Initialization

↓

Object Used

↓

Object No Longer Needed

↓

Object Destroyed / Garbage Collected

Python automatically manages object memory using its memory-management and garbage-collection mechanisms.

7.4 ATTRIBUTES AND METHODS

Attributes

Attributes are variables associated with a class or object. They represent the **data or state** of an object.

There are two important types:

1. Instance Attributes
2. Class Attributes

4.1 INSTANCE ATTRIBUTES

Instance attributes belong to a particular object.

They are usually created using self.

Example:

```
class Student:
    def __init__(self, name, age):
        self.name = name
        self.age = age
student1 = Student("Sonali", 25)
student2 = Student("Amit", 22)
print(student1.name)
print(student2.name)
```

Here:

```
self.name
self.age
```

are instance attributes.

Each object can have different values.

4.2 CLASS ATTRIBUTES

A class attribute belongs to the **class** and is shared by instances unless an instance overrides it.

Example:

```
class Student:
    college = "Wadia College"
    def __init__(self, name):
        self.name = name
student1 = Student("Sonali")
student2 = Student("Amit")
print(student1.college)
print(student2.college)
```

Both objects can access the same class attribute.

7.5 METHODS

Methods are **functions defined inside a class**. They generally represent the behavior of objects.

Python commonly uses three types of methods:

1. Instance Method

2. Class Method
3. Static Method

5.1 INSTANCE METHODS

An instance method works with a particular object.
It normally takes self as its first parameter.

Example:

```
class Student:
    def display(self):
        print("Student details")
student = Student()
student.display()
```

5.2 CLASS METHODS — @classmethod

A class method works with the **class itself** rather than a particular instance.
It uses the @classmethod decorator and normally takes cls as its first parameter.

Example:

```
class Student:
    college = "Wadia College"
    @classmethod
    def show_college(cls):
        print(cls.college)
Student.show_college()
```

Output: Wadia College

5.3 STATIC METHODS — @staticmethod

A static method does not require self or cls.
It is generally used for a utility operation related to the class.

Example:

```
class Calculator:
    @staticmethod
    def add(a, b):
        return a + b
print(Calculator.add(10, 20))
```

Output: 30

7.6 SPECIAL KEYWORDS

6.1 SELF KEYWORD

self refers to the **current object/instance**.

It is used to access instance attributes and instance methods.

Example:

```
class Student:
    def __init__(self, name):
        self.name = name
    def display(self):
        print(self.name)
student = Student("Sonali")
student.display()
```

Here:

```
self.name
```

refers to the name attribute of the current object.

Important: self is a conventional parameter name in Python instance methods. It is not a reserved keyword.

6.2 CLS KEYWORD

cls is the conventional name for the first parameter of a class method.

It refers to the **class itself**.

Example:

```
class Student:
    college = "Wadia College"
    @classmethod
    def show(cls):
        print(cls.college)
Student.show()
```

Important: Like self, cls is a convention, not a Python reserved keyword.

7.7 CONSTRUCTORS

7.1 INTRODUCTION TO CONSTRUCTORS

A constructor is a special method used to **initialize an object when it is created**.

In Python, the commonly used initialization method is:

```
__init__()
```

7.2 PURPOSE OF CONSTRUCTORS

Constructors are used to:

- Initialize object attributes.

- Set initial values.
- Perform setup required when an object is created.

7.3 __INIT__() METHOD

The `__init__()` method is automatically called when an object is created.

```
class Student:
    def __init__(self, name, age):
        self.name = name
        self.age = age
student = Student("Amit", 25)
print(student.name)
print(student.age)
```

Output:

```
Amit
25
```

7.4 DEFAULT CONSTRUCTOR

A class can have an `__init__()` method that takes no arguments other than `self`.

Example:

```
class Student:
    def __init__(self):
        self.name = "Amit"
        self.age = 25
student = Student()
print(student.name)
```

Output: Amit

If you don't define `__init__()`, Python can provide a default initialization behavior inherited from the object hierarchy, provided no initialization is otherwise required.

7.5 PARAMETERIZED CONSTRUCTOR

A parameterized constructor accepts values when an object is created.

Example:

```
class Student:
    def __init__(self, name, age):
        self.name = name
        self.age = age
student = Student("Amit", 25)
print(student.name)
print(student.age)
```

7.6 CONSTRUCTOR WITH DEFAULT ARGUMENTS

Constructor parameters can have default values.

Example:

```
class Student:
    def __init__(self, name="Unknown", age=0):
        self.name = name
        self.age = age
student1 = Student()
student2 = Student("Amit", 25)
print(student1.name)
print(student2.name)
```

Output:

```
Unknown
Amit
```

7.7 CONSTRUCTOR CHAINING USING SUPER()

super() is commonly used to call a method from a parent class.

It is especially useful when a child class needs to initialize attributes defined by its parent.

Example:

```
class Person:
    def __init__(self, name):
        self.name = name
class Student(Person):
    def __init__(self, name, course):
        super().__init__(name)
        self.course = course
student = Student("Sonali", "Data Analytics")
print(student.name)
print(student.course)
```

Output:

```
Sonali
Data Analytics
```

Here:

```
super().__init__(name)
```

calls the parent class constructor.

7.8 INHERITANCE

8.1 INTRODUCTION TO INHERITANCE

Inheritance is an OOP feature that allows one class to acquire attributes and methods from another class.

The existing class is called the **parent/base/superclass**, and the new class is called the **child/derived/subclass**.

Example:

```
class Animal:
    def eat(self):
        print("Animal eats")
class Dog(Animal):
    def bark(self):
        print("Dog barks")
dog = Dog()
dog.eat()
dog.bark()
```

Output:

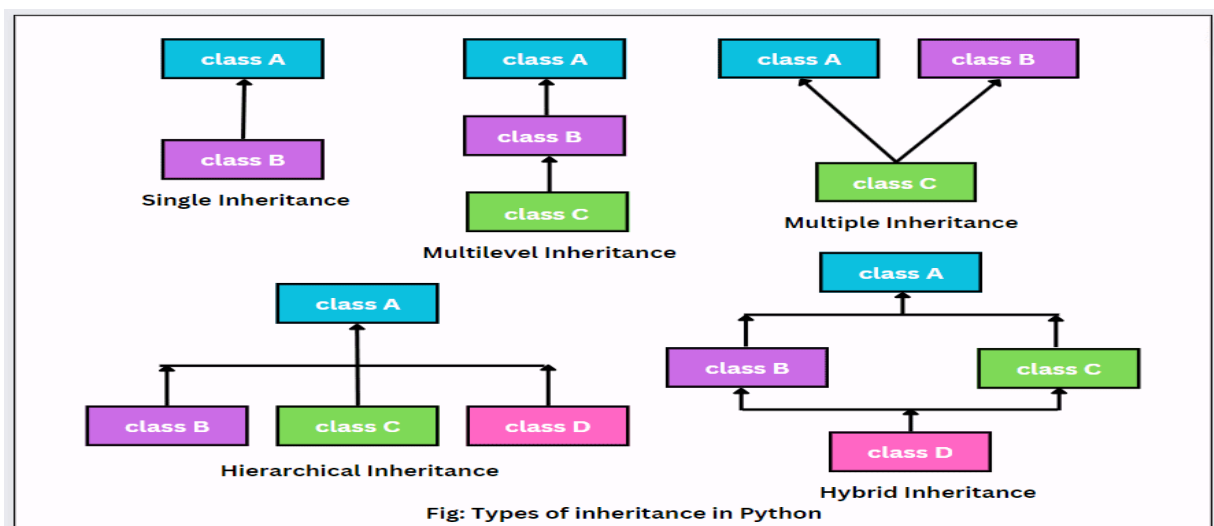
```
Animal eats
Dog barks
```

8.2 BENEFITS OF INHERITANCE

- Code reusability
- Reduced duplication
- Easier maintenance
- Supports hierarchical relationships
- Allows method overriding
- Helps create specialized classes

7.9 TYPES OF INHERITANCE

Python supports several inheritance patterns.



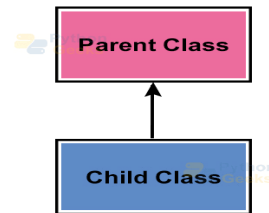
9.1 SINGLE INHERITANCE

One child class inherits from one parent class.

Example:

```
class Animal:
    def eat(self):
        print("Eating")
class Dog(Animal):
    def bark(self):
        print("Barking")
```

Single Inheritance



Single Inheritance

```
[33]: class Employee:
        def show_details(self):
            return "Employee Name: Nick"

        class Developer(Employee):
            def role(self):
                return "Role: Python Developer"

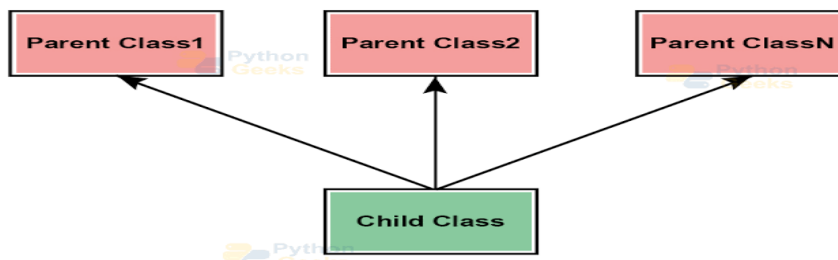
    d = Developer()
    print(d.show_details())
    print(d.role())
```

```
Employee Name: Nick
Role: Python Developer
```

9.2 MULTIPLE INHERITANCE

One child class inherits from multiple parent classes.

Multiple Inheritance in Python



Example:

```
class Father:
    def skill1(self):
        print("Driving")
class Mother:
    def skill2(self):
        print("Cooking")
class Child(Father, Mother):
    pass
child = Child()
```

```
child.skill1()
child.skill2()
```

MULTIPLE INHERITANCE

```
[35]: class Teacher:
        def teach(self):
            return "Teaches Subjects"

        class Researcher:
            def research(self):
                return "Conducts Research Work"

        class Professor(Teacher, Researcher):
            def position(self):
                return "Position: Professor"

    p = Professor()
    print(p.teach())
    print(p.research())
    print(p.position())
```

```
Teaches Subjects
Conducts Research Work
Position: Professor
```

9.3 MULTILEVEL INHERITANCE

A class inherits from another derived class, creating a chain.

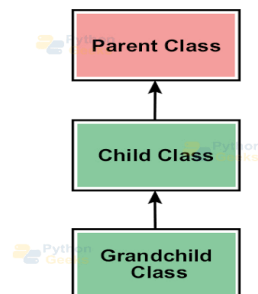
Example:

```
class Grandparent:
    def house(self):
        print("House")

class Parent(Grandparent):
    def car(self):
        print("Car")

class Child(Parent):
    def bike(self):
        print("Bike")
```

Multilevel Inheritance



The Child can access methods inherited through the chain.

Multilevel Inheritance

```
[34]: class Vehicle:
        def category(self):
            return "Category: Vehicle"

        class Car(Vehicle):
            def type(self):
                return "Car Type: Sedan"

        class SportsCar(Car):
            def speed(self):
                return "Top Speed: 300 km/h"

    s = SportsCar()
    print(s.category())
    print(s.type())
    print(s.speed())
```

```
Category: Vehicle
Car Type: Sedan
Top Speed: 300 km/h
```

9.4 HIERARCHICAL INHERITANCE

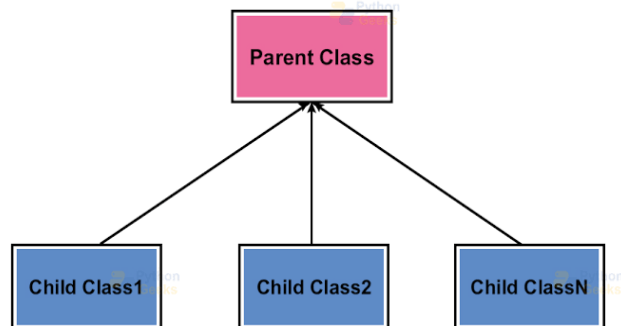
Multiple child classes inherit from the same parent.

Example:

```
class Animal:
    def eat(self):
        print("Eating")
class Dog(Animal):
    pass
class Cat(Animal):
    pass
```

Both Dog and Cat inherit from Animal.

Hierarchical Inheritance



HIERARCHICAL INHERITANCE

```
[36]: class Bank:
        def bank_name(self):
            return "Bank: SBI"

        class SavingsAccount(Bank):
            def interest(self):
                return "Savings Account Interest: 5%"

        class CurrentAccount(Bank):
            def overdraft(self):
                return "Overdraft Limit: ₹50,000"

s = SavingsAccount()
c = CurrentAccount()

print(s.bank_name())
print(s.interest())

print(c.bank_name())
print(c.overdraft())
```

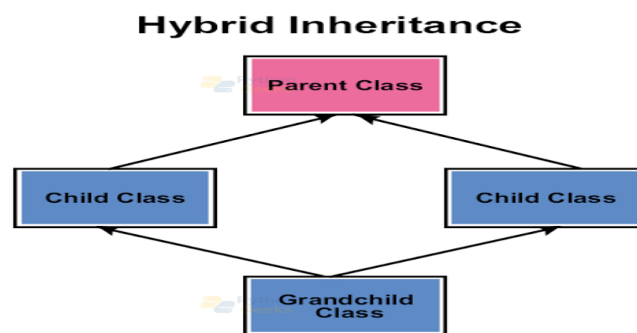
```
Bank: SBI
Savings Account Interest: 5%
Bank: SBI
Overdraft Limit: ₹50,000
```

9.5 HYBRID INHERITANCE

Hybrid inheritance is a combination of two or more inheritance patterns.

For example, a design may combine hierarchical and multiple inheritance.

Python supports such structures, but complex inheritance hierarchies should be designed carefully.



Hybrid Inheritance

```
[37]: class Staff:
        def work(self):
            return "General Staff Work"

        class Teacher(Staff):
            def teach(self):
                return "Teaches Students"

        class Admin(Staff):
            def manage(self):
                return "Handles Administration"

        class Coordinator(Teacher, Admin):
            def role(self):
                return "Role: Academic Coordinator"

        c = Coordinator()
        print(c.work())           # from Staff
        print(c.teach())          # from Teacher
        print(c.manage())         # from Admin
        print(c.role())           # own method
```

```
General Staff Work
Teaches Students
Handles Administration
Role: Academic Coordinator
```

7.10 RELATED CONCEPTS

10.1 METHOD OVERRIDING

Method overriding occurs when a child class provides its own implementation of a method inherited from the parent class.

Example:

```
class Animal:
    def sound(self):
        print("Animal makes a sound")
class Dog(Animal):
    def sound(self):
        print("Dog barks")
dog = Dog()
dog.sound()
```

Output: Dog barks

The Dog class overrides the sound() method.

Example:

```
class Parent:
    def display(self):
        print("Parent")
class Child (Parent):
    def display(self):
        print("Child")
obj = Child ()
obj.display()
```

Output: Child

10.2 SUPER() METHOD

super() is used to access methods or initialization behavior from a parent class.

Example:

```
class Animal:
    def sound(self):
        print("Animal sound")
class Dog(Animal):
    def sound(self):
        super().sound()
        print("Dog barks")
dog = Dog()
dog.sound()
```

Output:

Animal sound

Dog barks

Example:

```
class Child (Parent):  
    def display(self):  
        super (). display ()  
        print("Child")
```

super() function

```
[38]: class A:  
        def __init__(self):  
            print("A constructor")  
  
        class B(A):  
            def __init__(self):  
                super().__init__() # calling parent construct  
                print("B constructor")  
  
        # creating object of class B  
        b = B()  
  
A constructor  
B constructor
```

10.3 METHOD RESOLUTION ORDER (MRO)

MRO defines the order in which Python searches classes when looking for a method or attribute, especially in inheritance.

MRO defines the order in which Python searches for methods in multiple inheritance.

You can view the MRO using:

ClassName.mro()

Example:

```
class A:  
    pass  
class B(A):  
    pass  
print(B.mro())
```

The output shows the order in which Python searches for members.

MRO is particularly important in multiple inheritance.

7.11 POLYMORPHISM

11.1 INTRODUCTION TO POLYMORPHISM

Polymorphism means "many forms."

In OOP, the same method name or interface can behave differently depending on the object being used.

11.2 METHOD OVERRIDING — RUN-TIME POLYMORPHISM

Different classes can implement the same method differently.

Example:

```
class Dog:
    def sound(self):
        print("Dog barks")
class Cat:
    def sound(self):
        print("Cat meows")
for animal in [Dog(), Cat()]:
    animal.sound()
```

Output:

```
Dog barks
Cat
```

meowsThe same sound() call produces different behavior.

1. Method Overriding

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

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

        b = B()
        b.show()

Child
```

Example:

```
class Animal:
    def sound(self):
        print ("Animal makes a sound")
class Dog (Animal):
    def sound(self):
        print ("Dog barks")
obj = Dog ()
obj.sound()
```

Output: Dog barks

11.3 METHOD OVERLOADING USING DEFAULT ARGUMENTS

Traditional method overloading, where multiple methods have the same name but different parameter lists, is not directly supported in Python in the same way as languages such as Java.

A common Python approach is to use **default arguments** or flexible arguments.

2. Method Overloading

```
[40]: class Test:
      def add(self, a=0, b=0, c=0):
          return a + b + c

      t = Test()

      print(t.add(5))
      print(t.add(5, 10))
      print(t.add(5, 10, 15))

      5
      15
      30
```

Example:

```
class Calculator:
    def add(self, a, b=0, c=0):
        return a + b + c

    calc = Calculator()
    print(calc.add(10))
    print(calc.add(10, 20))
    print(calc.add(10, 20, 30))
```

Output:

```
10
30
60
```

The same method can handle different numbers of arguments.

11.4 DUCK TYPING

Duck typing means Python focuses on whether an object supports the required operation rather than its specific class type.

A common idea is:

If an object behaves like the required type, it can be used.

Example:

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

class Cat:
```

```

def sound(self):
    print("Meow")
def make_sound(animal):
    animal.sound()
    make_sound(Dog())
    make_sound(Cat())

```

Both objects work because they provide a sound() method.

11.5 OPERATOR OVERLOADING

Operator overloading allows operators such as +, -, ==, etc. to work with user-defined objects by implementing special methods.

Operator overloading means the **same operator performs different operations depending on the data type**.

Example:

```

class Number:
    def __init__(self, value):
        self.value = value
    def __add__(self, other):
        return self.value + other.value
num1 = Number(10)
○ num2 = Number(20)
print(num1 + num2)

```

Output: 30

Here:

`__add__()`

defines the behavior of the + operator for the objects.

3. Operator Overloading

```

[41]: class Point:
        def __init__(self, x, y):
            self.x = x
            self.y = y

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

p1 = Point(10, 20)
p2 = Point(5, 15)

result = p1 + p2    # calls __add__()
print("Result:", result.x, result.y)

Result: 15 35

```

Example:

`print(10 + 20)`

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

Output: 30

Hello Python

Method Overriding – Run-Time Polymorphism

Example:

```
class Animal:
    def sound(self):
        print("Animal Sound")
class Dog(Animal):
    def sound(self):
        print("Bark")
class Cat(Animal):
    def sound(self):
        print("Meow")
```

The sound() method behaves differently for different objects.

Method Overloading Using Default Arguments

Python does not support traditional method overloading in the same way as some languages.

However, similar behavior can be achieved using default arguments.

Example:

```
class Calculator:
    def add (self, a, b=0, c=0):
        return a + b + c
obj = Calculator ()
print(obj.add(10))
print (obj.add(10, 20))
print (obj.add(10, 20, 30))
```

7.12 ENCAPSULATION

12.1 INTRODUCTION TO ENCAPSULATION

Encapsulation means **combining data (attributes) and methods (functions) into a single unit**, usually a **class**.

It also helps **control access to data** and protects it from unwanted modification.

Encapsulation means combining data and methods within a class and controlling how the internal data is accessed or modified.

It helps protect the internal state of an object.

Example:

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

```

        self.name = name
        self.age = age
    def display(self):
        print ("Name:", self.name)
        print ("Age:", self.age)

```

Here:

name and age → **Data / Attributes**

display() → **Method**

Student → **Single unit (Class)**

The user cannot directly control all internal data. Instead, they interact with it through specific methods.

Data Hiding (Encapsulation)

```

[3]: class Bank:
      def __init__(self):
          self.__balance = 10000 # private variable

      def get_balance(self):
          return self.__balance

      def deposit(self, amount):
          self.__balance += amount

```

```

[4]: b = Bank()
      print(b.get_balance()) # allowed
      b.deposit(5000)
      print(b.get_balance())

```

```

10000
15000

```

12.2 PUBLIC MEMBERS

Public members can be accessed directly from outside the class.

Public members can be accessed from anywhere.

Example:

```

class Student:
    def __init__(self, name):
        self.name = name
    student = Student("Amit")
    print(student.name)

```

name is a public attribute.

Example:

```

class Student:
    def __init__(self):
        self.name = "Amit"
    obj = Student ()
    print(obj.name)

```

12.3 PROTECTED MEMBERS (_)

A single underscore is used by convention to indicate that a member is intended for internal or subclass use.

A private member starts with double underscores.

Example:

```
class Student:
    def __init__(self):
        self._marks = 85
```

Example:

```
class Student:
    def __init__(self):
        self._name = "Amit"
```

It is intended for internal use and subclasses.

Python does not enforce strict protected access. The underscore is mainly a naming convention.

12.4 PRIVATE MEMBERS (__)

A double underscore at the beginning triggers name mangling, making direct access using the original name more difficult.

A private member starts with double underscores.

Example:

```
class Student:
    def __init__(self):
        self.__marks = 85
    def show_marks(self):
        print(self.__marks)
student = Student()
student.show_marks()
```

Output: 85

Direct access like:

```
student.__marks
```

Example:

```
class Student:
    def __init__(self):
        self.__name = ""
    def set_name(self, name):
        self.__name = name
```

does not work normally because of name mangling.

12.5 ACCESS SPECIFIERS IN PYTHON

Python does not have strict access specifiers like some other languages.

Instead, naming conventions are commonly used:

Syntax	Meaning
name	Public
_name	Protected by convention
__name	Private-like through name mangling

12.6 GETTER METHODS

A getter method is used to retrieve the value of an attribute.

Getter methods are used to access data.

Example:

```
class Student:
    def __init__(self, marks):
        self.__marks = marks
    def get_marks(self):
        return self.__marks
student = Student(85)
print(student.get_marks())
```

Example:

```
class Student:
    def __init__(self):
        self.__name = "Sonali"
    def get_name(self):
        return self.__name
```

12.7 SETTER METHODS

A setter method is used to modify the value of an attribute.

Setter methods are used to modify data.

Example:

```
class Student:
    def __init__(self, marks):
        self.__marks = marks
    def get_marks(self):
        return self.__marks
    def set_marks(self, marks):
        if 0 <= marks <= 100:
            self.__marks = marks
student = Student(85)
student.set_marks(90)
print(student.get_marks())
```

Output:90

Example:

```
class Student:
    def __init__(self):
        self.__name = ""
    def set_name(self, name):
        self.__name = name
```

The setter can also validate the value before changing it.

12.8 PROPERTY DECORATOR — @PROPERTY

The @property decorator allows a method to be accessed like an attribute. The @property decorator provides a Pythonic way to use getter and setter methods.

Example:

```
class Student:
    def __init__(self, marks):
        self.__marks = marks
    @property
    def marks(self):
        return self.__marks
    @marks.setter
    def marks(self, value):
        if 0 <= value <= 100:
            self.__marks = value
        else:
            raise ValueError("Marks must be between 0 and 100.")
student = Student(85)
print(student.marks)
student.marks = 90
print(student.marks)
Output:
85
90
```

Example:

```
class Student:
    def __init__(self):
        self.__name = ""
    @property
```

```
def name(self):
    return self.__name
@name.setter
def name(self, value):
    self.__name = value
Usage:
obj = Student ()
obj.name = "Amit"
print(obj.name)
```

This provides controlled access while keeping a simple attribute-like syntax.

Benefits of Encapsulation

- Protects data
- Controls data access
- Improves code organization
- Makes programs easier to maintain

7.13 ABSTRACTION

13.1 WHAT IS ABSTRACTION?

Abstraction means hiding unnecessary implementation details and exposing only the essential functionality.

Abstraction focuses on what an object does rather than how it works internally.

Real-World Example:

When using an ATM, you select:

Withdraw Money

Deposit Money

Check Balance

You don't need to know the internal implementation of the banking system.

That is the basic idea of abstraction.

13.2 WHY ABSTRACTION?

Abstraction helps:

- Hide implementation details.
- Reduce complexity.
- Define a common interface.
- Improve code organization.
- Make systems easier to use and maintain.

13.3 ABSTRACT CLASSES

An abstract class is a class designed to be inherited from rather than directly instantiated when it contains abstract methods.

Python provides abstract class functionality through the abc module.

An abstract class is a class that acts as a blueprint for other classes.

It may contain one or more abstract methods.

```
from abc import ABC, abstractmethod  
class Animal (ABC):  
    pass
```

13.4 ABSTRACT METHODS

An abstract method defines a method that subclasses are expected to implement.

An abstract method is a method that is declared in the parent class but must be implemented by its child class.

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

Example:

```
class Animal(ABC):  
    @abstractmethod  
    def sound(self):  
        pass
```

Here, sound() is an abstract method.

A concrete child class should implement it.

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

```
class Animal(ABC):  
    @abstractmethod  
    def sound(self):  
        pass  
class Dog(Animal):  
    def sound(self):  
        print("Dog barks")  
dog = Dog()  
dog.sound()  
Output: Dog barks
```

13.5 ABC MODULE

abc stands for Abstract Base Classes.

it is a built-in Python module used to create abstract base classes.

It provides tools for creating:

- Abstract classes
- Abstract methods

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

Important components include:

- ABC
- abstractmethod

13.6 ABC CLASS

ABC is a base class provided by the abc module.

A class can inherit from ABC to become an abstract base class.

The ABC class is used as a base class for creating an abstract class.

Example:

```
from abc import ABC
class Vehicle(ABC):
    pass
from abc import ABC
class Animal(ABC):
    pass
```

Here, Animal is an abstract base class

13.7 @ABSTRACTMETHOD DECORATOR

The @abstractmethod decorator is used to declare an abstract method.

from abc import ABC, abstractmethod

- Example:**
- class Vehicle(ABC):
 - **@abstractmethod**
 - def start(self):
 - pass

Any concrete subclass must implement start() before it can normally be instantiated.

Complete Example:

```
from abc import ABC, abstractmethod
class Vehicle(ABC):
    @abstractmethod
    def start(self):
        pass
class Car(Vehicle):
    def start(self):
        print("Car starts with a key")
class ElectricCar(Vehicle):
    def start(self):
        print("Electric car starts with a button")
car = Car()
```

```
electric_car = ElectricCar()
car.start()
electric_car.start()
```

Output:

Car starts with a key

Electric car starts with a button

The abstract class defines the required interface, while each child class provides its own implementation.

Example:

```
from abc import ABC, abstractmethod
class Animal (ABC):
    @abstractmethod
    def sound(self):
        pass
```

Any child class must provide an implementation for sound ().

Complete Example:

```
from abc import ABC, abstractmethod
class Animal (ABC):
    @abstractmethod
    def sound(self):
        pass
class Dog (Animal):
    def sound(self):
        print ("Dog barks")
obj = Dog ()
obj.sound()
```

Output: Dog bark

Practice Questions

1. What is Object-Oriented Programming (OOP)?
2. Why do we use OOP in Python?
3. What are the main features of OOP?
4. What is the difference between procedural programming and object-oriented programming?
5. What is a class? Give a real-world example.
6. What is an object? How is it different from a class?
7. Write a Python program to create a class and an object.
8. What are attributes and methods in OOP?
9. What is the difference between an instance attribute and a class attribute?

10. What is the difference between an instance method, class method, and static method?
11. What is the purpose of the self keyword?
12. What is the purpose of the cls keyword?
13. What is a constructor? What is the purpose of the __init__() method?
14. Write a program using a parameterized constructor to store student details.
15. What is inheritance? What are its advantages?
16. Name and explain the different types of inheritance in Python.
17. What is method overriding? Write a simple example.
18. What is polymorphism in Python?
19. What is encapsulation? Explain public, protected, and private members.
20. What is abstraction? What is the purpose of an abstract class and the @abstractmethod decorator?



This is an excellent chapter for your Python book. Here is a **brief, beginner-friendly explanation** of each topic.

Modules, packages, and libraries help us **organize Python code, reuse existing code, and build large applications efficiently.**

8.1 INTRODUCTION TO MODULES

1.1 WHAT IS A MODULE?

A module is a Python file (.py) that contains reusable code such as:

- Variables
- Functions
- Classes
- Statements

A module allows us to write code once and use it in another Python program.

Example:

Create a file named calculator.py:

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

Now this file is a module.

Example:

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

Here, calculator.py is a module.

1.2 WHY USE MODULES?

Modules are useful because they:

1. **Reuse code** – Write code once and use it multiple times.
2. **Organize code** – Divide a large program into smaller files.
3. **Improve readability** – Each module can handle a specific task.
4. **Make maintenance easier** – Changes can be made in one place.
5. **Avoid code duplication.**
6. **Allow teamwork** – Different developers can work on different modules.

Example:

Instead of putting all operations into one large file:

```
project/
|
├── calculator.py
├── database.py
├── validation.py
└── main.py
```

Each module handles a specific responsibility.

1.3 TYPES OF MODULES

Python modules can be broadly classified into three types:

1. Built-in Modules
2. User-Defined Modules
3. Third-Party Modules.

3.1 BUILT-IN MODULES

Built-in modules are modules provided with Python.

Examples:

```
import math
import random
import datetime
import sys
import statistics
```

Example:

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

Output: 5.0

3.2 USER-DEFINED MODULES

These are modules created by the programmer.

For example, create:

```
calculator.py
with:
def add(a, b):
    return a + b
```

Another Python file can import it:

```
import calculator
print(calculator.add(10, 20))
```

Output: 30

THIRD-PARTY MODULES

Third-party modules are created by external developers or organizations and are installed separately.

Examples:

NumPy
Pandas
Matplotlib
Seaborn
Scikit-learn

Example:

```
import pandas as pd
```

Installation is commonly done using:

```
pip install pandas
```

8.2 IMPORTING MODULES

Importing a module means making its code available in another Python program.

2.1 IMPORT STATEMENT

Used to import an entire module.

Syntax:

```
import module_name
```

Example:

```
import math
print(math.sqrt(16))
```

Output: 4.0

The dot operator is used to access functions or variables from the module.

```
math.sqrt()
math.pi
```

2.2 FROM ... IMPORT

We can import specific functions or variables from a module.

Used to import specific functions or objects.

Syntax:

```
from module_name import item
```

Example:

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

Output: 5.0

We don't need to write:

```
math.sqrt()
```

because sqrt was imported directly.

Import Multiple Items

```
from math import sqrt, pi
print(sqrt(25))
print(pi)
```

2.3 IMPORT ... AS

The as keyword gives a module a shorter or alternative name.

Syntax:

```
import module_name as alias
```

Example:

```
import math as m
print(m.sqrt(36))
```

Output: 6.0

A common example is:

```
import pandas as pd
import numpy as np
```

2.4 IMPORTING MULTIPLE MODULES

Multiple modules can be imported separately:

```
import math
import random
import statistics
```

It is generally clearer to keep imports separate.

2.5 RELOADING MODULES

The reload () function can reload a module.

When a module is already imported, Python normally uses the loaded module rather than executing the module again.

To explicitly reload a module during development:

```
import importlib
import calculator
importlib.reload(calculator)
```

importlib.reload() is mainly useful when developing or testing modules interactively.

This is mainly useful when working interactively or after modifying a module.

2.6 MODULE SEARCH PATH (SYS.PATH)

Python searches specific locations to find modules.

When Python imports a module, it searches specific directories to find that module.

Python's module search locations can be viewed using:

```
import sys
print(sys.path)
```

sys.path contains directories where Python searches for modules.

sys.path displays the list of locations where Python searches for modules.

Example:

```
import sys
for path in sys.path:
    print(path)
```

This helps understand where Python looks for imported modules.

8.3 CREATING USER-DEFINED MODULES

3.1 CREATING A MODULE

A user-defined module is simply a Python file containing reusable code.

Example:

```
Create:
calculator.py
def add(a, b):
    return a + b
def multiply(a, b):
    return a * b
```

Example:

Create a Python file:

```
calculator.py
```

Add functions:

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

3.2 USING CUSTOM MODULES

Create another file:

```
main.py
Then:
import calculator
print(calculator.add(10, 20))
print(calculator.multiply(5, 4))
```

Output:

30

20

Create another Python file:

```
import calculator
print (calculator.add(10, 20))
```

Output: 30

3.3 ORGANIZING CODE INTO MODULES

A large application can be divided into modules according to functionality.

Example:

```
project/
|
├── main.py
├── calculator.py
├── validation.py
├── database.py
└── reports.py
```

Possible responsibilities:

Module	Purpose
main.py	Main program
calculator.py	Calculations
validation.py	Input validation
database.py	Database operations
reports.py	Report generation

This makes the application easier to understand and maintain

3.4 BEST PRACTICES FOR MODULES

Follow these practices:

1. Use meaningful names

Good:

```
calculator.py
database.py
employee.py
```

Avoid unclear names such as:

```
abc.py
xyz.py
test123.py
```

2. Keep related code together

Functions performing similar tasks should generally belong to the same module.

3. Avoid unnecessary global variables

Prefer functions and classes where appropriate.

4. Avoid circular imports

For example:

A imports B

B imports A

This can cause import problems.

5. Keep modules focused

A module should ideally have a clear responsibility.

8.4 PACKAGES

4.1 WHAT IS A PACKAGE?

A package is a way of organizing related Python modules into a directory structure.

For example:

```
project/
|
|-- main.py
|
|-- calculations/
|   |-- __init__.py
|   |-- arithmetic.py
|   |-- statistics.py
```

Here, calculations is a package containing multiple modules.

Simple Difference

Module → Python file

Package → Directory containing related modules

4.2 CREATING PACKAGES

Example structure:

```
myproject/
|
|-- main.py
|
|-- calculator/
|   |-- __init__.py
|   |-- addition.py
```

```
└─ subtraction.py
```

addition.py

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

subtraction.py

```
def subtract(a, b):  
    return a - b
```

main.py

```
from calculator.addition import add  
from calculator.subtraction import subtract  
print(add(10, 5))  
print(subtract(10, 5))
```

Output:

15

5

9.3 PACKAGE STRUCTURE

A typical package can look like:

```
project/  
├─  
└─ main.py  
├─  
└─ mypackage/  
    ├─ __init__.py  
    ├─ module1.py  
    ├─ module2.py  
    └─ module3.py
```

Here:

```
mypackage → package  
module1.py → module  
module2.py → module  
module3.py → module  
__init__.py → package initialization file
```

4.4 __INIT__.PY

The `__init__.py` file is traditionally used to identify a directory as a Python package and can also contain package initialization code.

`__init__.py` is a special file associated with Python packages.

It can be used to:

- Initialize a package
- Define package-level variables
- Expose selected functions/classes
- Control what is available through certain package imports

Example:

```
calculator/
|
├── __init__.py
├── addition.py
└── subtraction.py

__init__.py can contain:
print("Calculator package loaded")
```

Important Note

In modern Python, a directory can also function as a **namespace package** without an `__init__.py` file. However, `__init__.py` is still commonly used when creating regular packages because it provides explicit package initialization and organization.

4.5 IMPORTING FROM PACKAGES

Import a module

```
import calculator.addition
```

Import a specific function

```
from calculator.addition import add
```

Use alias

```
from calculator.addition import add as addition
print(addition(5, 10))
```

4.6 ADVANTAGES OF PACKAGES

Packages provide:

1. Better code organization
2. Reusability
3. Easier maintenance
4. Namespace management
5. Separation of functionality
6. Better project structure
7. Easier collaboration

8.5 BUILT-IN PYTHON LIBRARIES

Python provides many useful modules as part of its standard library.

Python provides many useful built-in libraries.

Some important modules are:

```
math
random
datetime
sys
statistics
```

MATH MODULE

The math module provides mathematical functions and constants.

Import:

math

1.1 MATHEMATICAL FUNCTIONS

Square Root

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

Output:5.0

Absolute Value

```
print(math.fabs(-10))
```

Output: 10.0

Factorial

```
print(math.factorial(5))
```

Output:120

1.2 CONSTANTS: PI AND E

math.pi

Represents the mathematical constant π .

```
import math
print(math.pi)
```

math.e

Represents Euler's number.

```
print(math.e)
```

1.3 POWER AND SQUARE ROOT

math.pow()

```
import math
print(math.pow(2, 3))
```

Output: 8.0

math.sqrt()

```
print(math.sqrt(49))
```

Output: 7.0

For simple integer exponentiation, Python's `**` operator is also commonly used:

```
print(2 ** 3)
```

Output: 8

1.4 TRIGONOMETRIC FUNCTIONS

The math module provides:

```
sin()
```

```
cos()
```

```
tan()
```

Example:

```
import math
print(math.sin(math.pi / 2))
print(math.cos(0))
print(math.tan(math.pi / 4))
```

Output is approximately:

```
1.0
```

```
1.0
```

```
1.0
```

The trigonometric functions use radians.

1.5 LOGARITHMIC FUNCTIONS

Natural Log

```
import math
print(math.log(10))
```

Log Base 10

```
print(math.log10(100))
```

Output: 2.0

Log Base 2

```
print(math.log2(8))
```

Output: 3.0

1.6 ROUNDING FUNCTIONS

ceil()

Rounds a number upward.

```
import math
print(math.ceil(4.2))
```

Output: 5

floor()

Rounds a number downward.

```
print(math.floor(4.8))
```

Output: 4

round()

Python also provides the built-in round() function:

```
print(round(4.6))
```

Output: 5

5.2 RANDOM MODULE

The random module is used for generating pseudo-random values.

Import:

```
import random
```

2.1 RANDOM NUMBERS

```
random.random()
```

Generates a random floating-point number from 0.0 up to but not including 1.0.

Example

```
import random  
print(random.random())
```

output: 0.735421

The exact value changes each time.

2.2 RANDOM INTEGERS

```
randint()
```

Generates a random integer between the given values, including both endpoints.

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

Possible output: 7

2.3 RANDOM FLOATING-POINT NUMBERS

```
uniform()
```

Generates a random floating-point number between two values.

```
import random  
print(random.uniform(1.5, 5.5))
```

output: 3.8271

2.4 CHOICE()

choice() selects one random item from a sequence.

```
import random  
fruits = ["Apple", "Banana", "Mango", "Orange"]  
print(random.choice(fruits))
```

Possible output: Mango

2.5 SAMPLE()

sample() selects multiple unique items without modifying the original sequence.

```
import random
numbers = [1, 2, 3, 4, 5]
print(random.sample(numbers, 3))
```

Possible output: [4, 1, 5]

2.6 SHUFFLE()

shuffle() randomly rearranges the items in a mutable sequence such as a list.

```
import random
numbers = [1, 2, 3, 4, 5]
random.shuffle(numbers)
print(numbers)
```

Possible output: [3, 5, 1, 4, 2]

The original list is modified.

2.7 RANDOM PASSWORD GENERATION

The random module can be combined with string to generate a simple random password.

```
import random
import string
characters = string.ascii_letters + string.digits + string.punctuation
password = ''.join(random.choice(characters) for _ in range(10))
print(password)
```

output: A7@kP2!xQ9

Note: For security-sensitive password or token generation, Python's secrets module is more appropriate than random.

5.3 DATETIME MODULE

The datetime module is used to work with:

- Dates

- Times

- Date and time

- Date arithmetic

- Time differences

- Import:

```
import datetime
```

3.1 CURRENT DATE AND TIME

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

Example:

2026-09-04 11:30:25.123456

The actual value depends on the current system date and time.

Current Date

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

3.2 DATE FORMATTING

The `strftime()` method formats date/time values into strings.

Common format codes:

Code	Meaning
%Y	Four-digit year
%m	Month
%d	Day
%A	Full weekday
%B	Full month

Example:

```
from datetime import datetime
now = datetime.now()
print(now.strftime("%d-%m-%Y"))
```

Possible output: 04-09-2026

3.3 TIME FORMATTING

Example:

```
from datetime import datetime
now = datetime.now()
print(now.strftime("%H:%M:%S"))
```

Possible output: 11:30:25

Here:

%H → hour

%M → minute

%S → second

3.4 DATE ARITHMETIC

We can add or subtract days from dates.

```
from datetime import date, timedelta
today = date.today()
future_date = today + timedelta(days=10)
```

```
print(future_date)
```

We can also subtract:

```
past_date = today - timedelta(days=10)
print(past_date)
```

3.5 TIMEDELTA

timedelta represents a duration or difference between dates/times.

Example:

```
from datetime import date, timedelta
today = date.today()
next_week = today + timedelta(weeks=1)
print(next_week)
```

Other useful arguments include:

```
timedelta(days=5)
timedelta(hours=3)
timedelta(minutes=30)
timedelta(seconds=20)
```

Finding Difference Between Dates

```
from datetime import date
date1 = date(2026, 9, 4)
date2 = date(2026, 9, 20)
difference = date2 - date1
print(difference.days)
```

Output: 16

5.4 SYS MODULE

The sys module provides access to variables and functions related to the Python interpreter and the system environment.

Import:

```
import sys
```

4.1 COMMAND-LINE ARGUMENTS

sys.argv stores command-line arguments.

Create:

```
test.py
import sys
print(sys.argv)
```

Run:

```
python test.py Amit 25
Possible output:
['test.py', 'Amit', '25']
```

Important:

sys.argv[0] → program name

sys.argv[1] → first argument

sys.argv[2] → second argument

Command-line arguments are initially stored as strings.

4.2 PYTHON VERSION

We can check the Python version using:

```
import sys
print(sys.version)
```

For a more structured version:

```
print(sys.version_info)
```

4.3 SYSTEM PATH

sys.path shows the locations where Python searches for modules.

```
import sys
print(sys.path)
```

It can also be used to inspect Python's module search path.

4.4 EXIT FUNCTIONS

The sys.exit() function terminates the program.

```
import sys
print("Program started")
sys.exit()
print("Program ended")
```

Output: Program started

The statement after sys.exit() is not executed.

4.5 STANDARD INPUT AND OUTPUT

Python provides standard input, output, and error streams.

Standard Input

```
import sys
name = sys.stdin.readline()
print(name)
```

Standard Output

```
import sys
sys.stdout.write("Hello Python")
```

Output: Hello Python

Normally, input() and print() are simpler for everyday programs.

5.5 STATISTICS MODULE

The statistics module provides functions for calculating common statistical measures.

```
Import:
import statistics
Suppose:
data = [10, 20, 30, 40, 50]
```

5.1 MEAN

The mean is the average of values.

Formula:

Mean = Sum of values / Number of values

Example:

```
import statistics
numbers = [10, 20, 30, 40, 50]
print(statistics.mean(numbers))
```

Output: 30

5.2 MEDIAN

The median is the middle value when data is arranged in order.

Example:

```
import statistics
numbers = [10, 20, 30, 40, 50]
print(statistics.median(numbers))
```

Output: 30

For an even number of observations, the median is the average of the two middle values.

5.3 MODE

The mode is the most frequently occurring value.

Example:

```
import statistics
numbers = [1, 2, 2, 3, 4]
print(statistics.mode(numbers))
```

Output: 2

5.4 VARIANCE

Variance measures how much values vary from their mean.

Example:

```
import statistics
numbers = [10, 20, 30, 40, 50]
print(statistics.variance(numbers))
```

By default, `statistics.variance()` calculates sample variance.

For population variance:

```
print(statistics.pvariance(numbers))
```

5.5 STANDARD DEVIATION

Standard deviation measures the spread of data around the mean.

Example:

```
import statistics
numbers = [10, 20, 30, 40, 50]
print(statistics.stdev(numbers))
```

For population standard deviation:

```
print(statistics.pstdev(numbers))
```

Relationship
Standard Deviation = $\sqrt{\text{Variance}}$

5.6 CORRELATION — INTRODUCTION

Correlation describes the strength and direction of a relationship between two variables.

The statistics module also provides functions for working with relationships between numerical datasets.

Correlation generally ranges from: -1 to +1

Correlation	Meaning
+1	Perfect positive relationship
0	No linear relationship
-1	Perfect negative relationship

Example

Suppose:

Hours Studied → Exam Marks

If students who study more generally get higher marks, the variables may have a positive correlation.

Important Note

The basic statistics module is useful for common descriptive statistics, but correlation analysis in practical data-analysis work is often performed using tools such as NumPy, Pandas, or other statistical libraries.

Practice Questions

1. What is a module in Python?
2. Why do we use modules?
3. What are the different types of modules in Python?
4. What is the difference between built-in, user-defined, and third-party modules?
5. Write the syntax to import a module using the import statement.
6. What is the difference between import module and from module import name?
7. What is the purpose of import ... as?
8. How do you create and use a user-defined module?
9. What is a package in Python?
10. What is the purpose of the __init__.py file?
11. What is the difference between a module and a package?
12. Write a program using the math module to find the square root of a number.
13. Write a program using the random module to generate a random number.
14. Write a program using the datetime module to display the current date and time.
15. Write a program using the statistics module to find the mean and median of a list of numbers.
16. What is the purpose of the sys module?
17. What is sys.path?
18. Write a program using random.choice() to select a random item from a list.
19. What is the difference between random.sample() and random.shuffle()?
- 20.** Write a program to calculate the mean, median, and standard deviation using the statistics module.



External Python Libraries

External Python libraries are collections of pre-written Python code that provide ready-to-use functionality.

External or third-party libraries are Python libraries created by developers and installed separately. They provide ready-made tools for tasks such as **numerical computing, data analysis, visualization, and machine learning**.

External Python libraries are libraries that are created by other developers and usually need to be installed before use.

They help developers perform tasks such as:

- Numerical calculations
- Data analysis
- Data visualization
- Machine learning
- Data preprocessing

Some important libraries for Data Analysts and Data Scientists are:

- NumPy → Numerical Computing
- Pandas → Data Analysis
- Matplotlib → Data Visualization
- Seaborn → Statistical Visualization
- Scikit-learn → Machine Learning

9.1 NUMPY – NUMERICAL COMPUTING

NumPy (Numerical Python) is the foundational open-source Python library used for scientific computing, data science, and machine learning. At its core is the ndarray object, a highly optimized, multi-dimensional array of homogeneous data types that runs up to 50 times faster than standard Python lists. [[1](#), [2](#), [3](#)]

NumPy is used for working with numbers and arrays.

NumPy stands for **Numerical Python**.

It is mainly used for:

- Numerical calculations
- Arrays
- Mathematical operations
- Statistical calculations
- Matrix operations
- Scientific computing

Installation

```
pip install numpy
```

Import

The standard convention is:
`import numpy as np`

1.1 ARRAYS

An array is a collection of elements stored in a structured form.
NumPy provides the ndarray object for working with arrays.

Creating an Array

```
import numpy as np
numbers = np.array([10, 20, 30, 40, 50])
print(numbers)
```

Output: [10 20 30 40 50]

Checking Array Type

```
print(type(numbers))
```

Output: <class 'numpy.ndarray'>

Checking Shape

```
print(numbers.shape)
```

Output: (5,)

2D Array

A two-dimensional array can be represented like a table.

```
import numpy as np
data = np.array([
    [10, 20, 30],
    [40, 50, 60]
])
print(data)
```

Output:
[[10 20 30]
 [40 50 60]]

Check its shape:

```
print(data.shape)
```

Output: (2, 3)

This means:

2 rows
3 columns

1.2 ARRAY OPERATIONS

NumPy allows us to perform operations directly on arrays.

Addition

```
import numpy as np
```

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

Output: [11 22 33]

Subtraction

```
print(a - b)
Output: [ 9 18 27]
```

Multiplication

```
print(a * b)
Output: [10 40 90]
```

Division

```
print(a / b)
Output: [10. 10. 10.]
```

Scalar Operations

We can perform an operation on every element using a single value.

```
numbers = np.array([10, 20, 30])
print(numbers * 2)
Output: [20 40 60]
```

1.3 MATHEMATICAL FUNCTIONS

NumPy provides many mathematical functions.

Square Root

```
import numpy as np
numbers = np.array([4, 9, 16, 25])
print(np.sqrt(numbers))
Output: [2. 3. 4. 5.]
```

Power

```
numbers = np.array([2, 3, 4])
print(np.power(numbers, 2))
Output: [ 4  9 16]
```

Absolute Value

```
numbers = np.array([-10, -20, 30])
print(np.abs(numbers))
Output: [10 20 30]
```

Exponential

```
print(np.exp([1, 2, 3]))
```

1.4 STATISTICAL FUNCTIONS

NumPy provides functions for basic statistical analysis.

Consider:

```
import numpy as np
marks = np.array([60, 70, 80, 90, 100])
```

Sum

```
print(np.sum(marks))
```

Output: 400

Mean

```
print(np.mean(marks))
```

Output: 80.0

Median

```
print(np.median(marks))
```

Output: 80.0

Minimum

```
print(np.min(marks))
```

Output: 60

Maximum

```
print(np.max(marks))
```

Output: 100

Standard Deviation

```
print(np.std(marks))
```

Variance

```
print(np.var(marks))
```

1.5 BROADCASTING

Broadcasting is a NumPy feature that allows operations between arrays of compatible shapes or between an array and a scalar without manually changing every element.

Example:

```
import numpy as np
numbers = np.array([10, 20, 30])
print(numbers + 5)
```

Output: [15 25 35]

Here, NumPy effectively applies 5 to every element:

$10 + 5 = 15$

$20 + 5 = 25$

$30 + 5 = 35$

Broadcasting with Different Shapes

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

```
b = np.array([10, 20, 30])
print(a + b)
Output:
[[11 22 33]
 [14 25 36]]
```

The one-dimensional array is broadcast across the rows because the shapes are compatible.

9.2 PANDAS – DATA ANALYSIS

Pandas is a Python library mainly used for:

- Data analysis
- Data cleaning
- Data transformation
- Data manipulation
- Working with tabular data
- Reading and writing files

It is one of the most important libraries for Data Analysts.

Installation

```
pip install pandas
```

Import

```
import pandas as pd
```

Pandas is used for working with structured data.

- Series
- DataFrames
- Data cleaning
- Filtering
- Grouping
- Merging datasets

2.1 SERIES

A Series is a one-dimensional labeled data structure.

Example:

```
import pandas as pd
marks = pd.Series([80, 70, 90, 85])
print(marks)
Output:
80
70
90
85
```

dtype: int64

The left side represents the index.

Creating Series with Custom Index

```
marks = pd.Series(  
    [80, 70, 90],  
    index=["Math", "Science", "English"]  
)  
print(marks)
```

2.2 DATA FRAME

A Data Frame is a two-dimensional labeled data structure consisting of rows and columns.

It is similar to a spreadsheet or database table.

Creating a Data Frame

```
import pandas as pd  
data = {  
    "Name": ["Kunal", "Amit", "Priya"],  
    "Age": [25, 26, 24],  
    "Marks": [85, 90, 88]  
}  
df = pd.DataFrame(data)  
print(df)
```

Output:

	Name	Age	Marks
0	Kunal	25	85
1	Amit	26	90
2	Priya	24	88

Series vs Data Frame

Feature	Series	DataFrame
Dimension	1D	2D
Structure	Single column-like object	Rows and columns
Example	pd.Series()	pd.DataFrame()
Similar to	Single column	Spreadsheet/table

2.3 DATA IMPORT & EXPORT

Pandas can read data from various file formats.

Reading CSV

```
import pandas as pd
```

```
df = pd.read_csv("employees.csv")
print(df)
```

Exporting to CSV

```
df.to_csv("output.csv", index=False)
```

index=False prevents Pandas from writing the DataFrame index as an extra column.

Reading Excel

```
df = pd.read_excel("employees.xlsx")
```

Exporting:

```
df.to_excel("output.xlsx", index=False)
```

2.4 DATA CLEANING

Data cleaning means identifying and correcting problems in a dataset.

Common problems include:

- Missing values
- Duplicate records
- Incorrect data types
- Invalid values
- Inconsistent data

Checking Missing Values

```
print(df.isnull())
```

Count missing values:

```
print(df.isnull().sum())
```

Removing Missing Values

```
df = df.dropna()
```

This removes rows containing missing values.

Filling Missing Values

```
df["Age"] = df["Age"].fillna(0)
```

We can also fill with the mean:

```
df["Age"] = df["Age"].fillna(df["Age"].mean())
```

Removing Duplicates

```
df = df.drop_duplicates()
```

2.5 FILTERING

Filtering allows us to select rows that satisfy a condition.

Suppose:

```
df = pd.DataFrame({
    "Name": ["Amit", "Priya", "Sonali"],
    "Age": [25, 30, 22],
```

```
"Salary": [30000, 50000, 40000]
})
```

Filter Age Greater Than 25

```
result = df[df["Age"] > 25]
print(result)
```

Multiple Conditions

Use:

& → AND

| → OR

~ → NOT

Example:

```
result = df[(df["Age"] > 22) & (df["Salary"] > 35000)]
print(result)
```

2.6 GROUP BY

groupby() is used to group data based on one or more columns and perform calculations on each group.

Example:

```
import pandas as pd
df = pd.DataFrame({
    "Department": ["IT", "HR", "IT", "HR"],
    "Salary": [50000, 40000, 60000, 45000]
})
result = df.groupby("Department")["Salary"].mean()
print(result)
```

Output:

```
Department
HR    42500.0
IT    55000.0
Name: Salary, dtype: float64
```

Common GroupBy Functions

```
df.groupby("Department")["Salary"].sum()
df.groupby("Department")["Salary"].mean()
df.groupby("Department")["Salary"].count()
df.groupby("Department")["Salary"].max()
df.groupby("Department")["Salary"].min()
```

2.7 MERGE & JOIN

Both merge() and join() are used to combine DataFrames.

Merge

merge() is commonly used to combine DataFrames based on matching columns.

Example:

```
import pandas as pd
employees = pd.DataFrame({
    "Emp_ID": [1, 2, 3],
    "Name": ["Amit", "Priya", "Kunal"]
})
salary = pd.DataFrame({
    "Emp_ID": [1, 2, 3],
    "Salary": [30000, 40000, 50000]
})
result = pd.merge(employees, salary, on="Emp_ID")
print(result)
```

Output:

	Emp_ID	Name	Salary
0	1	Amit	30000
1	2	Priya	40000
2	3	Kunal	50000

Join

join() commonly combines DataFrames using their indexes.

Example:

```
result = employees.join(salary.set_index("Emp_ID"), on="Emp_ID")
print(result)
```

Merge vs Join

Feature	merge()	join()
Common use	Combine using columns	Combine using indexes
Flexibility	More flexible	Convenient for index-based combining
SQL-like	Yes	Less SQL-like

9.3 MATPLOTLIB – DATA VISUALIZATION

Matplotlib is a Python library used to create graphs and charts.

It can create:

- Line charts
- Bar charts

Scatter plots
Histograms
Pie charts
Box plots

Installation

```
pip install matplotlib
```

Import

```
import matplotlib.pyplot as plt
```

3.1 LINE CHART

A line chart is useful for showing **trends over time**.

Example:

```
import matplotlib.pyplot as plt
months = ["Jan", "Feb", "Mar", "Apr"]
sales = [100, 120, 150, 140]
plt.plot(months, sales)
plt.xlabel("Month")
plt.ylabel("Sales")
plt.title("Monthly Sales")
plt.show()
```

Common uses:

Sales trends
Revenue trends
Stock prices
Temperature changes

3.2 BAR CHART

A bar chart compares values across categories.

```
import matplotlib.pyplot as plt
products = ["A", "B", "C"]
sales = [100, 150, 120]
plt.bar(products, sales)
plt.xlabel("Product")
plt.ylabel("Sales")
plt.title("Product Sales")
plt.show()
```

Common uses:

Product comparison
Department comparison

SCATTER PLOT

A scatter plot shows the relationship between two numerical variables.

```
import matplotlib.pyplot as plt
hours = [1, 2, 3, 4, 5]
marks = [50, 55, 65, 75, 85]
plt.scatter(hours, marks)
plt.xlabel("Study Hours")
plt.ylabel("Marks")
plt.title("Study Hours vs Marks")
plt.show()
```

Scatter plots are useful for identifying:

- Relationships
- Patterns
- Clusters
- Possible outliers

3.4 HISTOGRAM

A histogram shows the distribution of numerical data.

```
import matplotlib.pyplot as plt
ages = [20, 22, 22, 25, 26, 27, 27, 30, 32, 35]
plt.hist(ages, bins=5)
plt.xlabel("Age")
plt.ylabel("Frequency")
plt.title("Age Distribution")
plt.show()
```

A histogram groups numerical values into intervals called **bins**.

3.5 PIE CHART

A pie chart shows how a whole is divided among categories.

```
import matplotlib.pyplot as plt
labels = ["IT", "HR", "Sales"]
values = [50, 20, 30]
plt.pie(values, labels=labels, autopct="%1.1f%%")
plt.title("Department Distribution")
plt.show()
```

Pie charts are useful when showing **parts of a whole**, especially for a small number of categories.

3.6 BOX PLOT

A box plot is used to understand the distribution and spread of numerical data.

It can help identify:

- Median

- Quartiles

- Spread

- Potential outliers

Example:

```
import matplotlib.pyplot as plt
salary = [25000, 30000, 32000, 35000, 40000, 45000, 100000]
plt.boxplot(salary)
plt.title("Salary Distribution")
plt.show()
```

The value 100000 may appear as a potential outlier depending on the data distribution.

9.4 SEABORN – STATISTICAL VISUALIZATION

Seaborn is a Python visualization library built on top of Matplotlib.

It provides a simpler interface for many statistical visualizations and works especially well with Pandas DataFrames.

Seaborn is used to create statistical and attractive data visualizations.

- Count Plot

- Heatmap

- Pair Plot

- Distribution Plot

- Violin Plot

Installation

```
pip install seaborn
```

Import

```
import seaborn as sns
```

4.1 COUNT PLOT

A count plot displays the number of observations in each category.

Example:

```
import seaborn as sns
import matplotlib.pyplot as plt
data = {
    "Department": ["IT", "HR", "IT", "Sales", "HR", "IT"]
}
sns.countplot(x="Department", data=data)
```

```
plt.show()
```

In practice, Seaborn generally expects tabular data such as a Pandas DataFrame, so a common version is:

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
df = pd.DataFrame({
    "Department": ["IT", "HR", "IT", "Sales", "HR", "IT"]
})
sns.countplot(x="Department", data=df)
plt.show()
```

4.2 HEATMAP

A heatmap represents values using different visual intensities.

It is commonly used to display:

- Correlation matrices
- Frequency tables
- Performance data

Example:

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
df = pd.DataFrame({
    "Age": [20, 25, 30, 35],
    "Salary": [25000, 35000, 45000, 55000],
    "Experience": [1, 2, 5, 8]
})
correlation = df.corr()
sns.heatmap(correlation, annot=True)
plt.show()
annot=True displays the values inside the cells.
```

4.3 PAIR PLOT

A pair plot shows relationships between multiple numerical variables.

```
import seaborn as sns
import matplotlib.pyplot as plt
df = sns.load_dataset("iris")
sns.pairplot(df)
plt.show()
```

Pair plots are useful for:

Exploring relationships
Identifying patterns
Comparing multiple variables
Exploring datasets before machine learning

4.4 DISTRIBUTION PLOT

A distribution plot helps visualize how numerical values are distributed. Modern Seaborn commonly uses `histplot()` or `displot()` for distribution visualization.

Example:

```
import seaborn as sns
import matplotlib.pyplot as plt
df = sns.load_dataset("tips")
sns.histplot(data=df, x="total_bill", kde=True)
plt.show()
```

Here:

Histogram shows frequency distribution.
`kde=True` adds a smooth density curve.

4.5 VIOLIN PLOT

A violin plot shows the distribution and density of numerical data across categories.

Example:

```
import seaborn as sns
import matplotlib.pyplot as plt
df = sns.load_dataset("tips")
sns.violinplot(x="day", y="total_bill", data=df)
plt.show()
```

Violin plots are useful for comparing the distribution of a numerical variable across different categories.

9.5 SCIKIT-LEARN – MACHINE LEARNING

Scikit-learn is a popular Python library for machine learning.

It provides tools for:

- Data preprocessing
- Classification
- Regression
- Clustering
- Model selection
- Feature scaling

- Model evaluation

Scikit-learn is used to build and evaluate machine learning models.

- Data preprocessing
- Train-test split
- Feature scaling
- Building ML models
- Model evaluation

Installation

```
pip install scikit-learn
```

Import Example

```
from sklearn.model_selection import train_test_split
```

Note:

The package is installed using:

scikit-learn

but it is imported using:

sklearn

5.1 DATA PREPROCESSING

Data preprocessing prepares raw data before it is given to a machine learning model.

Common preprocessing tasks include:

Handling missing values

Encoding categorical data

Feature scaling

Removing or handling invalid values

Splitting data

Example: Standardization

```
from sklearn.preprocessing import StandardScaler
```

```
scaler = StandardScaler()
```

```
X_scaled = scaler.fit_transform(X)
```

5.2 TRAIN-TEST SPLIT

A dataset is commonly divided into:

Training data

Testing data

Training data is used to train the model.

Testing data is used to evaluate the model on unseen data.

Example:

```
from sklearn.model_selection import train_test_split
```

```
X_train, X_test, y_train, y_test = train_test_split(
```

```
X,  
y,  
test_size=0.2,  
random_state=42  
)
```

Here:

X → Features
y → Target
test_size=0.2 → 20% testing data
80% → Training data
random_state=42 → Reproducible split

5.3 FEATURE SCALING

Feature scaling brings numerical features to comparable scales.
It is especially important for algorithms that are sensitive to feature magnitude.
Two commonly used techniques are:

- Standardization
- Min-Max Scaling

3.1 Standardization

StandardScaler transforms data so that features are generally centered around mean 0 with unit variance.

```
from sklearn.preprocessing import StandardScaler  
scaler = StandardScaler()  
X_train_scaled = scaler.fit_transform(X_train)  
X_test_scaled = scaler.transform(X_test)
```

Notice:

Training data → fit_transform()

Testing data → transform()

This prevents information from the test set from being used when fitting the scaler.

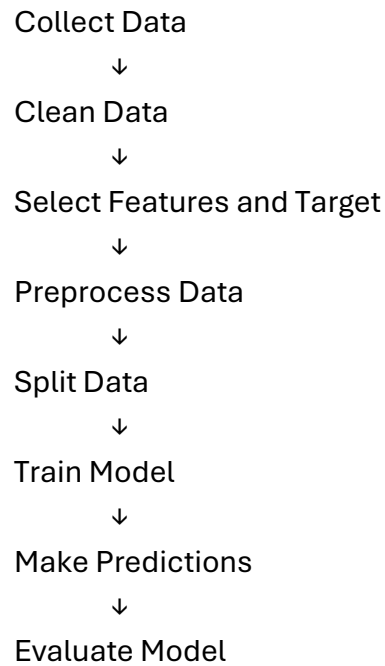
3.2 Min-Max Scaling

MinMaxScaler scales values to a specified range, commonly 0 to 1.

```
from sklearn.preprocessing import MinMaxScaler  
scaler = MinMaxScaler()  
X_scaled = scaler.fit_transform(X)
```

5.4 BUILDING ML MODELS

A general machine learning workflow is:



Simple Example

```
from sklearn.linear_model import LinearRegression
model = LinearRegression()
model.fit(X_train, y_train)
predictions = model.predict(X_test)
```

Here:

model.fit() → trains the model
model.predict() → generates predictions

5.5 MODEL EVALUATION

Model evaluation measures how well a machine learning model performs. Different types of models use different evaluation metrics.

Regression Metrics

Common metrics include:

MAE
MSE
RMSE
 R^2 Score

Example:

```
from sklearn.metrics import mean_absolute_error
mae = mean_absolute_error(y_test, predictions)
print(mae)
```

5.2 Classification Metrics

Common classification metrics include:

- Accuracy
- Precision
- Recall
- F1-score
- Confusion Matrix

Example:

```
from sklearn.metrics import accuracy_score
accuracy = accuracy_score(y_test, predictions)
print(accuracy)
```

Complete Library Comparison

Library	Main Purpose	Common Examples
NumPy	Numerical computing	Arrays, mathematical operations
Pandas	Data analysis	DataFrame, cleaning, filtering
Matplotlib	Data visualization	Line, bar, scatter
Seaborn	Statistical visualization	Heatmap, violin, pair plot
Scikit-learn	Machine learning	Preprocessing, models, evaluation

Quick Revision

NumPy

```
import numpy as np
arr = np.array([10, 20, 30])
print(np.mean(arr))
print(np.max(arr))
print(np.sqrt(arr))
```

Pandas

```
import pandas as pd
df = pd.read_csv("data.csv")
print(df.head())
print(df.isnull().sum())
```

Matplotlib

```
import matplotlib.pyplot as plt
plt.plot([1, 2, 3], [10, 20, 30])
plt.show()
```

Seaborn

```
import seaborn as sns
```

```
sns.heatmap(df.corr(), annot=True)
```

Scikit-learn

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
X, y, test_size=0.2, random_state=42
)
```

Practice Questions

NumPy

1. What is NumPy and why is it used?
2. What is a NumPy array?
3. Create a NumPy array containing five numbers.
4. Write a program to perform addition, subtraction, multiplication, and division on NumPy arrays.
5. Write a program to find the **sum, mean, maximum, and minimum** of a NumPy array.
6. What is broadcasting in NumPy?

Pandas

7. What is Pandas and why is it used?
8. What is the difference between a **Series** and a **DataFrame**?
9. Create a Pandas DataFrame containing student names and marks.
10. Write a program to read data from a CSV file.
11. How do you check and handle missing values in a DataFrame?
12. Write a program to filter rows based on a condition.
13. What is the purpose of `groupby()` in Pandas?
14. What is the difference between `merge()` and `join()`?

Matplotlib and Seaborn

15. Write a program to create a **line chart**.
16. Write a program to create a **bar chart**.
17. Write a program to create a **histogram**.
18. What is a **scatter plot** used for?
19. Write a program to create a **pie chart**.
20. What is the purpose of a **heatmap** in Seaborn?

Scikit-learn

21. What is Scikit-learn used for?
22. What is data preprocessing?
23. What is the purpose of `train_test_split()`?
24. What is feature scaling?
25. What are the common steps involved in building and evaluating a machine learning model?

